1. Introduction

1.1 Real-Time Systems

A real-time system is a computing system with real-time requirements. Let us consider the following two examples of real-time systems.

1.1.1 Two Examples

Deadline-Driven Scheduler

Consider a finite number of processes, say $p_1, p_2, \ldots, p_m$, which share a single processor. Each process p_i has a periodic behavior. In a period of length T_i, process p_i requests a constant amount of processor time C_i, where $C_i < T_i$.

We assume that the request periods for process p_i start at times $k \cdot T_i$, for $k = 0, 1, 2, 3, \ldots$.

The purpose of the scheduler is to grant processor time to the processes, i.e. to schedule the processes, so that process p_i runs on the processor for C_i time units in every period, for $i = 0, 1, \ldots, m$.

Figure 1.1 shows a schedule for first two periods of process p_i. In the first period, from time 0 to time T_i, three pieces of processor time, with durations C_{i_1}, C_{i_2} and C_{i_3} are scheduled for p_i. The requirement of p_i is fulfilled in the first period, since $C_i = C_{i_1} + C_{i_2} + C_{i_3}$. In the second period, from time T_i to time $2 \cdot T_i$, two pieces of processor time are scheduled for p_i. However, the requirement of p_i is not satisfied in the second period, as $C_i > C'_{i_1} + C'_{i_2}$.

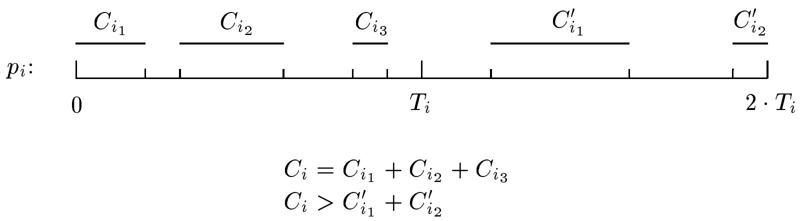

Fig. 1.1. Schedules for p_i in the first two periods

The requirement for the scheduler is to fulfill all requests of the processes. This is a real-time requirement, as any request of a process must be fulfilled before its expiration.

The deadline-driven scheduling algorithm was proposed in [85]. It satisfies this requirement, under the assumptions that the scheduler overhead is negligible and

$$\sum_{i=1}^{m} \frac{C_i}{T_i} \leq 1 .$$

In this algorithm, the expiration time of a request is called the *deadline* of the request. The algorithm *dynamically* assigns priority to each process according to the urgency, i.e. the deadline, of its current request. A process will be assigned the highest priority if it is the most urgent, i.e. the deadline of its current request is the nearest, and will be assigned the lowest priority if it is the least urgent, i.e. its deadline is the furthest. At any instant, only one of the processes, with the highest priority and an unfulfilled request, can be selected to *occupy* or even *preempt* the processor.

The correctness of the algorithm is not obvious. Reference [85] has provided an informal proof of it. □

Gas Burner

This example was first investigated in [145]. A gas burner is either heating when the flame is burning or idling when the flame is not burning, and it alternates indefinitely between heating and idling. Usually, no gas is flowing while it is idling. However, when it changes from idling to heating, gas must be flowing for a little time before it can be ignited, and when a flame failure occurs, gas will be flowing before the failure is detected and the gas valve is closed. Hence, there may be a time interval in which gas is flowing and the flame is not burning, i.e. where gas is *leaking*. A design of a safe gas burner must ensure that the time intervals where gas is leaking do not become too long.

Let us assume that the ventilation required for normal combustion would prevent a dangerous accumulation of gas provided that the proportion of leak time is not more than one-twentieth of the elapsed time for any time interval at least one minute long – otherwise the requirement would be violated immediately on the start of a leak. This is also a real-time requirement.

Turning next to the task of design, certain decisions must be taken about how the real-time requirement is to be met. For example, it could be decided that for any period where the requirement is guaranteed, any leak in this period should be detectable and stoppable within one second; and to prevent frequent leaks, it is acceptable that after any leak in this period, the gas burner rejects the switching on of gas for thirty seconds. The conjunction of these two decisions implies the original requirement, a fact which must be proved before implementation proceeds.

After justification of the design decisions, a computer program can be designed accordingly, and hosted in the gas burner. This program interacts with a flame sensor to detect flame failures, and controls the opening and closing of the gas valve, so that the design decisions, and hence the requirement, can be satisfied. □

Both the deadline-driven scheduler and the gas burner are real-time systems, although the first one is a software system, and the second is a software-embedded system, also called a hybrid system.

Duration calculus (abbreviated to DC) is a logical approach to designing real-time systems. Real numbers are used to model time, and functions from time to Boolean values are used to model the behavior of real-time systems. On the basis of interval logic, DC provides a formal notation to specify properties of real-time systems and a calculus to formally prove those properties, such as the satisfaction of the requirements for the deadline-driven scheduling algorithm and for the design decisions of the gas burner.

1.1.2 Real Time

At the level of requirements, real time is often understood popularly as continuous time. However, at the level of implementation, a piece of software is implemented in a computer where time progresses discretely according to the machine cycle of the computer.

For example, the gas burner, a software-embedded system, is used in an environment where time progresses continuously. However, the embedded software of the gas burner may run in a computer with a certain machine cycle, and interacts with other physical components via *sensors* and *actuators* which operate discretely.

Although the deadline-driven scheduler, a software system, is hosted in a computer where time progresses discretely, the correctness of the deadline-driven scheduling algorithm is expected to be independent of the specific host computer, i.e. the algorithm can be better understood in terms of continuous time.

Therefore, the interface between continuous time and discrete time has become an important research topic in designing real-time systems.

For DC, we have adopted *continuous* time and chosen real numbers to model this continuous time. Discrete time, as a countable subset of the real numbers, can be defined in DC. It is definitely true that not every requirement satisfiable in continuous time can be implemented by a computer. For example, no computer can send out two signals separated by a distance less than its machine cycle, although, because of the density of continuous time, one can always find two time instants with an arbitrarily small distance between them.

In [32], a subset of DC formulas is identified, from which discrete implementations called digital controllers can be synthesized. References [19, 20, 25, 137, 138, 156] introduce discrete states to approximate continuous states, and provide rules to refine continuous specifications expressed as DC formulas into discrete implementations.

1.1.3 State Models

In DC, *states* and *events* are used to model the behavior of real-time systems. However, the book concentrates on state models until Chap. 9, where state models are extended with the addition of events. A *Boolean state model* of a real-time system is a set $P_1, P_2, \ldots, P_i, \ldots$ of Boolean-valued (i.e. $\{0,1\}$-valued) functions over time, i.e.

$$P_i \ : \ \mathbb{T}\mathrm{ime} \ \to \ \{0,1\} \, ,$$

where $\mathbb{T}$ime is the set of the real numbers.

Each Boolean-valued function, also called a *Boolean state* (or simply a *state*) of the system, is a *characteristic* function of a specific aspect of the system behavior, and the whole set of Boolean-valued functions characterizes all of the relevant aspects of the behavior.

Deadline-Driven Scheduler

In order to prove the correctness of the deadline-driven scheduler, we introduce the following states to model the behavior of the scheduler:

$$\begin{aligned} \mathrm{Run}_i &: \mathbb{T}\mathrm{ime} \ \to \ \{0,1\} \\ \mathrm{Std}_i &: \mathbb{T}\mathrm{ime} \ \to \ \{0,1\} \\ \mathrm{Urg}_{ij} &: \mathbb{T}\mathrm{ime} \ \to \ \{0,1\} \, , \end{aligned}$$

for $i, j = 1, 2, \ldots, m$.

The states Run_i $(i = 1, 2, \ldots, m)$ are used to characterize the processor occupation. $\mathrm{Run}_i(t) = 1$ means that p_i is running in the processor at time t, while $\mathrm{Run}_i(t) = 0$ means that p_i is not running at t.

The states Std_i $(i = 1, 2, \ldots, m)$ characterize the standing of the request. $\mathrm{Std}_i(t) = 1$ means that at time t the current request of p_i is still standing. Namely, the current request of p_i is yet to be fulfilled at time t. $\mathrm{Std}_i(t) = 0$ means that at t the current request of p_i is not standing anymore. In other words, the current request of p_i has been fulfilled by time t.

For a pair of processes p_i and p_j $(i \neq j)$, the state Urg_{ij} describes which of the processes is more urgent, where urgency is defined in terms of the distance to the start of the next request period. Thus, $\mathrm{Urg}_{ij}(t) = 1$, for $i, j = 1, 2, \ldots, m$ and $i \neq j$, means that p_i is more urgent than p_j at time t, and $\mathrm{Urg}_{ij}(t) = 0$ means that p_i is less urgent than or as urgent as p_j at time t.

It is obvious that any set of the above functions which characterizes a possible behavior of the deadline-driven scheduler must satisfy certain properties. For example, at any time t, if $\mathrm{Run}_i(t) = 1$, then $\mathrm{Run}_j(t) = 0$ for $j \neq i$, as the processes share a *single* processor. The properties which capture the scheduling algorithm are more complicated.

DC provides a formal notation to specify the real-time properties of the scheduling algorithm in terms of states Run_i, Std_i and Urg_{ij}. Furthermore, the real-time requirement of the scheduler can also be expressed in DC through these states, and the correctness of the scheduling algorithm can then be verified using DC. □

A Boolean state model of a system represents an abstraction of the behavior of the system, and may be refined to more primitive states during the design and the implementation of the system. In particular, for designing a software-embedded system, a Boolean-valued state may be finally refined to real-valued functions which model the behavior of physical components of the system, as in control theory. We call the real-valued functions a *real state model* of the system. Consider the example of the gas burner.

Gas Burner

The gas burner is a software-embedded system. To verify the design decisions against the requirement, one may start with a single Boolean state to model the critical aspect of the system

$$\mathrm{Leak} : \mathbb{T}\mathrm{ime} \rightarrow \{0, 1\}\,,$$

where $\mathrm{Leak}(t) = 1$ means that gas is leaking at time t, and $\mathrm{Leak}(t) = 0$ means that gas is not leaking at t.

However, at a later stage of the design one may have to specify the phases of burning and idling of the gas burner, and introduce more primitive Boolean states of the system such as Gas and Flame to characterize the flowing and burning of gas. Then Leak can be pointwise defined as a Boolean expression containing Gas and Flame:

$$\mathrm{Leak}(t) \ \widehat{=} \ \mathrm{Gas}(t) \wedge \neg\mathrm{Flame}(t)\,,$$

for any $t \in \mathbb{T}\mathrm{ime}$.

Boolean operators (e.g. $\neg$ and $\wedge$) for states are therefore included in DC, so that a composite state of a real-time system can be refined to primitive states of the system.

However, the flow of gas is actually a real-valued function of time, and can be determined by the degree of opening of a gas valve. To describe the valve, a function

$$\mathrm{Valve} : \mathbb{T}\mathrm{ime} \rightarrow [0, \Theta]$$

is introduced, where $\mathrm{Valve}(t) = \theta$ means that the valve is opened to a degree θ $(0 \leq \theta \leq \Theta)$ at time t.

The Boolean state Gas can be regarded as an abstraction of the real-valued function Valve. For example, one may define this state such that gas is present at t when $\mathrm{Valve}(t)$ is above some threshold θ_0 $(0 < \theta_0 < \Theta)$:

$$\mathrm{Gas}(t) = \begin{cases} 1, \text{ if } \mathrm{Valve}(t) \geq \theta_0 \\ 0, \text{ otherwise} \end{cases} .$$

In other words, Gas becomes the characteristic function of a property of the real-valued function Valve.

Furthermore, the opening and the closing of the valve are controlled by a piece of software embedded in the gas burner, which governs the application of a force to open or close the valve. This applied force can be expressed as another real-valued function:

$$\mathrm{Force} \ : \ \mathbb{T}\mathrm{ime} \ \rightarrow \ [-\Omega, \Omega] \,,$$

where Ω stands for the greatest strength of the applied force. The real-valued functions Force and Valve are called *real states* of the gas burner, and join with other functions to form a *real state model* of the system. The relation between Force and Valve may be defined by a differential equation obtained from mechanics. □

As a design calculus for software-embedded systems, research on DC has explored possibilities to capture parts of real analysis (see [165, 170]), and hence to specify real state models of software-embedded systems. However, this book will not present a real state model.

1.1.4 State Durations

The notion of state *duration* is used to specify the behavior of real-time systems. The duration of a Boolean state over a time interval is the accumulated presence time of the state in the interval.

Let P be a Boolean state (i.e. $P : \mathbb{T}\mathrm{ime} \rightarrow \{0, 1\}$), and $[b, e]$ an interval (i.e. $b, e \in \mathbb{T}\mathrm{ime}$ and $e \geq b$). The duration of state P over $[b, e]$ equals the integral

$$\textstyle\int_b^e P(t)\, dt \,.$$

Let us use the two examples described above to illustrate the importance of state durations in specifying real-time behavior.

Deadline-Driven Scheduler

The real-time requirement of the scheduler is to fulfill all the process requests before their expiration. This requirement can be expressed in terms of durations of the states Run_i, for $i = 1, 2, \ldots, m$.

Let us assume that all the processes raise their first request at time 0. Thus, every nth request of p_i is raised at time $(n-1)T_i$ and expires at time nT_i, where $n = 1, 2, \ldots$. Therefore, the scheduler fulfills the nth request of p_i iff the accumulated run time of p_i in the interval $[(n-1)T_i, nT_i]$ equals the requested time C_i. Namely, the duration of state Run_i over the interval $[(n-1)T_i, nT_i]$ is equal to C_i:

$$\int_{(n-1)T_i}^{nT_i} \mathrm{Run}_i(t)\, dt = C_i \, .$$

Hence, the requirement is satisfied by the scheduler iff the duration of Run_i over the interval $[(n-1)T_i, nT_i]$ is equal to C_i, for all $i = 1, 2, \ldots, m$ and $n = 1, 2, \ldots$. □

Gas Burner

The real-time requirement of the gas burner is that the proportion of leak time in an interval is not more than one-twentieth of the interval, if the interval is at least one minute long. This requirement can be expressed in terms of the durations of Leak as follows:

$$(e-b) \geq 60\,\mathrm{s} \;\Rightarrow\; 20\int_b^e \mathrm{Leak}(t)\, dt \leq (e-b)\,,$$

for any interval $[b, e]$. □

A mathematical formulation of these two requirements can hardly leave out state durations. Since the processor may be preempted *dynamically*, the duration of Run_i extracts the accumulated running time of p_i from the dynamic occupation of the processor. Also, since gas leaks occur owing to *random* flame failures, the duration of Leak extracts the accumulated leak time of gas from the random flame failures. Therefore, state durations are adopted in DC to specify the behavior of real-time systems.

The *distance* between states (or events) is another important measurement of real-time systems. This was studied extensively before the development of DC, for example, by the use of timed automata [5], real-time logic [69], metric temporal logic [72] and explicit clock temporal logic [54].

However, state durations are more expressive than distances between states in the sense that the latter can be expressed in terms of the former, but not vice versa. With state durations, one can first express the *lasting period* of a state. That a presence of state P lasts for a period $[c, d]$ (for $d > c$), written $P[c, d]$, can be expressed as follows:

$$\int_c^d P(t)\, dt = (d-c) > 0\,,$$

if we do not care about instantaneous absences of P. This expression is read in real analysis as

"P appears almost everywhere in $[c, d]$" .

Thus, real-time constraints on the lasting periods of states can be expressed in terms of state durations.

Gas Burner

Consider the first design decision in the case of the gas burner. Let $[b, e]$ be an interval where we want to guarantee the requirement of the gas burner. The first design decision is that any leak in $[b, e]$ should not last for a period longer than one second. This can be expressed as

$$\forall c, d : b \leq c < d \leq e . (\text{Leak}[c, d] \;\Rightarrow\; (d - c) \leq 1\,\text{s}) .$$

Real-time constraints on distances between states can be expressed in terms of state durations similarly. Consider the second design decision in the case of the gas burner. The second design decision is that the distance between any two consecutive leaks in the guarantee period $[b, e]$ must be at least thirty seconds long:

$$\begin{aligned} &\forall c, d, r, s : b \leq c < r < s < d \leq e . \\ &\qquad (\text{Leak}[c, r] \wedge \text{NonLeak}[r, s] \wedge \text{Leak}[s, d]) \;\Rightarrow\; (s - r) \geq 30\,\text{s} , \end{aligned}$$

where NonLeak is a state defined from Leak using the negation ($\neg$):

$$\text{NonLeak}(t) \;\widehat{=}\; \neg\text{Leak}(t) ,$$

for any $t \in \mathbb{T}\text{ime}$.

The above formulation of the second design decision for the gas burner can be changed to a syntactically weaker but semantically equivalent one:

$$\begin{aligned} &\forall c, d, r, s : b \leq c < r < s < d \leq e . \\ &(\text{Leak}[c, r] \wedge \text{NonLeak}[r, s] \wedge \text{Leak}[s, d]) \;\Rightarrow\; (d - c) \geq 30\,\text{s} . \end{aligned}$$

The equivalence of these two formulas can be proved as follows. It is obvious that the first formula implies the second one. In order to prove the other implication, we assume that there are

$$c' < r < s < d' \text{ in } [b, e]$$

such that

$$\text{Leak}[c', r],\ \text{NonLeak}[r, s],\ \text{Leak}[s, d'] \text{ and } (s - r) < 30 .$$

Under this assumption, we let

$$\begin{aligned} \eta &= (30 - (s - r)) > 0 \\ c &= \max\{c', (r - (\eta/3))\} \\ d &= \min\{d', (s + (\eta/3))\} . \end{aligned}$$

Then, it is easy to prove that

$$c < r < s < d \text{ in } [b, e]$$

and

$$\text{Leak}[c, r],\ \text{NonLeak}[r, s],\ \text{Leak}[s, d] \text{ and } (d - c) < 30\,.$$

So, by the contraposition law of propositional logic, we complete the proof of the equivalence of the two formulations of the second design decision. □

However, the equivalence of these two formulas holds only for continuous time. In the rest of this book, when we are concerned with a continuous time domain, we shall adopt the second formulation, since it corresponds to a simpler formalization of the second design decision for the gas burner in DC. In Chap. 12 we shall deal with a discrete time domain and shall formalize the second design decision differently.

By axiomatizing integrals of Boolean-valued functions, DC provides a possible way to introduce notions of real analysis into formal techniques for designing software-embedded real-time systems. Notions of *integral* and/or *differential* have also been adopted in studies of automata [4, 99], statecharts [92], temporal logic of actions (TLA) [76] and communicating sequential processes (CSP) [55], when considering software-embedded systems.

State durations, as integrals of Boolean-valued functions, are functions from time intervals to real numbers. The state durations of DC have been axiomatized on the basis of the interval logics proposed in [1, 27, 43], which can be regarded as logics for functions of time intervals.

1.2 Interval Logic

By interval logic we mean logics in the sense of [1, 27, 43], for example. We view these logics as logics for time intervals. Let $\mathbb{I}\text{ntv}$ be the set of time intervals, i.e.

$$\mathbb{I}\text{ntv} \ \widehat{=}\ \{\, [b, e] \mid b, e \in \mathbb{T}\text{ime} \wedge b \leq e \,\}\,.$$

1.2.1 Interval Variables

In these logics, we can express properties of functions of time intervals, called *interval variables*.

Let v_i (for $i = 1, 2, 3, 4$) be interval variables, i.e.

$$v_i \ :\ \mathbb{I}\text{ntv} \ \rightarrow\ \mathbb{R},$$

where $\mathbb{R}$ denotes the set of real numbers.

A formula such as $v_1 \leq (v_2 + v_3 \cdot v_4)$ is interpreted in interval logic as a function from $\mathbb{I}\mathrm{ntv}$ to the truth values $\{\mathrm{tt},\mathrm{ff}\}$:

$$v_1 \leq (v_2 + v_3 \cdot v_4) \ : \ \mathbb{I}\mathrm{ntv} \ \to \ \{\mathrm{tt},\mathrm{ff}\}\,.$$

An interval $[b, e]$ satisfies the formula iff the value of v_1 of $[b, e]$ is less than or equal to the sum of the value v_2 of $[b, e]$ and the product of the values of v_3 and v_4 of $[b, e]$.

Therefore, interval logic provides a *functional* calculus for specifying and reasoning about properties of functions of intervals in a succinct way, such that the arguments of the functions (i.e. the intervals) are not referred to explicitly.

The interval *length* is a specific interval variable denoted ℓ, i.e.

$$\ell \ : \ \mathbb{I}\mathrm{ntv} \ \to \ \mathbb{R}\,.$$

For an arbitrarily given interval $[b, e]$, ℓ delivers the value $(e - b)$, i.e. the length of $[b, e]$.

The duration of the state P (written $\int P$) is another interval variable,

$$\textstyle\int P \ : \ \mathbb{I}\mathrm{ntv} \ \to \ \mathbb{R}\,.$$

For an arbitrarily given interval $[b, e]$, the value of the interval variable $\int P$ is the duration of P in $[b, e]$, i.e. the value

$$\textstyle\int_b^e P(t)\, dt\,.$$

Gas Burner

The requirement of the gas burner can be expressed in terms of the state duration $\int\mathrm{Leak}$ as

$$\mathit{GbReq} \ \widehat{=} \ \ell \geq 60 \ \Rightarrow \ 20\textstyle\int\mathrm{Leak} \leq \ell\,,$$

where 60 stands for 60 seconds. (Henceforth we choose the *second* as the time unit in the example of the gas burner.) □

1.2.2 Interval Modalities

The set of intervals $\mathbb{I}\mathrm{ntv}$ is the semantic domain of interval logic. In interval logic, *modalities* are used to define structures among intervals, such as one interval is a subinterval of another interval, or an interval is made of two adjacent subintervals. Those structures are present in the descriptions of the two design decisions for the gas burner. For example, the first design decision expresses a real-time property of a subinterval in which leaking occurs. The second design decision expresses a real-time requirement for three adjacent subintervals.

In the literature of mathematical logic, logics of modalities are called *modal* logics [15, 66]. The semantics domain of a modal logic is usually called a *frame* and it consists of a set of *worlds* and a *reachability relation* of the worlds. Thus, an interval logic is a modal logic which takes intervals as worlds.

In [1, 43, 147], twelve unary modalities and three binary modalities are suggested for defining various interval reachabilities. We list here four of the modalities, which are used later in this chapter.

The Subinterval Modality $\Diamond$

The subinterval modality $\Diamond$ (Fig. 1.2) is a unary modality. For any formula ϕ, $\Diamond\phi$ is a new formula which holds for an interval iff ϕ holds for some *subinterval.*

Mathematically, an arbitrary interval $[b, e]$ satisfies $\Diamond\phi$ iff there exist c, d such that $b \leq c \leq d \leq e$ and the interval $[c, d]$ satisfies ϕ. Thus, from the interval $[b, e]$ one can reach its subintervals with $\Diamond\phi$.

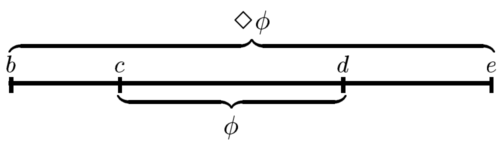

Fig. 1.2. The modality $\Diamond$

The dual of $\Diamond$ is $\Box$, which is defined as

$$\Box\phi \ \widehat{=}\ \neg\Diamond\neg\phi\,.$$

Hence, $[b, e]$ satisfies $\Box\phi$ iff any subinterval of $[b, e]$ satisfies ϕ.

With $\Box$, one can formulate the first design decision for the gas burner, that any leak in the guarantee period of the gas burner must be stoppable within one second.

First, the mathematical definition of $P[c, d]$ (i.e. P takes the value 1 almost everywhere in a nonpoint interval $[c, d]$) can be expressed as a formula without mentioning the interval explicitly:

$$\llbracket P \rrbracket \ \widehat{=}\ \textstyle\int P = \ell \ \wedge\ \ell > 0\,.$$

Then, the following formula is a formalization of the first design decision:

$$Des_1 \ \widehat{=}\ \Box(\llbracket \text{Leak} \rrbracket \ \Rightarrow\ \ell \leq 1)\,.$$

$\Box$

The Chop Modality $\frown$

The chop modality $\frown$ (Fig. 1.3) is a binary modality introduced into interval logic by [43]. For formulas ϕ and ψ, the new formula $\phi\frown\psi$ is satisfied by an interval iff the interval can be chopped into two adjacent subintervals such that the first subinterval satisfies ϕ and the second one satisfies ψ.

In other words, the interval $[b, e]$ satisfies the formula $\phi\frown\psi$ iff there exists m ($b \le m \le e$) such that $[b, m]$ satisfies ϕ and $[m, e]$ satisfies ψ.

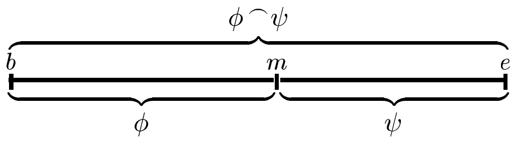

Fig. 1.3. The modality $\frown$

The reachability relation defined by $\frown$ is a ternary one. It provides access to adjacent subintervals of an interval, and hence defines a temporal *order* among subintervals of an interval.

With $\frown$ and $\Box$, one can formalize the second formulation of the second design decision for the gas burner given in Sect. 1.1.4:

$$Des_2 \;\widehat{=}\; \Box((\llbracket \mathrm{Leak} \rrbracket \frown \llbracket \neg\mathrm{Leak} \rrbracket \frown \llbracket \mathrm{Leak} \rrbracket) \;\Rightarrow\; \ell \ge 30)\,.$$

To prove the correctness of the two design decisions is therefore to prove the validity of the formula

$$(Des_1 \wedge Des_2) \;\Rightarrow\; GbReq\,.$$

In fact, the subinterval modality $\Diamond$ can be derived from the chop modality, since

$$\Diamond\phi \;\Leftrightarrow\; (\mathrm{true}\frown(\phi\frown\mathrm{true}))\,,$$

where "true" stands for a formula which is satisfied by any interval. Therefore, the second design decision (as well as the first one) for the gas burner can be expressed in an interval logic of state durations with $\frown$ as the only modality. $\Box$

A modality is called *contracting* if the modality provides access only to *inside* parts of a given interval, i.e. subintervals of the interval. $\Diamond$ and $\frown$ are two examples of contracting modalities. With the contracting modality $\frown$, we have expressed the two design decisions for the gas burner which can guarantee the *safety*-critical requirement of the gas burner.

However, contracting modalities cannot express *unbounded liveness* and *fairness* properties of computing systems, since these properties are about properties *outside* any given time interval. Modalities which provide access to the region outside a given interval are called *expanding* modalities. In the following we give two examples of expanding modalities.

The Right Neighborhood Modality $\Diamond_r$

The modality $\Diamond_r$ (Fig. 1.4) is a unary modality. An interval satisfies $\Diamond_r\phi$ iff a *right neighborhood* of the *ending* point of the interval satisfies ϕ.

Mathematically, $[b, e]$ satisfies $\Diamond_r\phi$ iff there exists $d \geq e$ such that interval $[e, d]$ satisfies ϕ.

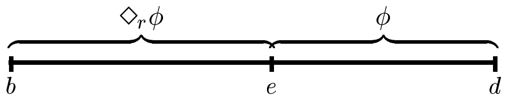

Fig. 1.4. The modality $\Diamond_r$

Thus, $\Diamond_r$ provides access to right neighborhoods of e from $[b, e]$. Since right neighborhoods of e are outside $[b, e]$, $\Diamond_r$ is an expanding modality.

The modality $\Box_r$ is the dual of $\Diamond_r$ and is defined as

$$\Box_r\phi \;\widehat{=}\; \neg\Diamond_r\neg\phi\,.$$

That is, an interval satisfies $\Box_r\phi$ iff any right neighborhood of the ending point of the interval satisfies ϕ.

With $\Diamond_r$, one can specify properties related to future time, such as liveness and fairness properties of computing systems. Consider the example of the gas burner. Let HeatReq be a state to characterize a request for heat from the gas burner. The formula

$$\llbracket \mathrm{HeatReq} \rrbracket \;\Rightarrow\; \Diamond_r(\textstyle\int \mathrm{Flame} > 0)$$

expresses the condition that if one raises a heat request, then there will exist a presence of Flame in the future. This formula can represent an additional requirement for the gas burner, to reject a safe but *dead* gas burner.

The Left Neighborhood Modality $\Diamond_l$

The modality $\Diamond_l$ (Fig. 1.5) is a unary modality. An interval satisfies $\Diamond_l\phi$ iff a *left neighborhood* of the *beginning* point of the interval satisfies ϕ.

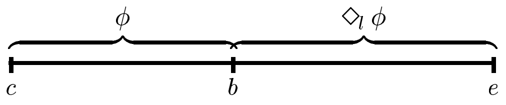

Fig. 1.5. The modality $\Diamond_l$

Mathematically, $[b,e]$ satisfies $\Diamond_l \phi$ iff there exists $c \le b$ such that interval $[c,b]$ satisfies ϕ.

Thus, the modality $\Diamond_l$ provides access to the past time of a given interval. It is also an expanding modality.

The dual of $\Diamond_l$ is designated by $\Box_l$. An interval $[b,e]$ satisfies $\Box_l \phi$ iff any left neighborhood of b satisfies ϕ:

$$\Box_l \phi \ \widehat{=} \ \neg \Diamond_l \neg \phi \, .$$

$\Box$

In Chap. 11 of this book, it is proved that all twelve unary modalities and three binary modalities of interval logic can be derived from $\Diamond_r$ and $\Diamond_l$ in a first-order logic with interval length ℓ. However, this book will use $\frown$ as the only modality, except in Chap. 11, where the liveness and fairness properties of computing systems are discussed.

1.3 Duration Calculus

Research on DC was initiated by the case study [145] in connection with the ProCoS project (ESPRIT BRA 3104 and 7071). Several real-time formalisms were investigated in order to specify requirements and design decisions for a gas burner system; but they all failed in this case study. Two main observations of this case study were that the notion of a time interval was useful and that the notion of a state duration was convenient. This led to the first publication on DC [168] in 1991. Since then, research on DC has considered different models of real-time systems, applications of DC and mechanical support tools for DC.

In [161], there is a brief overview of early research on DC, and in [51], there is a detailed account of the logical foundations of DC.

1.3.1 Models

Different models are used by designers of real-time systems at different design stages. In order to accommodate all necessary models, sets of functions over time, called *states*, are used to model real-time systems in DC. In the state models, real-valued functions are called *real states* of systems, and characteristic functions of properties of underlying real states are called *Boolean states.* Boolean states are assumed *stable*, i.e. any presence (or absence) of a Boolean state must last for some period, and are represented by Boolean-valued *step* functions. *Events* are taken to be transitions of Boolean states.

First, a basic calculus – the calculus for durations of Boolean states – was developed, and then other models were introduced by adding to the basic calculus extra axioms, which formalize the models and also their interrelations with the Boolean state model.

Boolean State Model

The basic calculus of DC [168] axiomatizes state durations for the Boolean state model, i.e. integrals of Boolean-valued functions, under an assumption of *finite variability* (also called the *non-Zeno phenomenon*) of states. The assumption of finite variability stipulates that any state can only change its presence and absence finitely many times in any bounded time period. That is, only finitely many state transitions can take place in any bounded time period. The interval modality used in the basic calculus is the chop modality $\frown$. This calculus can be used to specify and verify state-based safety properties of real-time systems. Formalizations of other models are *conservative* extensions of this calculus.

Boolean State and Event Model

The Boolean state and event model was studied in [164, 169].

In [169], an event is a Boolean-valued δ-function, i.e. a Boolean-valued function with a value of 1 at *discrete* points. This means that an event is an instant action, and an event takes place at a given time point iff the Boolean-valued δ-function of the event takes the value 1 at that point. By linking events to state transitions, this model can be used to refine from state-based requirements, via mixed state and event specifications, to event-based specifications of programs.

However, with integrals of functions, one cannot capture the value of a function at a point, since the integral of a function at a point is always equal to zero, no matter what the value of the function at that point is. In [169], integrals of Boolean-valued functions are replaced by their *mean values.* The mean value of a Boolean-valued function P, designated $\overline{P}$, is a function from intervals to $[0,1]$, i.e.

$$\overline{P} \ : \ \mathbb{I}\mathrm{ntv} \ \rightarrow \ [0,1]\,,$$

and is defined in real analysis as follows:

$$\overline{P}([b,e]) \;=\; \begin{cases} \int_b^e P(t)\,dt/(e-b) & \text{if } e > b \\ P(e) & \text{if } e = b \end{cases},$$

for any interval $[b,e]$.

Therefore, one can describe point properties of Boolean-valued functions by using their mean values in point intervals, and at the same time one can also define the integral of a Boolean-valued function P:

$$\textstyle\int P \;\widehat{=}\; \overline{P} \cdot \ell\,.$$

Additional axioms and rules for reasoning about δ-functions and state transitions were developed in [169].

The approach in [164] is to continue using the basic calculus for the integral of a Boolean-valued function, but atomic formulas to stand for events are added to the basic calculus. This book will follow the approach of [164] to introduce state transitions and events into the Boolean state model.

Real State Model

A real state model consists of a set of real-valued functions which describe the behavior of physical components of a software-embedded system. By using a real state model, we introduce structures into Boolean states, and a Boolean state becomes a characteristic function of a property of real states of the model. Therefore, specifications and reasoning at the level of the state may have to employ real analysis.

In [170], it was investigated how DC can be combined with real analysis, so that real state models can be specified within the framework of DC. In [165], this research was further developed by the formalization of some parts of real analysis using the left and right neighborhood modalities.

Dependability

The dependability of an implementation with regard to a given requirement can be quantitatively measured by a satisfaction probability of the requirement for this implementation.

In the context of the Boolean state model and a discrete time domain, the work presented in [86, 87, 89, 90] provides designers with a set of rules to reason about and calculate the satisfaction probability of a given requirement, formalized using DC, with respect to an implementation represented as a finite automaton with history-independent transition probabilities.

In [22], this work was generalized to a continuous time domain.

Finite-Divergence Model

The assumption of finite variability of states and events stipulates that within a finite time period, state transitions and events can happen only finitely many times. The finite-variability assumption is always adopted in the case of software systems where time progresses discretely.

The notion opposite to finite variability is called *finite divergence* (also called the *Zeno phenomenon*). Continuous mathematics does not reject finite divergence, and introduces the notion of a limit in order to study finite divergence. In [48], the finite-divergence model was formalized by introducing into DC some rules to calculate a state duration in a finite-divergence model as a limit of its approximations in a finite-variability model.

Superdense Computation

A *superdense computation* is a sequence of operations which is assumed to be timeless. This is an abstraction of a real-time computation within a context with a grand time granularity. This assumption is known as the *synchrony* hypothesis and has been adopted in the case of digital control systems, where the cycle time of an embedded computer may be nanoseconds, while the sampling period of a controller may be seconds. Therefore, the computation time of the embedded software of the digital control system is negligible, and computational operations can be abstracted as timeless actions.

To accommodate timeless operations, [164] adapts the chop modality and renames it the *superdense chop*. This can chop a time point in a grand time space into multiple points in a finer space, and hence the superdense chop introduces structure into a time point.

By generalizing the projection operator [97] of interval temporal logic, [42] introduced into DC the *visible* and *invisible* states, and computed non-negligible time through projection onto the visible state.

Thus, the properties of superdense computation can also be specified and verified in DC. In [107, 114], other approaches are considered for treating the synchrony hypothesis within the framework of DC.

Expanding Modalities

With contracting modalities such as $\frown$ and $\Diamond$, one can specify only *safety* properties of real-time systems.

In order to specify *unbounded liveness* and *fairness* properties of real-time systems within the framework of DC, [31, 103, 139, 165] introduced expanding modalities. In [165], it was proved that the left and right neighborhood modalities $\Diamond_l$ and $\Diamond_r$ are adequate, in the sense that the other contracting and expanding modalities suggested in [1, 43, 147] can be derived from them in a first-order logic with an interval length ℓ. The completeness of the first-order calculus for $\Diamond_l$ and $\Diamond_r$ given in [165] was proved in [9], and, in [8], the completeness was proved for a combination of a first-order temporal logic and an interval logic with neighborhood modalities.

In [31], an interval logic where intervals have a direction was suggested. This logic is based on the chop modality, but the "chop point" is allowed to be outside the interval under consideration, and in this way the chop modality becomes expanding. This logic, called *signed interval logic* (SIL), was further developed in [120, 123].

Infinite Intervals

The behavior of a real-time system, such as the deadline-driven scheduler or the gas burner considered here, is often assumed to be *infinite*. However, DC is a logic of finite intervals. An infinite behavior is therefore specified in DC as the set of all finite prefixes of the behavior. To specify liveness and fairness properties of the behavior of a system in terms of its finite prefixes, expanding modalities have been introduced.

An alternative to expanding modalities is to introduce infinite intervals into DC. Extensions of DC which allow infinite intervals were established in [117, 162]. These extensions include both finite and infinite intervals, and can straightforwardly express and reason about both *terminating* and *infinite* behaviors of real-time systems. References [117, 118, 119] also compare the expressive power of these extensions with the expressive power of monadic logic of order.

Higher-Order and Iteration Operators

When DC is applied to real-time programming, it becomes inevitable that one introduces advanced operators into DC corresponding to the programming notions of local variables and channels, and of the loop.

In [39, 41, 60, 108, 110, 163], the semantics and proof rules of the (higher-order) quantifiers over states and the μ operator were investigated. It is interesting to discover that, because of the finite variability of states, the quantifiers over states can be reduced to first-order quantifiers over global variables, and also that the superdense chop can be derived from the higher-order quantifiers.

1.3.2 Applications

The applications of DC focus on the formal design of real-time systems.

Case Studies of Software-Embedded Systems

DC has been applied to case studies of many software-embedded systems, such as an autopilot [126], a railway crossing [141] and interlock [127], a water level monitor [30, 64], a gas burner [127], a steam boiler [31, 83, 135], an air traffic controller [68], a production cell [113], a motor-load control system [157], an inverted pendulum [151], a chemical concentration control system [153], a heating control system [155], a redundant control system [36] and a hydraulic actuator system [125]. A case study for formalizing and synthesizing an optimal design of a double-tank control system was conducted in [62].

On the basis of these case studies, a methodology and notation for designing software-embedded systems were studied and developed in [16, 21, 149, 171].

Real-Time Semantics, Specification and Verification

In order to apply DC to the specification and verification of real-time systems, techniques for integrating DC with other formalisms such as CSP, phase transition systems, Verilog and RAISE have been developed in [37, 57, 59, 61, 78, 152], where DC has been used to define the underlying semantics. In [88], a uniform framework for DC and timed linear temporal logic was presented.

In [63], CSP, Object-Z and DC were combined into a uniform framework for the specification of processes, data and time, based on a smooth integration of the underlying semantic models.

In [58, 133, 134, 164, 166], DC was used to define the real-time semantics for OCCAM-like languages. In [164], it was assumed, in the semantics of an OCCAM-like language, that assignments and message passings take no time, and can form a superdense computation. In [171], a semantics was given to a CSP language with continuous variables which was proposed in [55] and can be used to describe software-embedded systems.

In [98], DC was used to define a real-time semantics for SDL, while [95] embedded a subset of DC into a first-order logic of *timed frames* and hence into SDL. Reference [109] defined a DC semantics for Esterel. Reference [71] proposed a DC semantics for a graphical language called Constraint Diagrams. Reference [46] gives, in terms of DC, a formal meaning of fault trees. References [37, 78] define a DC semantics for a timed RAISE Specification Language and [136, 173] define a DC semantics for Verilog. In [24, 146], a DC semantics was given to programmable logic controller (PLC) automata and, furthermore, a tool was developed for designing PLC automata from DC specifications.

In [52], DC was used to specify and reason about real-time properties of circuits. Reference [128] applied DC to prove the correctness of Fischer's mutual-exclusion protocol. References [17, 20, 25] specified and verified the correctness of the biphase mark protocol through DC. Reference [160] applied DC to specify and verify the deadline-driven scheduler, and [14, 26] presented formal specifications of several well-known real-time schedulers for processes with shared resources. In [112], DC was used to specify and verify properties of real-time database systems, and, in [49], DC was used to specify and analyze availability properties of security protocols.

Refinement of DC Specifications

In [94], there was a first attempt to define refinement laws for a restricted set of formulas of DC toward formulas called DC *implementables*, which describe properties such as timed progress and stability. A full exposition of these ideas is given in the monograph [124]. In this monograph, there is also a study of how to ensure that a set of implementables is *feasible*, i.e. that it is consistent and extendable in time. Techniques to refine a feasible set of DC implementables via a mixed specification and programming language into an executable program were developed in [100, 133, 134].

References [21, 74, 75, 132] represent work on refining DC formulas into automata. References [153, 154]proposed approaches to refining DC specifications into programs following the paradigms of the Hoare logic and the assumption-commitment logic.

1.3.3 Tools

Interesting results about the *completeness* of the calculi for interval modalities and state durations and about *decision* procedures and *model-checking* algorithms for DC subsets have been published.

In [27], the completeness of the interval logic described in Chap. 2 was proved for an *abstract* domain. A similar result was proved in [9] for the neighborhood logic described in Chap. 11. The duration calculus described in Chap. 3 has been proved to be *relatively* complete [50]. It can also be complete for an abstract domain if we use ω-rules as in [38].

Decidable subsets of DC and the complexity of decision algorithms were discovered and analyzed in [2, 18, 32, 35, 47, 79, 102, 115, 116, 131, 167]. In order to check whether state transition sequences of a subset of timed (even hybrid) automata satisfy a linear inequality of the state durations, [12, 70, 80, 81, 82, 84, 158, 159, 172] developed algorithms which employ techniques from linear and integer programming.

On the basis of the above results, a proof assistant for DC was developed in [93, 140, 144] as an extension of PVS [101], and a decision procedure [167] for DC was incorporated into this proof assistant. For example, the soundness proof in [50] of the induction rules for DC was checked by this proof assistant. Furthermore, several proofs used in case studies were checked in [140] using the DC extension of PVS, e.g. the studies of the simple gas burner system proposed in [168] and of the railway crossing proposed in [141]. In these applications of the proof assistant, errors in the original proofs were spotted. In [23], there is an analysis and comparison of the use of model-checking and logical-reasoning techniques.

In [142], a tool to check the validity of a subclass of DC was presented. Furthermore, [105] developed a tool (DCVALID) to check the validity of a subclass of discrete-time higher-order DC. In [150], DCVALID was used to verify the correctness of a multimedia communication protocol. In [34], a bounded model construction for discrete-time DC was presented, which was shown to be NP-complete.

The proof theory for signed interval logic was developed and investigated in [121, 122, 123], and SIL is encoded in the generic theorem prover Isabelle [111].

1.4 Book Structure

Chapter 2 (Interval Logic) develops the syntax, semantics, axioms and rules of a first-order interval logic. It is the logical foundation of the axiomatizations of DC models presented in this book. This first-order interval logic includes *chop* as its only modality, and it is complete for an *abstract* time domain. An abstract time domain is not necessarily the set of real numbers, but an arbitrary set which satisfies certain axioms.

Chapter 3 (Duration Calculus) presents the calculus for durations for the Boolean states. It is based on the interval logic described in Chap. 2, and the assumption of finite variability of states. The gas burner example is used in this chapter to explain the syntax, semantics, axioms and rules of DC.

Chapter 4 (Deadline-Driven Scheduler) specifies and verifies the deadline-driven scheduling algorithm in DC. This demonstrates an application of DC to a rather complicated software system.

Chapter 5 (Relative Completeness) proves the *relative* completeness of DC with respect to a continuous time domain represented by the set of real numbers. By relative completeness, we mean that, in the context of this continuous time domain, any valid formula of DC is provable in DC, provided any valid formula of interval logic can be taken as a theorem of DC.

Chapter 6 and 7 (Decidability and Undecidability) describe decidable and undecidable subsets of DC formulas in discrete and continuous time domains. The decidability of a subset of DC is proved by reducing the validity of a formula in the subset to the decidable emptiness problem of regular languages. The undecidability of a subset of DC is obtained by reducing the undecidable halting problem for two-counter machines to satisfiability of formulas in the subset.

Chapter 8 (Model Checking: Linear Duration Invariants) presents an algorithm to decide whether an implementation of a real-time system satisfies a requirement written in DC as a finite number of linear inequalities of state durations, where the implementation is taken to be a real-time automaton having an upper time bound and a lower time bound for each transition. The satisfaction problem is reduced by the algorithm to finitely many simple linear programming problems.

Chapter 9 (State Transitions and Events) introduces extra atomic formulas and axioms to express and to reason about state transitions and events. With this extension, one can refine state-based requirements into state and event mixed (or event-based) implementations. In this chapter, an implementation as a real-time automaton is verified for the gas burner example against the two design decisions.

Chapter 10 (Superdense State Transitions) treats the synchrony hypothesis, and introduces the superdense chop modality. With the superdense chop, this chapter presents a real-time semantics for an OCCAM-like language. In the semantics, it is assumed that assignments and message passings take no time.

Chapter 11 (Neighborhood Logic) introduces the left and right neighborhood modalities. It proves the adequacy of these two modalities, and applies them to specify unbounded liveness and fairness.

Chapter 12 (Probabilistic Duration Calculus) assumes that an implementation of a real-time system is represented by a *probabilistic* automaton having a probability distribution over discrete time for each transition. Axioms and rules are developed to calculate and reason about the satisfaction probability of a requirement, formalized using DC, for a probabilistic automaton over a specified time interval. The gas burner is used as an example to explain the notions and techniques involved.