Chapter 3
Dimensions of the Time Modeling Problem

Modeling is all about abstraction: select which aspects should be included in the model, the details of their descriptions, and the form the descriptions should take. Models of *time*, in particular, must consider several distinctive issues that have to do with the nature of time and how it is represented. This chapter presents these *dimensions* of the time modeling problem within the general framework of the book.

Some of the dimensions denote issues that are pervasive in the modeling of time in the literature, for example, the use of discrete or continuous time domains. Others shed light on aspects specific to some classes of formalisms, for example, the presence of an explicit or implicit reference to time.

The dimensions will guide the presentation and comparison in the following chapters of how the various notations model time; they will focus the presentation on the most significant instances and equip readers with references and skills useful for analyzing any other formalism that includes some notion of time, beyond those detailed in this book. Correspondingly, the dimensions can guide a knowledgeable choice – and possibly a tailoring – of the notation most appropriate to specific modeling needs (as will be briefly discussed in the epilogue).

The "dimensions" of this chapter, however, informally refer to aspects that are neither necessarily exhaustive nor independent. Unlike the dimensions in an orthonormal mathematical basis, some dimensions of time modeling depend on each other, with the result that only certain combinations are sometimes possible, reasonable, or relevant in practice. The following chapters will illustrate the dependencies among different dimensions in several concrete examples.

3.1 Discrete Versus Dense Time Domains

A first natural categorization of the formalisms dealing with time-dependent systems is between the use of discrete and dense sets as domains for the "time variable".

C.A. Furia et al., *Modeling Time in Computing*, Monographs in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-642-32332-4_3,

Recall that a discrete set consists of isolated points whereas a dense set, ordered by "$<$", is such that for every two points t_1, t_2, with $t_1 < t_2$, there is always another point t_3 in between: $t_1 < t_3 < t_2$. In the scientific literature and applications, the most widely adopted discrete time models are natural and integer numbers – denoted by $\mathbb{N}$ and $\mathbb{Z}$, respectively – whereas the typical dense models are rational and real numbers – denoted by $\mathbb{Q}$ and $\mathbb{R}$, respectively. For instance, differential equations normally assume the real (or even the complex) numbers as variable domains, whereas difference equations are defined over integers. Computing devices are formalized through discrete models when their behavior is paced by a clock, so that it is natural to measure time by counting clock ticks.

In addition to the well-known classification into discrete and dense domains, a few more accurate distinctions are useful for better evaluating and comparing the various formalisms available in the literature and those that will be proposed in the future.

3.1.1 Continuous Versus Non-continuous Time Models

Dense domains include both continuous and non-continuous sets. For some models, the distinction is relevant and must be considered.

The notion of continuous domain originated from the observation that there exist *incommensurable* physical quantities: two values v_1, v_2 are incommensurable if there exist no integers n, m such that $n \cdot v_1 = m \cdot v_2$; hence the ratio v_1/v_2 is not a rational number in the dense non-continuous set $\mathbb{Q}$. For example, the diameter and the circumference of every circle are incommensurable, and the *irrational* number π denotes their constant ratio. Other irrational numbers are introduced to denote the results of operations naturally applicable to every rational number whose results are not rational, such as the square root of 2. The extension of a dense non-continuous set such as $\mathbb{Q}$ with all irrational numbers gives a *continuous* domain; the real and complex numbers are the most widely known and used continuous domains.

The problem with incommensurable quantities is relevant also when measuring time: the periods of two clocks that are not perfectly synchronous are likely incommensurable. We do not have to look for contrived examples of this phenomenon: the solar day and year are indeed incommensurable time spans. Adopting a continuous set as the time domain makes it possible to model incommensurable times precisely, thus making the analysis more general and uniform; for example, showing that a model has behaviors with certain characteristics may be simpler under the assumption of a continuous time domain.

On the other hand, the greater generality of continuous domains becomes an obstacle when performing numerical and algorithmic analyses of the models, because irrational numbers (which are the overwhelming majority in a continuous set) have no finite representation as series of digits; hence a digital computer can only rely on *approximations* of their exact values in terms of rational numbers. The finite precision of the approximations must allow for the computation of

solutions with an error that is acceptable for the application domain. For example, the incommensurability of day and year requires an approximation to construct calendars. The simple convention of approximating 1 year to 365 days introduces a considerable drift, which accumulates and becomes unacceptable after only a few years; the Julian calendar introduced a more precise approximation using a leap year every 4 years; the Gregorian calendar further refined the approximation (years that are exactly divisible by 100 are not leap years unless they are exactly divisible by 400) but still introduces an error of 1 day every few thousand; in general, every approximation introduces a drift between calendar and astronomical day after a sufficiently long period of time.

Another, more specific, context in which the distinction between continuous and merely dense time domains is relevant is the algorithmic analysis of timed models: some sophisticated time analysis algorithms work correctly only under the restriction that certain time parameters of the model are rational. We will mention examples of such algorithms when discussing timed automata in Chap. 7 and Petri nets in Chap. 8.

3.1.2 Bounded, Finite, and Periodic Time Models

System modeling often assumes behaviors that may proceed indefinitely in the future (and maybe in the past), so it is natural to model time as an unbounded set. This typically complicates the analysis of system properties, which may become undecidable[1] in the general case because no observation over a finite amount of time can be conclusive about the longer-term behavior (see Sect. 3.8.2 for more comments on the aspect of decidability).

There are significant cases, however, where all relevant system behavior can be a priori enclosed within a *bounded* "time window". For instance, braking a car to a full stop requires at most a few seconds; thus, if we want to model and analyze the behavior of an antilock braking system, there is no loss of generality if we assume as a temporal domain, say, the real range $[0, 60]$ seconds. In many cases, a restriction to bounded time highly simplifies algorithmic analysis and simulation.

When a domain is not only bounded but also discrete, it becomes *finite*. For systems where the time domain and every other domain are finite, all system properties are, in principle, decidable, because behavioral analysis reduces to the enumeration of a finite number of system configurations. If a domain is not discrete but only bounded, its *discretization* – that is, the discrete approximation of its values – may support an exhaustive analysis of system behavior that is precise

[1] A property is decidable if there exists an algorithmic procedure that can determine, in finite time, whether the property holds in any given system model; otherwise, it is undecidable. Chapter 6 introduces the notion with more precision for readers unfamiliar with the theory of computability.

enough to replace the exact analysis on dense domains. Section 3.1.3 discusses the widely used *sampling* technique to achieve discretization.

A special case of unbounded behavior occurs when a system is *periodic*, that is, recurrently returns to certain states during its evolution. The time between two consecutive visits to a repeated state is called *period*. Since the evolution over an unbounded time domain consists entirely of infinite repetitions of the period, the analysis of a periodic system's behavior reduces to the analysis over bounded time: the properties holding over the whole time domain follow from the behavior over a single finite period.

The well-known problem of studying the termination of computer programs illustrates how periodicity can simplify timing analysis. Determining whether a generic program halts for a given input boils down to timing analysis: "determine if there exists a time t such that the program, run with the given input, stops after t time units". Termination is undecidable in the general case: the best we can do is run the program with the input, but then we can never conclude that it will not halt in the future if it has not halted after a finite (arbitrarily large) amount of time. If, however, we observe that the system behavior is periodic, then there exists a period Δ such that the state of the computation – which comprises the memory and the input – is the same at all times $t_1, t_1 + \Delta, t_1 + 2\Delta, t_1 + 3\Delta$, and so on indefinitely. If termination only depends on the state, nontermination over a single period entails nontermination everywhere, because the behavior over a period characterizes the overall behavior.

Section 3.8.3 mentions several analysis techniques that rely on periodicity and finiteness of behaviors and domains to achieve automation by means of exhaustive enumeration. Bounded model checking, presented in Chap. 11, is a prime example of such techniques with significant practical impact.

3.1.3 Hybrid Systems

The discussion about dense vs. discrete domains of the present section focuses on time, but system models must select discrete or dense domains also for other state components and variables. Chapter 2, for example, presented some models of physical systems where all domains – e.g., time, speed, and position – are continuous. Computing systems, in contrast, are usually modeled with discrete time (paced by the clock) and state (sequences of digital bits).

Combining discrete time with discrete state variables and dense time with dense state domains is a common choice, but alternatives exist: *hybrid* system models combine discrete and dense domains. All combinations are possible: discrete time and dense state space, dense time and discrete state space, and even cases where the time model integrates discrete and dense components of time, or discrete and dense state domains coexist. Indeed, there are several circumstances in which hybrid models are the natural choice; they are mainly, but not exclusively, related to the problem of integrating heterogeneous components.

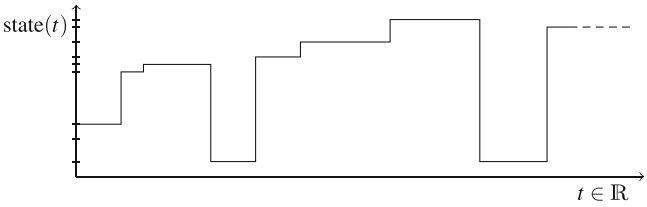

Fig. 3.1 A square-wave form over dense time

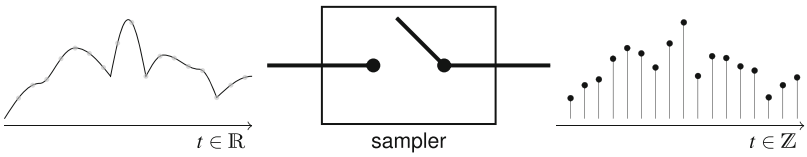

Fig. 3.2 A continuous behavior sampled

A typical example is a system consisting of a power plant controlled by some digital computing device. Differential equations on continuous time are a natural model for the physical process (for example, a chemical reaction or the production of electric power). The digital computer has a discrete, possibly finite, state space over a discrete time domain. The two components interact through devices such as samplers and holders.

More generally, dense and discrete domains coexist in hybrid models in different ways:

- Components with a discrete – possibly finite – set of states evolve over a dense time domain. In such cases, behaviors are graphically described as *square wave* forms and evolve as piecewise constant functions of time, as shown in Fig. 3.1.
- Sampling the state at regular (discrete) intervals provides an approximation of the behavior of state variables over a continuous time domain, as pictured in Fig. 3.2. A classic problem of control and information theory is how to guarantee that the approximation introduced by sampling does not lose any relevant information about the continuous-time behavior. Some sections of Chaps. 7 and 9 discuss the role of sampling techniques for operational and descriptive temporal models.
- The *time domain* is hybrid when it consists of a discrete sequence of "macro-steps", but between each pair of discrete steps there exist finer-grain dynamics modeled over dense time. This setting accommodates, for example, the abstraction of electronic components into logic gates – discussed in Chap. 5 – as well as discrete-time systems with time-outs that can occur asynchronously. Finite-state automata augmented with dense-timed clock variables, such as the timed and hybrid automata of Chap. 7, are more general examples of this type of hybridism.

Example 3.1. The braking system mentioned in Example 2.3 is a hybrid model: the system senses the information coming from the wheels in the form of variables

varying over continuous time and state domains (e.g., angular speed and friction) and processes it through a digital embedded device that computes, in real time, the ideal pressure to be applied to the calipers. Processing the continuous components with a digital system – with finite precision and synchronized to a discrete clock – requires sampling and approximating the information coming from the sensors; then, the actuators translate the discrete series of values output by the computer to the calipers, which operate over "physical" continuous time. ■

We will go back to the issue of hybrid models in Sect. 3.7, when discussing composition of modules.

Exercise 3.2 (♣). Which of the following systems or processes are naturally described by a hybrid model? What are the discrete and the dense/continuous components? To which of the three aforementioned classes of hybrid models do the systems naturally belong?

- A thermostat controlling the temperature of a room by turning on and off heating.
- The controller of a railroad junction.
- The emission of light from a heated chemical element. ■

3.2 Ordering Versus Metric

A formalism may permit the expression of metric constraints on time, or, equivalently, of constraints that exploit the metric structure of the underlying time model (if it has any).

A domain (possibly a time domain) has a *metric* when it is equipped with a notion of *distance*, that is a *measure* function $d(t_1, t_2)$ associated with pairs of points t_1, t_2 of the domain that satisfies the properties of

(i) Nonnegativity: $d(t_1, t_2) \geq 0$;
(ii) Identity of indiscernibles: $d(t_1, t_2) = 0$ if and only if $t_1 = t_2$;
(iii) Symmetry: $d(t_1, t_2) = d(t_2, t_1)$;
(iv) Subadditivity (also called triangle inequality): $d(t_1, t_3) \leq d(t_1, t_2) + d(t_2, t_3)$.

The typical time domains – the usual discrete and dense numerical sets $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ – all have a "natural" metric in terms of Euclidean distance between two points: $d(t_1, t_2) = |t_1 - t_2|$.

Although all common choices for time domains possess a metric, an issue is whether the *language* in which the system is described permits using the same form of metric information as that embedded in the underlying time domain. For instance, some languages allow for stating that an event p (e.g., "push button") must precede temporally another event q (e.g., "take picture"), but do not include constructs to specify how much time elapses between the occurrence of p and that of q; thus, they cannot distinguish between the case in which the delay between p and q is one time unit from the case in which the delay is 100 time units. Languages where the

relative *ordering* of events is expressible, but metric constraints are not, support a purely *qualitative* notion of time, as opposed to the *quantitative* time expressible with metric languages.

Example 3.3 (Parallel and real-time systems). In purely parallel systems, the correctness of a computation only depends on the *relative ordering* of computational steps, irrespective of their absolute distances. Reactive systems, where a controller component evolves concurrently with the controlled environment, are often purely parallel in this sense. For the formal description of such systems, a purely qualitative language is sufficient. Real-time systems also usually perform in parallel, but their correctness depends as well on the *time distance* between events; thus, the complete model of real-time systems requires quantitative languages, supporting the expression of metric constraints.

As a simple example of purely parallel system, consider two tasks T_1 and T_2 that exchange messages. T_1 can perform an action a only after receiving a datum from T_2; and T_2 produces the datum by performing another action b. This data dependency forces the ordering of actions a and b: a follows one or more occurrences of b, independently of the relative speed of the two tasks or of the transmission channel.

The same two-task system becomes real time if T_2 produces data at a fixed rate of n actions b per second and puts them in a buffer composed of a single slot, and we want to avoid the situation where T_2 tries to put a datum in a full buffer. T_1 then has to be fast enough: a specification of correct behavior may require, in addition to actions a following actions b, that no execution of a takes more than $1/n$ seconds. ■

Following the difference between purely parallel and real-time systems, the research in the field of formal languages for system description has evolved from dealing with purely qualitative models to the more difficult task of expressing and reasoning about metric constraints. Consider, for instance, two sequences σ_1 and σ_2 of events p and q, where exactly one event per time step occurs,

$$\sigma_1 = p\,q\,p\,q\,p\,q\,\ldots,$$
$$\sigma_2 = p\,p\,q\,q\,p\,p\,q\,q\,\ldots,$$

that share the following property, expressible without referencing any metric information: "every occurrence of p is eventually followed by an occurrence of q"; in contrast, "p occurs in every instant" is a qualitative property that is false for both behaviors. Some metric properties, instead, discriminate between σ_1 and σ_2, as in "every occurrence of q is followed by another occurrence of q after two time steps", which holds for σ_1 but not for σ_2.

The notion of *invariance under stuttering* is an alternative characterization of the properties expressible qualitatively. Consider, for example, the discrete-time behavior σ_3 consisting of the following sequence of states, one per time step:

$$\sigma_3 = s_1\,s_2\,s_3\,s_4\,s_5\,s_6\,s_7\,s_8\,\ldots.$$

A time step i such that $s_i = s_{i+1}$ is called a "stuttering" step; for example, the first step in sequence σ_2 is stuttering (if we interpret it as a sequence of states rather than as a sequence of events). Adding or removing stuttering steps from a behavior does not affect the qualitative properties it satisfies. If two behaviors are identical up to the addition or removal of any number of stuttering steps, the two behaviors are called *stutter-equivalent* or *equivalent under stuttering*. For example, behaviors σ_1, σ_2 above are stutter-equivalent: every odd instant of time corresponds to a stuttering step in σ_2, and σ_1 equals σ_2 with all stuttering steps removed.

Stutter equivalence is an equivalence relation; the equivalence classes it induces precisely identify classes of behaviors that share identical qualitative properties. Note that stutter invariance is defined for discrete time models only.

Exercise 3.4 (♣). Argue that the property

"sequences of events a and sequences of events b alternate" (3.1)

is qualitative, whereas the property

"events a and b *strictly* alternate" (3.2)

is quantitative. ■

Exercise 3.5. With reference to Exercise 3.4, characterize the set of behaviors corresponding to (3.1) and show that any two members of the set are stutter-equivalent. Then, characterize the set of behaviors corresponding to (3.2) and show that there exist behaviors which are stutter-equivalent to elements of the set but are not in the set. ■

3.2.1 Total Versus Partial Ordering

The discussion so far assumed time and other domains with a *total* ordering: for every pair of distinct elements x, y in the domain, either x precedes y in the ordering (usually written $x < y$), or y precedes x ($y < x$). The definition of dense sets, in particular, is simpler for totally ordered domains, and so is the definition of a metric. There are circumstances, however, where sets with only partial ordering – where neither $x < y$ nor $y < x$ for some distinct elements x, y – are the best choice for the temporal domain in a system model.

Example 3.6. Modern cars implement several functions on their on-board embedded electronics as software. The antilock braking system mentioned in Sect. 3.1.2 is a common example; another subsystem electronically controlled is the one responsible for moving the car windows. Each subsystem must meet its timing requirements: among other things, the braking system must release the breaks within, say, $1/10$ second whenever the wheels are blocked and the vehicle is

moving, and the motorized windows must shut completely within, say, 7 s whenever a passenger clicks the button. In an overall model of the car, the events "wheels become blocked" and "breaks released" are strictly ordered, and so are the two other events "button clicked" and "windows become closed". However, there is no reason to define an order between events of the braking subsystem and events of the window control: the events in the overall system are only partially ordered, and so are the instants of time when they may occur. ■

The above example suggests that partial orderings arise naturally when composing the behavior of subsystems into composite systems with heterogeneous components: the events happening in different subsystems are usually unrelated, and synchronization among subsystems relies on explicit "messages" sent at the subsystems' interface. Part II of the book will present some notations that introduce partial orders when composing unrelated events, as well as others that always define a total order among events. Section 3.3 discusses another dimension that relies on the notion of total and partial ordering.

3.2.2 Time Granularity

System models with metric time usually possess a "natural" time scale, corresponding to the abstraction level of the temporal behavior in the model. In Example 3.6, the braking system operates within fractions of seconds, whereas the window system is paced by an order of magnitude slower time scale. The notion of *time granularity* captures this idea of "time scale", and different components in a composite system have different time granularities when their natural time scales differ, possibly by orders of magnitude.

In some sense, time granularity is a form of hybridism (see Sect. 3.1.3), which is frequent in complex composite systems where processes that evolve in the order of seconds or minutes – or even days or months (such as a chemical process, or a process at a hydroelectric power plant) – are controlled by fast digital electronic devices. In principle, a continuous time domain, such as the real numbers, can accommodate system models with arbitrarily heterogeneous time granularities: conversion among different time units is always possible, with possibly an arbitrarily small loss of precision if some units happen to be incommensurable (see Sect. 3.1.1).

If, however, the underlying time domain is discrete, the approximation error introduced when converting the coarser time units can be non-negligible and raise subtle semantic issues. Consider, for instance, the sentences

> Every month, if an employee works, then she gets her salary.

and

> Whenever an employee is assigned a job, this job should be completed within three days.

If the sentences are part of the same specification of an office system, we have to find a way to reconcile their time units. A discrete temporal domain with the day as time unit seems a natural choice, because the other time unit, the month, is of coarser granularity. However, a simple change of time units from months to days alters the meaning of the quantification "every": the specification "every month, if an employee works, then she gets her salary" has a different meaning than "every day, if an employee works, then she gets her salary", because working for 1 month means working for 22 variable days during the month, whereas getting a monthly salary means that there is one fixed customary day of every month when salaries for the whole month get paid. A change in the time unit (from months to days) is insufficient to capture the correct meaning of the original sentence.

In the other example, your boss states that "this job has to be finished within three days from now" at 4 P.M. on 16 June 2012. What does she mean exactly? "This job has to be finished within $3 \cdot 24 \cdot 60 \cdot 60$ seconds counting from now", or "this job has to be finished by 6 P.M. on 19 June 2012", or even "this job has to be finished by midnight on 19 June 2012"? Each interpretation may be valid, depending on the context of the claim.

Chapter 9 presents an approach to deal rigorously with different time granularities in the context of temporal logics.

Example 3.7. Consider the following structurally similar sentences:

- Tomorrow, I will eat.
- Tomorrow, I will work.
- Tomorrow, I will go to the bank to pay my monthly bills.
- Tomorrow, I will stay in the city.

Depending on the time unit used to interpret the sentences, the meaning of "Tomorrow, I will…" changes from sentence to sentence. In particular, if we introduce the finer granularity of hours, the first two sentences read as "Tomorrow there will be *some* (few) hours when I will be eating" and "Tomorrow there will be *some* hours (say, eight) when I will be working"; the third sentence probably translates to "Tomorrow there will be *one* hour during which I will pay my bills"; the fourth one likely refers to the fact that "Tomorrow, during *all* hours of the day I will be somewhere in the city". The different meanings of the verbs ("eat", "work", "go", "stay") hint at different scopes ("some", "all", "one") in terms of hours during the day. ■

Exercise 3.8 (♣). Determine the most appropriate time units to interpret the following sentences:

- Tomorrow, I will work, and then I will go out.
- Tomorrow, I will have two classes, separated by a short break.
- Tomorrow, I will be on vacation. ■

Exercise 3.9 (♣). A hydroelectric power production system consists of a reservoir, an electric production station, and pipes connecting the dam of the reservoir to the power station; sluice gates to control the amount of water to be sent to the power

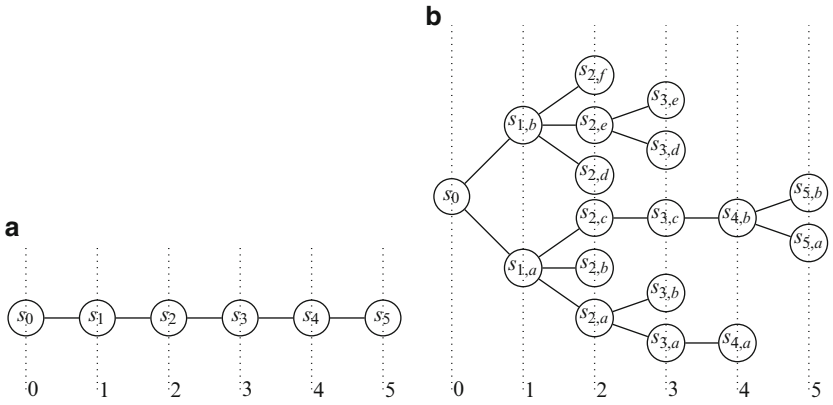

Fig. 3.3 A linear (**a**) and a branching (**b**) time model

station; and controlling devices which manage the sluice gates and the turbines that generate the power (e.g., to keep the frequency constant).

Consider a model of the global behavior of the system including the dynamics of the reservoir in terms of the amount of water coming in (from incoming rivers and the rain) and flowing out (from the pipes), the amount of power delivered by the plant, the control goals for the power supplied and water consumed, and the behavior of the digital controllers.

Which time unit would you use for each component of such a model? How would you combine them into a unique model? ■

3.3 Linear Versus Branching Time Models

The terms *linear* and *branching* refer to the structures on which formal languages are interpreted: *linear*-time formalisms are interpreted over *linear* sequences of states, whereas *branching*-time formalisms are interpreted over *trees* of states. In other words, a system description adopting a linear notion of time refers to linear behaviors, where the future evolution, from a given state at a given time, is always unique. Conversely, branching-time interpretations refer to behaviors structured in trees, where each "present state" may evolve into different "possible futures". Assuming discrete time, Fig. 3.3 pictures a linear sequence of states and a tree of states, over six time instants.

Linear behaviors are special cases of trees. Conversely, trees represent sets of linear behaviors that share common prefixes (i.e., that are prefix-closed). Under this duality, linear and branching models can be put on a common ground and compared; this has been extensively done in the literature.

In Fig. 3.3, the linear model (a) defines a total ordering of the symbols $s_0, s_1, \ldots, s_5$, whereas the branching model (b) induces a partial ordering of $s_0, s_{1,a}, \ldots, s_{5,b}$. Incidentally, the figure also suggests a metric on the time domain, so that symbols such as $s_{1,a}$ and $s_{1,b}$ are, in principle, not ordered but mark the same absolute time. If we ignore the metric information, we have a genuinely partial order where pairs such as $s_{1,a}$ and $s_{1,b}$ or $s_{1,a}$ and $s_{3,d}$ are unordered and it is undefined whether one occurs before or after the other.

The meaning of the branches in a branching-time model depends on the context and the system modeled. For example, the branch from s_0 to $s_{1,a}$ and $s_{1,b}$ in Fig. 3.3b may capture the fact that the system spawns two new parallel processes, whose behavior is described in each branch independently of the other. In a different interpretation, the same branch may describe a nondeterministic choice between two alternatives: each path in the tree is a totally ordered sequence of events or states that may happen in one of the possible computations, but elements in different branches have no order because they belong to mutually exclusive alternatives. Section 3.4 gives more details on such interpretations of branching time in the presence of nondeterministic computations.

Linear or branching semantic structures are then matched in the formal languages by corresponding syntactic elements that can express properties of specific features of the interpretation. This is possible, in principle, with all formal languages, but it is especially relevant for logic languages, and for temporal logics in particular. Linear-time temporal logics are interpreted over linear structures, and express properties of behaviors with unique futures, such as "if event p happens, then event q will happen eventually in the future". On the other hand, branching-time temporal logics are interpreted over tree structures, and state properties of branching futures, such as "if event p occurs, event q will occur along *some* of the originating branches". Chapter 9 discusses similar examples in greater depth with reference to temporal logics.

It is also possible to have semantic structures that are *branching in the past*, where different pasts merge into a unique present. Branching-in-the-past models are, however, uncommon in practical applications, so we will not deal with them.

Exercise 3.10 (♣). Consider a formalization of the game of chess in which a *state* is a given configuration of the pieces on the board, time instants coincide with moves of the players, and at every time instant there is a transition from the current state to the next one, determined by the move. Discuss whether linear or branching-time models adequately represent the following:

- A single match;
- The set of all matches starting with a given opening (e.g., all matches starting with the "Danish gambit" opening);
- A chess problem, such as: "starting from the given configuration, white to move and checkmate in three moves". ■

3.4 Deterministic, Nondeterministic, and Probabilistic Models

Linear time and branching time are features of languages and of the structures on which those are interpreted, whereas deterministic, nondeterministic, and probabilistic behaviors are attributes of the systems modeled or analyzed.

3.4.1 Deterministic Versus Nondeterministic Models

Consider systems including a notion of *input*, which evolve over time by reading the input, and changing the current state accordingly. A system is *deterministic* whenever the current value of state and input *uniquely* determine the future state. For instance, a light switch is a deterministic system, where pressing the button (input) when the light is in state off yields the unique possible future state of light on. Notice that, for a given input sequence, the initial state completely determines the behavior of a deterministic system.

Conversely, systems are *nondeterministic* if they can evolve to different future states from the same present state and input by making arbitrary "choices". For example, a resource arbiter is nondeterministic if it responds to two requests happening at the same time by "choosing" arbitrarily whom to grant the resource first, and the same pair of simultaneous requests may result in a different choice every time.

Example 3.11 (Ada's rendezvous). The rendezvous mechanism of the Ada programming language is a significant example of nondeterminism applied to the synchronization of parallel tasks. Consider two client tasks *Producer* and *Consumer* that depend on a server task *Buffer* to perform operations *Put* and *Get* respectively. In the Ada framework, when *Buffer* is ready to execute both *Put* and *Get* (i.e., it is neither full nor empty and there are pending requests – entry calls in Ada terminology), it chooses any of them nondeterministically. This behavior is embodied in the semantics of the **accept** statement. Figure 3.4 has a typical example of Ada code using this feature. Whenever the execution of *Buffer* reaches the **select** statement, all conditions ("guards") expressed by the **when** clauses are checked to determine which ones are enabled. If multiple **when** conditions hold and there are pending requests from other tasks for the corresponding entry, the system arbitrarily selects, in a nondeterministic fashion, one of the enclosed **accept** statements to be executed. The actual choice is resolved either by the compiler or by the operating system. Programmers must build programs which behave correctly independently of how the nondeterministic choices are actually resolved. ■

We will further discuss the semantics of Ada's rendezvous mechanism in Sect. 3.7.

A notation with nondeterministic features supports a high level of abstraction, where details that pertain solely to the implementation, such as the precise order of

Fig. 3.4 The rendezvous mechanism in Ada

```
task Buffer is
   entry Put (Item: in Integer);
   entry Get (Item: out Integer);
end;

task body Buffer is
begin loop
   select
      when Count < MAX => -- buffer not full
         accept Put (Item: in Integer) do
            ...
         end; -- accept
   or
      when Count > 0 =>    -- buffer not empty
         accept Get (Item: out Integer) do
            ...
         end; -- accept
   end select;
end loop;
end Buffer;
```

execution of tasks, are hidden as nondeterministic choices. At the same time, the lack of control over such lower-level details introduces subtleties which may make a full understanding of system behavior more difficult. For instance, in the loop of the Fig. 3.4 there is no control over the choice of pending request served at each iteration: the *Buffer* may always favor *Put* over *Get*, leaving the *Consumer* task waiting idly for a time that is related to the capacity of the buffer. In extreme cases – for instance, if the **accept** statements are not under the control of suitable **when** conditions or such conditions are always true – a process may even "starve" while waiting on an entry call because the server always selects other processes (we will discuss the notion of a *starving process* in Chap. 8).

More generally, nondeterminism is a powerful abstraction mechanism for incomplete knowledge in the description of systems. The sources of incompleteness and the semantics of the nondeterminism abstraction may vary with the application context.

In the example of Fig. 3.4, it is undetermined which **accept** statement should be executed when both are enabled. Similarly, system requirements may not commit to selecting from possible acceptable alternatives, with the objective of not over-constraining the implementation policies. For instance, a requirement of the type

> When signal S occurs, the system must react by signaling T no later than ten seconds and no sooner than five seconds.

is met by systems that always produce T after 8 s, by other systems that react after 6 s in certain conditions and after 9 s in others, as well as by many other implementations.

Another context in which nondeterminism can formalize incomplete knowledge occurs in search problems in unstructured spaces or, more generally, where there

is no a priori criterion for selecting from different options. Think, for example, of searching a generic graph or tree for nodes with certain properties.

Systems embedded in an external environment whose behavior is only partially known – power plants, distributed social groups, and so on – may also avail of nondeterministic abstractions to model imperfect knowledge of the environment.

The semantics of nondeterministic choice also depends on the application context and on the nature of knowledge incompleteness to be dealt with. For instance, in Example 3.11, the programmer may be oblivious to how the runtime system will choose from among enabled **accept** statements; hence he must guarantee that the program will meet its requirements *for all* possible operational conditions encompassed by its nondeterministic choice. Symmetrically, many generic search algorithms are modeled as nondeterministic visits of data structures: a visit is successful when *there exists some* sequence of nondeterministic choices that leads to the searched element.

We refer to the first type of nondeterminism – where every nondeterministic choice must produce an acceptable behavior – as *universal nondeterminism* (in analogy with universal logic quantification). Conversely, we will call the second type of nondeterminism – where it is sufficient that one nondeterministic choice leads to a valid solution – as *existential nondeterminism*. The following chapters will show various application contexts of nondeterministic abstractions and models, with references to the classification just introduced.

3.4.2 Nondeterministic Versus Probabilistic Models

A nondeterministic system can evolve into different future states from the same current state and input. The choice of which future state to take is arbitrary, and all possible alternatives are considered. In other words, nondeterminism is a convenient abstraction for sets of alternatives that can happen over different runs.

Probabilistic systems (also called "stochastic systems") can also choose from among different future states for a given current state and input. Unlike nondeterministic systems, however, the choice relies on probability distributions: the system selects the next state by drawing a value from a distribution and proceeding accordingly. An unbiased coin is an obvious example of a probabilistic system without input. Each state corresponds to the coin showing heads or tails. Flipping the coin moves the system to the next state, which is heads with probability $1/2$ and tails with probability $1/2$. The probability distribution associated with each transition induces a probability distribution on sets of behaviors. For example, the set of sequences of coin flips such that the first two draws are both head has a probability of $1/2 \cdot 1/2 = 1/4$.

Nondeterministic and probabilistic models are both concerned with representing incomplete information about the system's behavior: in the nondeterministic case there is no information at all about how choices are resolved, whereas in the probabilistic case the probability distribution gives a measure of the partial information

available. This quantitative difference results in a sharp conceptual difference in the types of questions addressed in the analysis of the two families of models and, consequently, in the type of mathematics needed to perform such an analysis. The analysis of nondeterministic models addresses "yes/no" questions about whether every possible behavior meets some requirements (such as "all pending requests are served within three seconds") or, symmetrically, whether there exists any behavior that achieves a certain goal. The analysis of probabilistic models addresses typically quantitative questions about the "likelihood" of certain events happening or not happening. For example, we can ask if "pending requests are served within three seconds 90 % of the time" – or, equivalently, with a probability of at least 90 %.

The conceptual difference between nondeterministic and probabilistic models persists even if we compare nondeterministic transitions to probabilistic ones with uniform probability. For example, the unbiased coin described above could also be modeled as a series of nondeterministic transitions that arbitrarily choose between heads and tails. The difference between the probabilistic and the nondeterministic model of the coin lies in the different weights given to possible sequences of flips. In the nondeterministic model, every sequence is on equal ground with all the others; an unbounded sequence of heads, for example, is perfectly legitimate behavior. In the probabilistic model, in contrast, different sequences have different probabilities and hence different likelihoods of happening. An unbounded sequence of heads, in particular, has zero probability of happening; hence it is essentially ruled out by the abstraction of the model.

Probabilistic modeling is a natural choice for systems whose dynamics are known only partially and empirically or are too complex to model exactly. Most physical phenomena happening in the natural world are of this type. For example, geological data may suggest the probability of an earthquake of a certain magnitude happening in a certain region during 1 year. Systems including human users are also often conveniently modeled with probabilities, for example, to quantify the chance that an operator performs a sequence of events in an incorrect, unsafe order, or to model the accesses to a Web server resulting from users visiting a certain HTML page.

The design of probabilistic algorithms is another, more sophisticated, application area within computing, where probabilistic models of timing properties are widely deployed. It turns out that several computationally complex algorithmic tasks can be sped up significantly by randomizing certain choices during the computation. Probabilistic algorithms give correct answers only with certain finite probabilities; in practice, however, this probability can often be made arbitrarily close to 1, so as to favorably leverage the trade-off between time spent computing and correctness of the result. Section 6.4 describes some examples of probabilistic computational models and algorithms.

The examples of nondeterministic and probabilistic behavior discussed so far focused on the choice from among different future *states*, but the same abstractions apply to the choices of waiting times, delays, and other time intervals. For example, in a communication protocol for handshaking (such as TCP's three-way handshake in Example 2.1), the event "acknowledge" always follows deterministically the event "start connection", but the time elapsing between an occurrence of "start

connect" and the corresponding occurrence of "acknowledge" is nondeterministic and varies according to the conditions of the communication network. In general, purely nondeterministic models consider arbitrary delays between a minimum $T_{\min}$ and a maximum $T_{\max}$ (which can be 0 and ∞ if every delay is possible), whereas probabilistic models associate a probability distribution T with the interval $[T_{\min}, T_{\max}]$ such that the delay is $t \in [T_{\min}, T_{\max}]$ with probability $T(t)$.

The term "stochastic" sometimes specifically refers to the application of probabilistic models to delays and timing information, as opposed to "probabilistic" models where the choice is of different future states. This terminology is, however, not universally accepted, and different research areas often use different conventions. This book uses the attributes "probabilistic" and "stochastic" as synonyms.

Finally, notice that, as with nondeterminism, probabilistic behavior may abstract incomplete knowledge of different origins: the input provided by the environment to the system may be known only statistically (the uncertainty may be in the input values or in its timing), and the system itself may react according to a deterministic or a probabilistic policy. When both system and environment have stochastic behavior, the overall model of systems embedded in the environment follows probabilities that depend on those of each component – this corresponds to the notion of *conditional probability*. Chapter 6 and the following ones describe various forms of probabilistic models, for different sources of uncertainty.

Exercise 3.12. Consider multiple consecutive iterations of the *Buffer* loop in Fig. 3.4 such that, at every iteration, both guards $Count > 0$ and $Count < MAX$ are true and there are pending requests for both *Get* and *Put*. Assume that *Buffer* always spends one time unit to perform a *Get* and two time units to perform a *Put*.

- Build a branching-time behavior consisting of a tree that summarizes all possible sequences of events over four iterations.
- Describe the set of linear-time behaviors representing the same sequences of events as the tree.
- Under the same conditions, what are the minimum and maximum times *Buffer* takes to execute ten consecutive loop iterations?
- Assume that there are always some pending requests for both *Get* and *Put* by some client; how much time, at most, must elapse before a call for a *Get* is certainly served? How much time for a call for a *Put*? ■

Exercise 3.13. Consider again multiple consecutive iterations of the *Buffer* loop in Fig. 3.4 such that, at every iteration, both guards $Count > 0$ and $Count < MAX$ are true and there are pending requests for both *Get* and *Put*. Assume that, at every iteration, the *Buffer* chooses to execute *Put* with probability $p = 40\,\%$ and *Get* with probability $q = 60\,\%$.

- What is the probability that *Put* has never been executed after ten loop iterations?
- How many iterations are needed to guarantee that both operations have been executed at least once with probability greater than 95 %?

- What values for the probabilities p and q minimize the number of iterations to guarantee that both operations have been executed at least once with probability greater than 90 %? ■

Exercise 3.14 (♦). Consider again multiple consecutive iterations of the *Buffer* loop in Fig. 3.4 such that, at every iteration, both guards $Count > 0$ and $Count < MAX$ are true and there are pending requests for both *Get* and *Put*. Assume that, at every iteration, the *Buffer* chooses to execute *Put* and *Get* deterministically in strict alternation. Every execution of *Put* and *Get* takes time whose probability is uniformly distributed in the interval [10, 20] milliseconds.

- What is the average duration of ten loop iterations?
- What is the probability that executing ten loop iterations takes more than 120 ms?

(*Hint*: the exercise is simpler if the interval [10, 20] is taken to be discrete rather than continuous). ■

3.4.3 Deterministic, Probabilistic, and Nondeterministic Versus Linear- and Branching-Time Models

There is a natural coupling between, on one side, deterministic systems and linear models, and, on the other side, nondeterministic or probabilistic systems and branching models. In linear-time models the future of any instant is unique, and hence the modeled system is deterministic, whereas in branching-time models each instant branches into different futures, corresponding to possible nondeterministic choices.

This natural correspondence notwithstanding, determinism and linearity of time are distinct concepts, which target different concerns. For instance, linear-time models are often preferred – even for nondeterministic systems – for their intuitiveness and simplicity. The discussion of Petri nets in Chap. 8 will provide examples of linear time domains expressing the semantics of nondeterministic formalisms. On the other hand, branching-time models can describe sets of computations of deterministic systems for different input values. For instance, the branches of a tree can describe all possible computations of an array sorting algorithm, where each branch corresponds to a choice made by the algorithm on the basis of comparisons between array elements. Analyzing the tree gives measures of the minimum, maximum, and average execution times.

3.5 Implicit Versus Explicit Time Reference

Some languages for the description of temporal properties make explicit reference to temporal items (attributes or entities of "type time", such as the occurrence times of events and the durations of states or actions), whereas other formalisms leave such references implicit in their syntax.

To illustrate, consider the case of pure first-order predicate calculus to specify system behavior and its properties, as done in some examples of Chap. 2. Formulae explicitly refer to terms ranging over the time domain and combine them with quantifiers; such formulae give properties where explicit time references are frequent, such as in the sentence

> For every instant of time t, the safe is open if and only if there exists another time instant u, smaller than t and at least as large as $t - 3$, such that the correct code has been entered at u.

which corresponds to a part of formula (2.6) in Chap. 2. On the contrary, formulae of classic temporal logic, despite its name, do not mention any temporal quantities explicitly, and express temporal properties in terms of an implicit "current time" and the ordering of events with respect to it; for example, a simple sentence in this style reads

> If the correct code is entered [implicitly assuming the adverb *now*], then the safe will open sometime in the future, and then it will close again.

Most formalisms adopt some kind of intermediate approach between the extremes of purely explicit and purely implicit references. For instance, many types of abstract machines can specify explicitly the duration of activities with implicit reference to their starting time (Statecharts, discussed in Chap. 7, and Petri Nets, presented in Chap. 8, are two representative examples). Other languages inspired by temporal logic (such as MTL, presented in Chap. 9) keep its basic approach of referring any formula to an implicit current instant (the *now* time) but can explicitly express time distances with respect to it. Such logics can express properties such as

> If the correct code is entered [*now*], then the safe will open immediately, and then it will close again after exactly three time units.

Using implicit references to time instants – in particular an implicit *now* – is quite natural and convenient when modeling so-called "time-invariant systems", which are the majority of real-life systems: in most cases, in fact, the system behavior does not depend on the absolute value of time but only on the relative time distances. Therefore, expressing explicitly where the *now* is located along the time axis is irrelevant for such system models.

Example 3.15 (Explicit and implicit time). Sentences stating historical facts typically use explicit time references[2]:

- During the year 1625, a dramatic famine struck Europe; the famine lasted until the beginning of the year 1630.
- The starving population was an easy target for an epidemic of plague, which began in 1629 and lasted until 1631.
- During the years 1625–1631, life expectancy dropped from 50 to 37 years.

[2]The following three sentences refer to some real historical facts mentioned in Alessandro Manzoni's *The Betrothed*; the dates and figures are plausible but not necessarily accurate.

Observations about morals usually use implicit time references to convey timelessness:

- Every lie is eventually uncovered.
- You can fool some of the people all of the time, and all of the people some of the time, but you cannot fool all of the people all of the time.[3]

Engineering artifacts are often time-invariant systems, naturally described with an implicit "now":

- The speed of a braking car decreases proportionally relative to the time since when braking starts (see Example 2.3).
- The discharge time of a capacitor attached to a resistor depends only on the resistor's resistance, the capacitor's capacity, and the initial charge accumulated, irrespective of the absolute time when discharging starts (the example is developed further in Chap. 4). ■

Exercise 3.16 (♣). Analyze the following sentences in natural language, and determine the kind of implicit or explicit time references they contain.

- World War II lasted 6 years from 1939.
- The last death of a US president in office occurred in 1963.
- The final agreement must be signed within 30 days from the subscription of the letter of intent.
- After he reached the age of 60, he was never in good health for more than 3 months.
- A vast majority of the "baby boomers" will not be able to retire before the age of 65.
- Life expectancy has steadily increased in the last three centuries, and it is now over 80 years in a few countries.
- You tried your best, and you failed miserably. The lesson is, never try. ■

3.6 The Time Advancement Problem

The problem of time advancement arises when the model of a timed system exhibits behaviors that do not progress past some instant. Usually, such standstill behaviors do not correspond to any physical "real" phenomena; they may be the consequence of some incompleteness and inconsistency in the formalization of the system, and must thus be ruled out.

The simplest manifestation of the time advancement problem arises when transitions that occur in null time are possible. For instance, several automata-based formalisms such as Statecharts and timed versions of Petri nets support such

[3] This quotation is usually attributed to Abraham Lincoln, but this is allegedly apocryphal.

abstract zero-time transitions (see Chaps. 7 and 8). Although truly instantaneous actions are physically unfeasible, they nonetheless are useful abstractions for events that take an amount of time which is negligible with respect to the overall dynamics of the system; pushing a button is an example of an action whose actual duration can usually be ignored and that can thus be represented abstractly as a zero-time event. When zero-time transitions are allowed, an infinite number of such transitions may accumulate in an arbitrarily small interval, thus modeling a fictitious infinite computation where time does not advance past the interval. Behaviors where time does not advance are usually called "Zeno" behaviors, from the ancient philosopher Zeno of Elea[4] and his paradoxes on time advancement. From a rigorous point of view, even the notion of behavior as a function – whose domain is time and whose range is the system state (see Chap. 4) – is ill-defined with zero-time transitions: if the transition is instantaneous, the system is both at the source state and at the target state in the same instant.

Even if actions are non-instantaneous, Zeno behaviors can still occur if time advances only by *arbitrarily small* amounts. Consider, for instance, a system that produces an unbounded sequence of events p_k, for $k \in \mathbb{N}$; each event p_k happens exactly t_k time units after the previous one (i.e., p_{k-1}). If the series of the relative times t_k (that is, the infinite sum $\sum_k t_k$ of the time distances between consecutive events) converges to a finite limit t, then the absolute time never surpasses t; in other words, *time stops* at t, while an infinite number of events occur in the finite time between any t_k and t.

Zeno behaviors exist also for continuous-valued time-dependent functions of time that vary smoothly. Take, for instance, the real-valued function of time

$$b(t) \quad = \quad \begin{cases} \exp\left(-\frac{1}{(t-t_0)^2}\right) \sin\left(\frac{1}{t-t_0}\right) & t \neq t_0\,, \\ 0 & t = t_0\,. \end{cases} \tag{3.3}$$

$b(t)$ is very smooth, as it possesses continuous derivatives of all orders. Nonetheless, its sign changes an infinite number of times in any interval containing the time instant t_0; therefore, if we consider the event of function $b(t)$ changing its sign, an unbounded sequence of such events takes place before t_0, without time advancing past t_0; natural notions such as "the last or next instant at which the sign of b changes" are not defined at time t_0, and, consequently, we cannot describe the system by relating its behavior to such – otherwise well-defined – notions. Indeed, as will be explained precisely in Chap. 9 when discussing temporal logics, absence of Zenoness may be obtained through the mathematical notion of *analyticity*, which is even stronger than infinite derivability.

Even when Zeno behaviors are ruled out, and hence time progresses, the occurrence of an unbounded number of events in intervals of fixed finite length may lead to "irregular" behaviors that complicate the analysis. For example, the distance

[4] Circa 490–425 B.C.

between consecutive events may get indefinitely smaller while time diverges, such as in the harmonic sequence defined by $t_{k+1} = t_k + 1/k$. These behaviors are called "Berkeley", after the philosopher George Berkeley[5] and his investigations arguing against the notion of infinitesimal. Systems with Berkeley behaviors cannot be controlled by digital controllers operating with a fixed sampling rate since the behaviors cannot be suitably discretized. On the other hand, several real-life systems cannot guarantee an a priori bound on the "speed" of events; hence their model must include Berkeley behaviors.

Some well-known problems of – possibly – concurrent computation such as *termination*, *deadlocks*, and *fairness* can be considered as *dual* problems to time advancement, because they describe processes that fail to advance their *states*, while time keeps on flowing. Examples of these problems and their solutions are discussed with reference to a variety of formalisms in Part II of the book.

Two different approaches manage the time advancement problem: we refer to them as "a priori" and "a posteriori" methods. In a priori methods, the syntax or the semantics of the formal notation is restricted beforehand, in order to guarantee that every system model is exempt from time advancement problems by construction. For instance, in some notations every transition must necessarily take a positive time greater than some fixed value c. A less restrictive assumption, which guarantees a good level of abstraction while still avoiding a priori the risk of Zeno and even Berkeley behaviors, allows for only finite sequences of zero-time transitions that are followed by an event that takes a minimum fixed time. This view does not explicitly restrict the number of events occurring in any finite time, but ensures that no infinite sequence ever accumulates. It is well suited, for instance, for expressing a sequence of logic gate switches in a hardware processor that occur within a single clock interval.

A posteriori methods, in contrast, deal with time advancement issues only *after* the system specification has been built; the specification is analyzed against a formal definition of time advancement, in order to check that all of its actual behaviors do not run into the time advancement problem. A posteriori methods may be particularly useful for detecting possible criticalities in the behavior of real systems already built. For instance, the oscillations exhibited by a mathematical model with a frequency that goes to infinity within a finite time interval, as in the function $b(t)$ mentioned in (3.3) above, may be the symptom of some instability in the modeled physical system, in the same way a physical quantity – say, a temperature or a pressure – that tends to infinity within a finite time in the model is the symptom of a serious possible failure in the real system.

The same "duality" – a priori avoidance vs. a posteriori verification – is often assumed to deal with the symmetric problem of *process* advancement.

[5] Kilkenny, 1685–Oxford, 1753.

Exercise 3.17. Consider a system whose state s evolves according to the function of time $s(t) = \sin(\omega t^2)$. How would you classify such a behavior? A Zeno behavior? A Berkeley behavior? None of them? ■

Exercise 3.18. An unbounded sequence of events occur each at time $t_1, t_2, \ldots, t_i$, $t_{i+1}, \ldots$, where

$$t_k = \begin{cases} 0 & k = 1\,, \\ t_{k-1} + d_{k-1} & k > 1\,. \end{cases}$$

Define, if possible, a sequence of values $d_1, d_2, d_3, \ldots$ such that the resulting sequence of events is:

1. Zeno and all events but the first occur at irrational times;
2. Zeno and all events occur at integer times;
3. Non-Zeno and Berkeley;
4. Zeno and Non-Berkeley;
5. Non-Berkeley and all events but the first occur at irrational times;
6. Non-Berkeley and all events occur at integer times. ■

3.7 Concurrency and Composition

Most real *systems* – as the term itself suggests – are complex enough that it is useful, if not outright unavoidable, to model, analyze, and synthesize them as the composition of several subsystems. Such a composition/decomposition process may be iterated until each component is simple enough to be analyzed directly.

Composition and decomposition, also referred to as *modularization*, are general and powerful design principles in any field of engineering. In particular, in the case of – mostly sequential – software design, they have originated a rich collection of techniques and language constructs, from subroutines to abstract data types and object orientation.

The application of the same principles of modularity to concurrent and timed systems is definitely less mature, and in fact only a few programming languages deal explicitly with concurrency. From a programming language viewpoint, the central issue with the modeling of concurrency is the *synchronization* of activities (embodied in different constructs such as processes, tasks, and threads) when they access shared resources or exchange messages. The etymology of the word "synchronization" is quite descriptive of the timing issues that are at stake: "synchronization" combines the Greek words $\sigma\upsilon\nu$ (which means "together") and $\chi\rho\sigma\nu\sigma$ (which means "time"), and in fact, concurrent activities evolve in parallel independently, until they must synchronize and meet at the same time. Synchronization may require that faster activities slow down and wait for the slower activities to meet themselves at the "same time".

When the concurrent activities of the modules are heterogeneous in nature, formally modeling the synchronization of components becomes even more intricate

because of the difficulty of coming up with a uniform time model. For instance, a plant, a vehicle, and a group of people can each be one module, interacting with other modules for monitoring and control implemented in hardware and software. Consequently, time references can be implicit for some activities and explicit for others; also, the overall system model might include parts in which time is represented simply as an ordering of events and parts that are described through a metric notion of time; finally, the system may even be hybrid, with different components referring to time domains of different natures (discrete or continuous).

It is often convenient to distinguish, within concurrent components, between the *environment* and the system *embedded* into it, which typically monitors, controls, or manages the environment. We will see that the models of, and the roles attached to, system and environment significantly vary with notations and application domains. In some cases, the environment models an independent external entity that only supplies stimuli to the system, which inputs them; the system's inputs from the environment may be modeled as nondeterministic sequences of events. In other cases, the environment is just one of the components of an overall global system, and it forms a feedback loop with the other modules both by providing them with input and by reacting to their output. Models of the first type, where the environment is an independent external module, are called *open systems* (that is, open to the external environment), whereas models of the second type are called *closed systems*.

The following subsections provide a basic classification of the approaches dealing with the concurrent composition of timed units.

3.7.1 Synchronous Versus Asynchronous Composition

Synchronous and asynchronous compositions are two paradigms for combining the temporal evolution of concurrent modules.

Synchronous composition constrains state changes of the various units to occur at the very same time, or at time instants that are strictly and rigidly related. Time models with synchronous composition naturally refer to discrete time domains, although exceptions are possible where the overall system synchronizes over a continuous time domain.

Conversely, in *asynchronous* composition, each unit can progress independently of the others. In this view, there is no need to know in which state each unit is at every instant; in some cases this is even impossible: for instance, if we are dealing with a system that is geographically distributed over a wide area and the state of a given component changes in a time period that is shorter than that needed to send information about the state to other components. A similar situation occurs in totally different realms, such as the global stock market, where the differences in local times at locations all over the world make it impossible to define certain states about the market, such as when it is "closed".

While units progress independently most of the time, the "real" synchronization of asynchronously composed systems occurs with dedicated events at special

"meeting points", and according to specific rules. The *rendezvous* mechanism of the Ada programming language, which we mentioned in Sect. 3.4.1 for its nondeterministic features, is a typical example of synchronization between asynchronous tasks: a task owning a resource (the *Buffer* in Example 3.11) waits to grant it until it receives a request thereof; symmetrically, a task that needs to access the resources raises a request (an *entry call*) and waits until the owner is ready to accept it. When both conditions are verified (an entry call is issued and the owner is ready to accept it) the rendezvous occurs, and the two tasks are synchronized. At the end of the entry execution by the owner, the tasks split again and continue their asynchronous execution. As we saw in Sect. 3.4, Ada combines this mechanism with a nondeterministic choice in case two or more different rendezvous are possible between the task owning the resources and those asking for their use at a given time.

Many formalisms feature some kind of asynchronous composition. Among these, Petri nets (described in depth in Chap. 8) exhibit similarities with the Ada task system.

Unsurprisingly, asynchronous composition is usually more complex to formalize precisely than synchronous composition. Chapters 7, 8, and 10 present several representative approaches to this problem.

Exercise 3.19. Consider the standard concurrency libraries of the following general-purpose programming languages:

- C (processes);
- Java (threads);
- C# (threads);
- Eiffel (processors and SCOOP).

Are the models of parallelism they implement synchronous or asynchronous? What kinds of synchronization mechanisms do they offer? Do you know any programming language or modeling notation that features a purely synchronous concurrency model? ■

3.7.2 Message Passing Versus Resource Sharing

Another major classification of the mechanisms to compose and coordinate concurrent system components is into message passing and resource sharing. The two terms are rather self-explanatory: in message-passing coordination, components exchange messages over communication channels (such as buffers or pipes) to synchronize; in resource-sharing coordination, different components have access to the same resources (such as memory locations), and communication takes place when a unit reads from the shared space what another unit has written. "Google Docs" is an example of Internet-based application with shared concurrent access by multiple users, whereas email is a typical message-passing mechanism.

At lower levels of abstraction, communication will ultimately involve concurrent access to some shared resource. For example, in a high-level programming language with routine parameters passed by reference, the passage of actual parameters is akin to a message-passing communication mechanism even when it is implemented by means of sharing of global variables, accessed in a disciplined way. Even email communication involves several implementation steps where buffers, communication channels, and other resources are shared in the various stages of the transmission.

At higher abstraction levels of applications, however, message passing and resource sharing feature different and peculiar properties, and involve some clear trade-offs. Both types of coordination support asynchronous interaction among activities (processes, threads, and so on), though, in principle, they could both be deployed in fully synchronous systems. Message passing, on the one hand, decouples the timing of concurrent activities almost completely; an email message, for instance, might never be received. Resource sharing, on the other hand, usually requires stricter coordination rules, in particular to manage read and write access rights; this makes it more likely that some activities have to wait explicitly before accessing the shared resources.

Many modeling formalisms with some notion of module composition feature coordination modeling primitives that correspond to message-passing mechanisms (for example, channels and send/receive message primitives), to memory-sharing ones (for example, global variables), or to both. Chapters 4 and 10 describe some relevant examples.

In terms of other dimensions of time modeling, both message-passing and resource-sharing synchronization influence the *ordering* of events. For instance, reading an email message is possible only after it has been sent and subsequently delivered; the order is partial because messages need not be read in the same order in which they were sent. On the other hand, synchronization strategies may also depend on *metric* aspects of time in parallel systems with real-time requirements.

Exercise 3.20. Consider the following functional programming languages:

- Haskell;
- Erlang;
- Scala.

What kind of coordination mechanism do they offer: message passing or memory sharing? ■

3.8 Analysis and Verification Issues

Formal models must be amenable to analysis to be useful, so that probing the models can determine whether the systems will behave as expected and will possess the desired features. The characterizing properties that the model (and then the system) must exhibit are often called *requirements*; hence the task of checking that a given

model satisfies a set of requirements is called *verification*. Although this book is not about verification, the discussion and comparison of formalisms must refer to several notions related to verification and, more generally, formal analysis. The rest of the current section presents these notions, which have broad scope, focusing the discussion on their relevance to timed models.

3.8.1 Expressiveness

The notion of *expressiveness* refers to the possibility of characterizing extensive classes of properties; it is a fundamental criterion for the classification of formal languages. A language L_1 is more expressive than another language L_2 if the sentences of L_1 can define sets of behaviors that no sentence of L_2 can identify precisely, whereas everything definable with L_2 is definable with L_1 as well. This informal definition implies that the expressiveness relation among languages is a partial order, as there are pairs of formal languages whose expressive power is incomparable: for each language, there exist properties that can be expressed only with the other language. In other cases, different formalisms have the same expressive power; hence they can express the very same properties with different syntax. Expressiveness only deals with the logical possibility of expressing properties; hence it differs from other – somewhat subjective, but nonetheless very relevant – characterizations such as conciseness, readability, naturalness, and ease of use.

Several of the other dimensions of time modeling often mutually influence the expressiveness of the formalisms. For example, a language that can only constrain the ordering of events is less expressive, by definition, than a similar language that includes primitives to declare the temporal distance between consecutive events. In other cases, the possibility of expressing metric constraints depends on other dimensions; for example, classic temporal logic, presented in Chap. 9, can express time distances – even if somehow cumbersomely – only over discrete time domains.

3.8.2 Decidability and Complexity

Although in principle one might prefer the "most expressive" formalism, in order not to be restrained in what can be expressed, there is a fundamental trade-off between expressiveness and another important characteristic of a formal notation: *decidability*. A certain property is *decidable* for a formal language if there exists an algorithmic procedure that is capable of determining, for any model formalized in that language, whether the property holds or not in the model. Therefore, the verification of decidable properties is – at least in principle – a totally automated process. The trade-off between expressiveness and decidability arises because properties may become undecidable with more expressive languages. The verification of undecidable properties can only resort to semi-automated or manual methods, or to

partial techniques such as testing and simulation. "Partial" means that the results of the analysis may be incomplete or incorrect for a subset of all possible behaviors of the models.

Let us consider the property of *termination* to illustrate the trade-off between expressiveness and decidability. The verification problem reads as follows: given any program expressed in a programming language L, determine if it will terminate for every possible input. Termination is a temporal property, where time can feature implicitly ("The program will *eventually* halt") or explicitly ("There exists a future time t such that the program will halt at t"). Whether termination is decidable depends on the expressiveness of the programming language L.

As discussed in Chap. 6, general-purpose programming languages, such as C and Lisp, achieve maximum expressiveness, and consequently termination is undecidable for programs in such languages. If, however, the expressive power is sufficiently restricted, termination becomes decidable. For example, the termination of programs written in a subset of C where dynamic memory allocation, recursion, and the preprocessor are disabled is decidable, because such programs use an a priori bounded amount of memory. This subset is, however, less expressive than the full language, and many C programs cannot be encoded under these restrictions.

While decidability is just a "yes/no" property, *complexity* analysis provides, in the case where a given property is decidable, a measure of the computational effort required by an algorithm to decide whether the property holds or not for a model. The computational effort is typically measured in terms of the amount of memory or time required to perform the computation, as a function of the length of the input (that is, the size of its encoding). Chapter 6 presents more details about this classical view of computational complexity.

3.8.3 Analysis and Verification Techniques

There exist two broad families of verification techniques: those based on *exhaustive enumeration* procedures, and those based on *syntactic transformations* like deduction or rewriting, typically in the context of some axiomatic description. Although large, these two classes do not cover the whole spectrum of verification algorithms, which comprises very different techniques and methods; here, however, we limit ourselves to sketching a minimal definition of these two basic techniques.

Exhaustive enumeration techniques are mostly automated, and are based on the exploration of graphs or other structures representing an operational model of the system, or of the space of all possible interpretations for formulae expressing the required properties. A typical example of this kind of technique is model checking, illustrated in Chap. 11.

Techniques based on *syntactic transformations* typically address the verification problem by means of logic deduction. These techniques can be applied when the model, its requirements, or both are in descriptive form; then the verification may consist of successive applications of deduction schemata, until the requirements

are shown to be a logical consequence of the system model. Mathematical logic is a classic example of formalism focused on syntactic transformations, as shown in Chap. 2. Chapter 2 also exemplified how applying deduction schemata incurs in a trade-off between expressiveness and decidability: simple propositional logic supports deduction schemata where every expressible property is decidable, but its expressive power is limited to simple behaviors. The much more expressive predicate logic is therefore preferable for complex system specification, but no verification technique can decide the validity of every sentence of predicate logic. Chapter 9 will discuss similar trade-offs for logics supporting a notion of time.

3.8.3.1 Summing Up

This chapter presented some dimensions that characterize languages and methods for the modeling and analysis of timed systems. The dimensions will support the presentation of the languages in the rest of the book, and will help readers classify, compare, and evaluate other similar notations, and possibly even derive new ones if needed. The dimensions of this chapter are not orthogonal, and indeed we discussed many examples of mutual influence and dependence among them. The dimensions are also often qualitative, in that a rigid classification of every language against every dimension would often be vacuous, unsubstantiated, or even misleading for widely different notations with heterogeneous scopes. The rest of the book explicitly discusses the most relevant dimensions for each notation; when doing so in Part II of the book, the keywords referring to the dimensions discussed are graphically **EMPHASIZED** and referenced in the index with sub-entries corresponding to the formalisms under discussion. Dealing with the dimensions not mentioned explicitly is a useful exercise that will improve your understanding of the book's content.

3.9 Bibliographic Remarks

Koymans discusses the nature of time domains for real-time modeling [18]. Some textbooks consider hybrid systems in a general setting [22, 29]. A few authors address the issue of different time granularities [6, 8, 28]. The classical theory of sampling considers the equivalence between continuous-time signals and their discrete-time samplings [4, 9]. Wirth [30] first pointed out the difference between purely parallel and real-time systems.

The concept of invariance under stuttering was introduced by Lamport [21] and characterized for temporal logics [19, 24]. The differences between linear- and branching-time models were discussed extensively in classic references on temporal logics [10, 11, 18, 20] and real-time logics [1]; Koymans [18], in particular, mentioned branching-in-the-past time models. Classic temporal logic with the notion of an implicit current time was introduced by philosophers [17]. Pnueli pioneered its usage as a modeling tool in computing [26]. Abadi and Lamport

introduced the attribute "Zeno" to describe behaviors where time stops; by analogy, we suggested the attribute "Berkeley" to characterize a different category of time advancement problems [13, 14]. Notions of nonstandard real analysis allow for a simpler approach to model and reason about Zenoness in the presence of zero-time transitions [15].

Fairness and other concurrency problems are described in various texts on parallel programming [2, 3, 12]. Most software engineering books elucidate the notions of requirements and verification [16, 25, 27].

The main features of the Ada programming language are described in many texts, such as Booch and Bryan [5]. A first critical semantic analysis of the rendezvous mechanism and a possible formalization by means of Petri nets was published by Mandrioli et al. [7, 23].

References

1. Alur, R., Henzinger, T.A.: Logics and models of real time: a survey. In: Real Time: Theory in Practice. Lecture Notes in Computer Science, vol. 600, pp. 74–106. Springer, Berlin/New York (1992)
2. Andrews, G.R.: Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley, Reading (2000)
3. Ben-Ari, M.: Principles of Concurrent and Distributed Programming, 2nd edn. Addison-Wesley, Harlow/New York (2006)
4. Benedetto, J.J., Ferreira, P.J. (eds.): Modern Sampling Theory. Birkhäuser, Boston (2001)
5. Booch, G., Bryan, D.: Software Engineering with ADA. Addison-Wesley, Boston (1994)
6. Burns, A., Hayes, I.J.: A timeband framework for modelling real-time systems. Real-Time Syst. **45**(1–2), 106–142 (2010)
7. Cocco, N., Mandrioli, D., Milanese, V.: The Ada task system and real-time applications: an implementation schema. J. Comput. Lang. **10**(3/4), 189–209 (1985)
8. Corsetti, E., Crivelli, E., Mandrioli, D., Morzenti, A., Montanari, A., San Pietro, P., Ratto, E.: Dealing with different time scales in formal specifications. In: Proceedings of the 6th International Workshop on Software Specification and Design, pp. 92–101. IEEE, Los Alamitos (1991)
9. Deming, W.E.: Some Theory of Sampling. Dover, New York (2010)
10. Emerson, E.A.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 996–1072. Elsevier, Amsterdam/New York (1990)
11. Emerson, E.A., Halpern, J.Y.: "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. J. ACM **33**(1), 151–178 (1986)
12. Francez, N.: Fairness. Monographs in Computer Science. Springer, New York (1986)
13. Furia, C.A., Rossi, M.: A theory of sampling for continuous-time metric temporal logic. ACM Trans. Comput. Log. **12**(1), 1–40 (2010). Article 8
14. Furia, C.A., Pradella, M., Rossi, M.: Automated verification of dense-time MTL specifications via discrete-time approximation. In: Cuéllar, J., Maibaum, T., Sere, K. (eds.) Proceedings of the 15th International Symposium on Formal Methods (FM'08). Lecture Notes in Computer Science, vol. 5014, pp. 132–147. Springer, Berlin/New York (2008)
15. Gargantini, A., Mandrioli, D., Morzenti, A.: Dealing with zero-time transitions in axiom systems. Inf. Comput. **150**(2), 119–131 (1999)
16. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering, 2nd edn. Prentice Hall, Harlow (2002)

17. Kamp, J.A.W.: Tense logic and the theory of linear order. Ph.D. thesis, University of California at Los Angeles (1968)
18. Koymans, R.: (Real) time: a philosophical perspective. In: de Bakker, J.W., Huizing, C., de Roever, W.P., Rozenberg, G. (eds.) Proceedings of the REX Workshop: "Real-Time: Theory in Practice". Lecture Notes in Computer Science. vol. 600, pp. 353–370. Springer, Berlin/New York (1992)
19. Kučera, A., Strejček, J.: The stuttering principle revisited. Acta Inform. **41**(7/8), 415–434 (2005)
20. Lamport, L.: "Sometime" is sometimes "not never": on the temporal logic of programs. In: Proceedings of the 7th ACM Symposium on Principles of Programming Languages (SIGPLAN-SIGACT), pp. 174–185. ACM, New York (1980)
21. Lamport, L.: What good is temporal logic? In: Mason, R.E.A. (ed.) Proceedings of the 9th IFIP World Congress. Information Processing, vol. 83, pp. 657–668. North-Holland, Amsterdam/New York/Oxford (1983)
22. Lygeros, J., Tomlin, C., Sastry, S.: Hybrid Systems: modeling, Analysis and Control. Available online https://inst.eecs.berkeley.edu/~ee291e/sp09/handouts/book.pdf (2008)
23. Mandrioli, D., Zicari, R., Ghezzi, C., Tisato, F.: Modeling the Ada task system by Petri nets. J. Comput. Lang. **10**(1), 43–61 (1985)
24. Peled, D., Wilke, T.: Stutter-invariant temporal properties are expressible without the next-time operator. Inf. Process. Lett. **63**(5), 243–246 (1997)
25. Pfleeger, S.L., Atlee, J.: Software Engineering: Theory and Practice, 3rd edn. Prentice Hall, Upper Saddle River (2005)
26. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS'77), pp. 46–67. IEEE, New York/Long Beach (1977)
27. Pressman, R.: Software Engineering: A Practitioner's Approach, 7th edn. McGraw-Hill, Dubuque (2009)
28. Roman, G.C.: Formal specification of geographic data processing requirements. IEEE Trans. Knowl. Data Eng. **2**(4), 370–380 (1990)
29. van der Schaft, A., Schumacher, H.: An Introduction to Hybrid Dynamical Systems. Lecture Notes in Control and Information Sciences, vol. 251. Springer, London/New York (2000)
30. Wirth, N.: Toward a discipline of real-time programming. Commun. ACM **20**(8), 577–583 (1977)

http://www.springer.com/978-3-642-32331-7

Modeling Time in Computing

Furia, C.A.; Mandrioli, D.; Morzenti, A.; Rossi, M.

2012, XVI, 424 p., Hardcover
ISBN: 978-3-642-32331-7