Chapter 2
Languages and Interpretations

Everybody is familiar with the notion of language – at least to the extent that they can speak one. The scope of language, however, extends well beyond interhuman verbal exchange, and comprises any form of communication that takes place according to some rules. *Natural languages* originate in the natural world to support communication among people; the spoken languages – such as English, Italian, and Chinese – are obvious examples, but non-verbal natural idioms – such as gestures, pictures, and music – are also common. This book is about a different kind of language: *artificial languages* designed for the description and analysis of phenomena – in particular, their temporal aspects. The choice of artificial languages is also vast and varied, ranging from mathematical notation – algebra, graphs, mathematical logic – to programming languages – such as Java and Haskell – and communication protocols – such as Internet's TCP and HTTP.

Whether natural or artificial, every language is structured as a collection of *sentences*. Each sentence is an arrangement of elementary blocks from the language's *alphabet*. Often, the alphabet is finite and its elements are combined into linear sentences, such as the English alphabet and sentences, but infinite alphabets or more complex arrangements are also possible – for example, the alphabet of all sounds is uncountably infinite, and sentences develop bidimensionally in most visual graphic languages.

2.1 Syntax and Semantics

The definition of a language covers its *syntax* and its *semantics*. The rules of *syntax* define how to build correct sentences from the alphabet; in other words, syntax defines which sentences, from among all possible combinations of alphabet elements, are *well-formed* (acceptable sentences of the language) and which are not. Consider, for example, the English language. The English vocabulary defines

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_2,

all valid combinations of letters into words; the English grammar describes the rules to build sentences out of words, as in

> A *sentence* consists of a *noun phrase* followed by a *verb phrase*. The *noun phrase* is a *noun* (possibly preceded by a *determiner* such as an article) or a *pronoun*. The *verb phrase* is a *verb* (possibly followed by a *noun phrase* such as an object). The *number* of the noun in the noun phrase and of the verb in the verb phrase must agree.

The English grammar and vocabulary collectively describe the syntax of the language. Another example is the syntax of programming languages, such as Java:

> A *block* is a sequence of *elements* within braces. Each *element* is a *statement*, a *local class declaration*, or a *local variable declaration*.

Semantics associates a *meaning* to every syntactically well-formed sentence of a language; in other words, semantics connects the sentences – which are symbols – to an *interpretation* – which is the content they express, and which can belong to any domain, such as those of measurements, decisions, references to facts, and so on. For example, the natural language sentence

> In case of fire, do not use the elevator.

expresses a suggestion (possibly, an order) about how to behave in the case of a fire to minimize the risk of personal injury. The Java semantics associates with the sentence

> **if** $(x > 3)$ { $x = x + 1;$ } **else** { $x = x - 1;$ }

a behavior that depends on the variable x: if it evaluates to a value greater than 3, increment it; otherwise decrement it.

The semantics of a sentence is *ambiguous* if the sentence may have multiple different meanings. For example,

> Eats shoots and leaves.

may refer to a panda (whose diet consists of bamboo shoots and leaves) or to a gunman (who fires his weapon after eating, and then abandons the place), according to whether we interpret "shoots and leaves" as nouns or verbs. Conversely, syntactically different sentences may convey the same meaning according to the semantics. For example, the three sentences

> The gardener sprays water on the roses.
> The gardener sprays the roses with water.
> Water is sprayed on the roses by the gardener.

all essentially convey the same picture even if they combine words according to different structures. In a formal setting, the two sets of linear equalities

$$\begin{cases} x = 5\,, \\ y = 6\,, \end{cases} \qquad \begin{cases} x = 5\,, \\ y = x + 1\,, \end{cases}$$

indicate the same pair of values for x and y, but with different syntax.

2.2 Language Features

Given that the notion of language is very broad, very different classification criteria for languages are possible, according to the features of interest within a certain scope. For instance, understandability is very relevant in a teaching context, whereas conciseness is important when space is a concern.

Chapter 3 presents the dimensions that are specific to the central topic of the book, namely time modeling. The current section illustrates more generic features, applicable to languages independently of their application domains, which will also recur in the book's presentation of modeling languages.

2.2.1 Formality

The syntax and semantics of most natural languages are *informal*: even when there are standardized vocabularies and grammars, they lack absolute precision, and their interpretation may be ambiguous or subjective, so that the well-formedness or the meaning of certain sentences depends on the context in which they are used. Imprecision is the price to pay for naturalness: the attribution of meaning to sentences and idioms evolve without regulation, according to social customs and recurring practices; hence syntactic and semantic rules must accommodate unpredictable changes and the specialization of a stable language into dialects, jargons, and slangs.

In contrast, *formal* languages such as mathematics have been defined with an unambiguous syntax and a very precise semantics, so that the fact that a sentence such as

$$\int_0^{\pi} \cos(x)\mathrm{d}x \;=\; 0$$

is well formed and true is not subject to dispute.

Between the two extremes of natural languages and mathematics, there are several intermediate degrees of formality. Even if mathematics ultimately is fully formal, the presentation of mathematical theories and results often embeds mathematical notation in natural language text, such as in the book you are reading and in every other scientific textbook. In other cases, a language may have a formal syntax but a semantics that omits or overlooks some details. Such notations are often called "semiformal" languages.

Example 2.1 (UML). The Unified Modeling Language (UML) graphical notation is a diagrammatic notation widely used in software engineering. The UML standard defines the syntax of diagrams quite precisely, but their semantics only in natural language. This increases the flexibility of the notation, but it also implies that the meaning of UML diagrams is not always unambiguous. Take, for example, the UML *sequence diagrams* of Fig. 2.1.

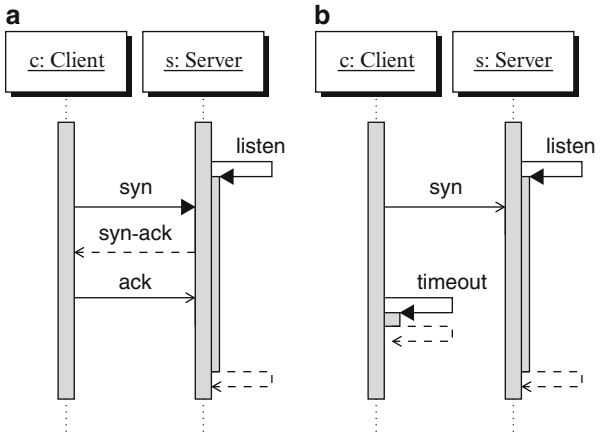

Fig. 2.1 UML sequence diagrams describing the "three-way handshake" of the TCP protocol

The UML diagrams are syntactically correct, and they arguably describe the "three-way handshake" of the TCP communication protocol in the cases (a) in which the communication is successfully established and (b) in which a time-out occurs after the syn message. This meaning is, however, only conveyed informally, and even within the level of abstraction provided by the diagrams several aspects have multiple possible interpretations. For example, the UML standard does not provide a precise meaning for the combination of the diagrams, so what happens if the syn-ack message is received after the time-out is triggered (is the communication successfully established or not?) is not well defined: our own intuition and knowledge of the protocol leads us to conclude that the communication is not successfully established, but this is not prescribed by the semantics of UML. ■

Programming languages are also often defined only semiformally. While there are standard notations to formalize *syntax* – the Backus-Naur form or variants thereof – their semantics is often described only informally using natural language. This is not only a matter of form, because informality in the definition of programming languages has the same drawbacks as it does in natural languages: the recurring practices take the place of a formal semantics in defining the correct "meaning", with the result that different, possibly incompatible, "dialects" of a language develop according to the compilers or techniques programmers use. Even widely used and standardized languages such as C have had to face these problems. Indeed, a complete and completely formal semantics is available only for a few general-purpose programming languages.

2.2.2 Medium

A language *medium* is the usual means by which sentences of the language are expressed. The preferred media of natural languages are speech and writing – with syllabic alphabets for most Western languages and ideogrammatic notations for several Asian languages – but they often encompass other complementary media such as gestures and facial expressions.

For more formal languages such as those described in this book, the primary medium is textual, over a finite alphabet (typically including Latin and Greek alphanumerical characters). A textual syntax is often supplemented by a graphical one, whose semantics may have different degrees of formality. One of the recurring themes of the book is the analysis of the most-delicate aspects that hamper the definition of a sound formal semantics, or make the semantics counterintuitive.

2.3 Languages for System Modeling

A *system* is a collection of *components* that work together within an *environment* to perform one or more *functions*.[1] To analyze and design systems, it is fundamental to construct system *models*: abstract representations of systems, which include their essential features and support analysis and prediction of their behavior.

A system model has several aspects:

Structure: the components in the system, and how they are connected and communicate (with one another and with the environment).

Behavior: how the components work, and how they interact with one another and with the environment.

Requirements: the function and goals that the system should achieve (relative to the environment).

"Modeling languages" are suitable notations to describe system models. Some modeling languages are sufficiently rich to be applicable to every aspect of the model; others target a specific one.

This book describes many modeling languages, with particular emphasis on models of *dynamic* systems: systems characterized by their behavior over time. We will develop the theme of time modeling thoroughly in the book; Fig. 2.2 suggests a very preliminary and informal view of the activity. Two equally informal examples follow.

[1] In fact, the word "system" derives from the Greek words $\sigma\upsilon\nu$ ("together") and $\iota\sigma\tau\eta\mu\iota$ ("to put, to compose").

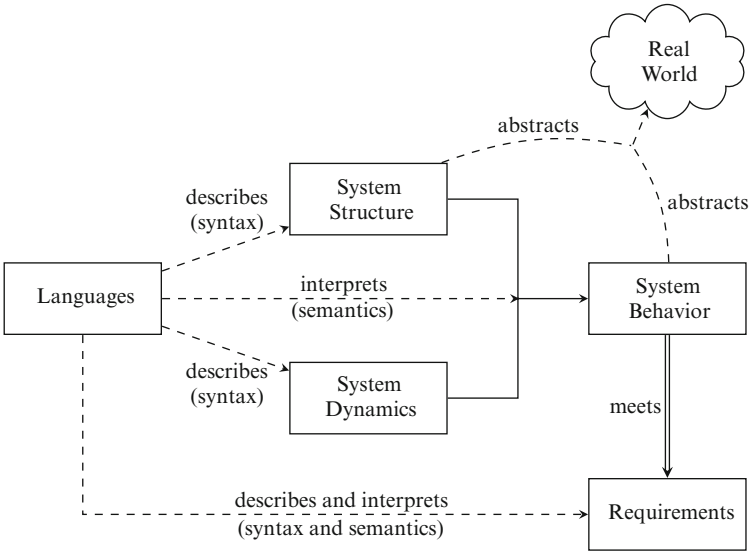

Fig. 2.2 Models, behaviors, languages, and requirements of dynamic systems

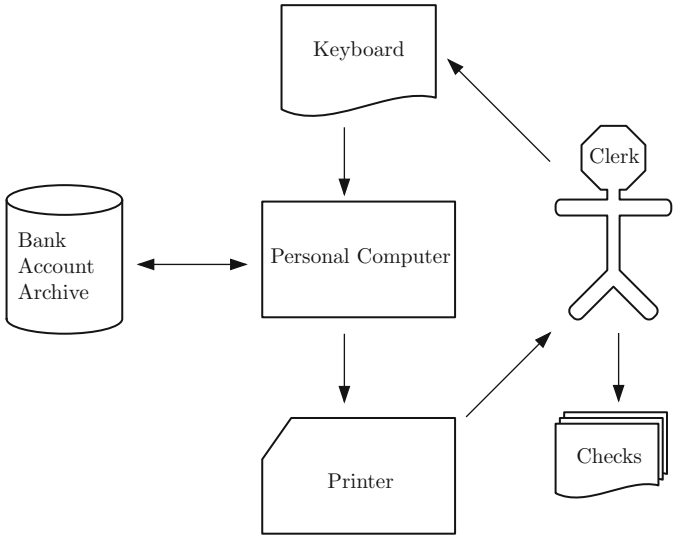

Fig. 2.3 The structural model of a simple banking system

Example 2.2. Figure 2.3 is an informal graphical model of the *structure* of a system where users interact with a personal computer through an input keyboard and an output printer; the computer can read and write a local file system.

The system *behavior* is also described informally, using natural language: the user is inserting the data about a collection of checks into the computer, where

a bank management system runs. For every check, she types the bank account numbers of the drawer and the recipient on the keyboard; then she types the amount and commits. Upon commit, the banking system records the information in the archive stored in the file system and prints a receipt slip for auditing.

The system *requirements* specify that the user click commit only if the system signals that the drawer's account has enough money. Notice that the requirements refer to the system as a whole, including the user whose behavior they constrain to achieve a defined goal. ■

Example 2.3. As an example of a system where timing is central, consider the dynamics of a braking car. A simple mathematical description models the car as a point mass of P kilograms, and its braking interaction with the road as a kinetic friction with friction coefficient of $\mu\text{m/s}^2$. Then, the system dynamic behavior specifies the car speed $v(t)$ at generic time t, assuming that the speed is v_0 when the braking starts, with the equation

$$v(t) \;=\; v_0 - \mu t. \tag{2.1}$$

An example of requirements for such a system is that if V denotes the maximum speed of the car, the breaking always takes no longer than T seconds and D meters. We can determine if the system behaves according to the requirements by analyzing (2.1) with the tools of elementary calculus and mechanics. ■

2.4 Operational and Descriptive Languages

The principle of "separation of concerns" is an engineering cornerstone, as it enables the analysis of complex systems by separating different dimensions. A special instance of the general principle is the "what vs. how" prescription about how to structure the engineering of a system: first define *what* to achieve, and then detail *how* to achieve it. Figure 2.2 adheres to this prescription by separating the *requirements* (what the system has to achieve) from the structural and behavioral *design* (how the system meets the requirements). At a lower level of abstraction, the duality between *specification* and *implementation* mirrors the one between requirements and design.

Modeling languages often target only one of several concerns. Some languages are explicitly designed, or simply better suited, for describing system behavior as sequences of transitions between configurations; we call these languages "operational". Other notations lend themselves to describing and formalizing system requirements abstractly; we call these languages "descriptive".

Abstract machines and dynamical systems are paradigmatic examples of operational notations, which describe the evolution of the *state* as a reaction to the input stimuli; for this reason, *state-based* is a synonym of operational. Logic is instead a classic instance of descriptive language, which formalizes *properties* and

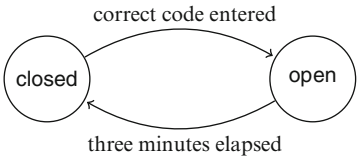

Fig. 2.4 A two-state machine describing the safe of Example 2.4 operationally

implications. The classification into operational and descriptive is, however, largely a matter of style and conventions, and the same language can often describe both transitions between configurations and system properties, as the following example demonstrates.

Example 2.4 (Natural language model of a safe).

Operational formulation: When the last digit of the correct security code is entered, the safe opens; then, if the safe remains open for three minutes, it automatically closes.

Descriptive formulation: The safe is open if and only if the correct security code has been entered no more than three minutes ago. ■

The remainder of the current chapter gives a few sketchy examples of operational and descriptive notations; the rest of the book gives a much more extensive and systematic presentation of several formalisms in both categories.

2.4.1 Operational Formalisms

Example 2.3 is distinctly operational, because Eq. (2.1) describes the evolution of the state (the speed v) over time from an initial value (v_0). Correspondingly, the derived behavior is also operational; for example, the displacement $x(t)$ of the car at time t is computed through elementary kinematics as

$$x(t) \;=\; \int_0^t (v_0 - \mu t)\,\mathrm{d}t \;=\; v_0\,t - \frac{\mu}{2}t^2\,, \tag{2.2}$$

which is valid only until the car reaches a full stop at time v_0/μ.

The possible states (that is, the values of x and v over time) in the car example range over a bounded interval of the real numbers. In contrast, the operational description of the safe in Example 2.4 suggests only two distinct states, corresponding to the safe being *open* or *closed*; the actions of *opening* the safe and *closing* it correspond to transitions between the two states. Figure 2.4 gives a graphical representation of the safe's operational model using a widely used graphical convention – which the rest of the book will use and improve on several occasions.

The operational model of Fig. 2.4 is very abstract, as the two transitions summarize actions that consist of sequences of simpler events. For example, entering the

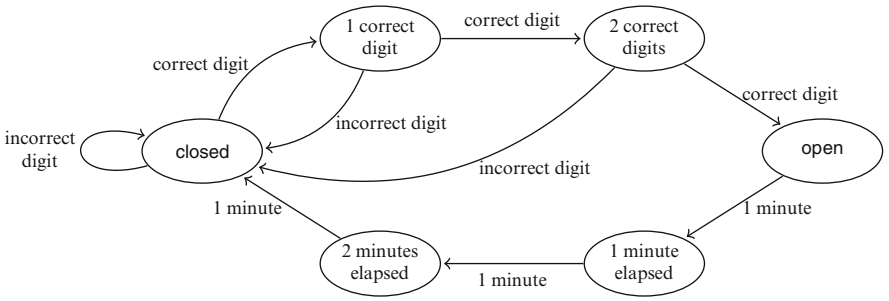

Fig. 2.5 A state machine refining the model of Fig. 2.4

three-digit code does not happen instantaneously, but one digit at a time; if one of the digits is incorrect, a new typing of the code must start over. Correspondingly, we can introduce two intermediate states to "count" the number of correct digits entered. Similarly, we can refine the elapsing of three minutes of time into a sequence of intermediate states, one per minute. Figure 2.5 represents this more detailed model with the same graphical notation of Fig. 2.4.

Exercise 2.5. Extend the safe operational model of Fig. 2.5 to accommodate:

- A command "stay open", which resets the time-out counter when the safe is open.
- A command "leave open", which stops the time-out counter until another command "close now" is issued.
- A security mechanism that, whenever an incorrect digit is entered, makes it impossible to enter a new code for the next two minutes.
- A further security mechanism that guarantees that the user is notified of the incorrectness of the code only after all digits have been entered. ■

General-purpose imperative programming languages – C, Pascal, Java, Eiffel, etc. – essentially are operational notations, as their programs consist of *instructions* that change the state when executed. Indeed, a common approach to defining the formal semantics of programming languages is *operational*: the effect of each instruction is defined in terms of how it affects the state (memory) of an abstract machine (e.g., a formalization of the Java Virtual Machine for Java). In contrast, logic programming languages – such as Prolog and Curry – indirectly describe computations through the defining properties of their output; hence it is natural to formalize their semantics with some form of logic.

2.4.2 Descriptive Formalisms: Mathematical Logic

Mathematical logic originates from the efforts of ancient philosophers (Aristotle, in particular) to model the rules of reasoning rigorously. Logic is a very versatile

language, suitable for describing facts and properties, and hence a fundamental descriptive notation. This section presents the basics of mathematical logic and gives an idea of the kind of descriptive models that logic can express.

2.4.2.1 Propositional Logic

Propositional logic (also called "propositional calculus") is the simplest variant of mathematical logic, at the core of every more expressive logic language. Propositional logic sentences are called "formulae". Formulae are built out of a countable infinite alphabet of "propositional letters" (or simply "propositions"), which are just character identifiers such as

$$A,\ B,\ \ldots,\ X,\ \ldots,\ \mathsf{open},\ \mathsf{closed},\ \ldots,$$
$$\mathsf{1\_minute\_elapsed},\ \ldots,\ \mathsf{Train\_1\_faster\_than\_Train\_2},\ \ldots\ .$$

Propositional letters are combined with "logical connectives" (also called logical "operators"), $\neg$ ("not"), $\wedge$ ("and"), $\vee$ ("or"), $\Rightarrow$ ("implication"), and $\Longleftrightarrow$ ("double implication", "co-implication", or "equivalence"), according to the following rules:

(i) Every propositional letter L is a well-formed formula;
(ii) If F, G are well-formed formulae, then the following are well-formed formulae:
   (a) $\neg F$ ("not F"),
   (b) $F \wedge G$ ("F and G"),
   (c) $F \vee G$ ("F or G"),
   (d) $F \Rightarrow G$ ("F implies G"),
   (e) $F \Longleftrightarrow G$ ("F if and only if G").

When multiple connectives are present in the same formula, the binding power decreases, in the same order as above, from $\neg$ down to $\Longleftrightarrow$; as in mathematics, parentheses are used to enforce a different order of application of connectives. Thus, for example,

$$A \wedge B \Rightarrow C \Longleftrightarrow \neg D$$

is the same as

$$((A \wedge B) \Rightarrow C) \Longleftrightarrow (\neg D)\,.$$

Given a well-formed formula in propositional logic, an *interpretation* is an assignment of a value in the domain $\{\mathbf{True}, \mathbf{False}\}$ to every predicate letter appearing in the formula. The assigned values are *truth values*, because they declare which propositions hold and which do not in the interpretation. The truth value of a formula follows from the interpretation of its propositions according to the following rules (which mirror the natural language meaning of logic connectives):

$\neg F$	is **True**	if and only if	F is **False**;
$F \wedge G$	is **True**	if and only if	both F and G are **True**;
$F \vee F$	is **True**	if and only if	F or G, or both, are **True**;
$F \Rightarrow G$	is **True**	if and only if	F is **False**, or else both F and G are **True**;
$F \Longleftrightarrow G$	is **True**	if and only if	F and G are both **True**, or both **False**.

Exercise 2.6. The rules show that some of the connectives are redundant, in that they are subsumed by the others. More precisely, two formulae are equivalent if, for every interpretation of propositions, they are always both **True** or both **False**. Show that:

- Every double implication $A \Longleftrightarrow B$ is equivalent to $(A \Rightarrow B) \wedge (B \Rightarrow A)$.
- Every implication $A \Rightarrow B$ is equivalent to $\neg A \vee B$.
- De Morgan's laws: Every disjunction $A \vee B$ is equivalent to $\neg(\neg A \wedge \neg B)$, and every conjunction $A \wedge B$ is equivalent to $\neg(\neg A \vee \neg B)$. ■

A propositional logic formula is "valid" when it evaluates to **True** for every interpretation of the propositions; it is "satisfiable" when it evaluates to **True** for some interpretation (at least one). A valid formula is **True** entirely on the basis of its propositional structure, independently of any interpretation; for example, $A \vee \neg A$ is valid because every propositional letter evaluates to **True** or **False**.

Example 2.7 (Propositional logic model of the safe). The simple descriptive model of the safe (Example 2.4) translates into a propositional formula over the propositions open, closed, and correct_code_entered_within_three_minutes:

$$\begin{array}{l} (\mathsf{open} \;\Rightarrow\; \neg\mathsf{closed}) \quad \wedge \\ (\mathsf{open} \;\Rightarrow\; \mathsf{correct\_code\_entered\_within\_three\_minutes}) \end{array} \tag{2.3}$$

Consider an extension of the safe model, where a command "stay open" forces the safe to stay open indefinitely. The propositional logic model would accommodate this extended behavior with new propositions to represent the command and whether the safe was opened in the past:

$$\mathsf{open} \Longleftrightarrow \left(\begin{array}{c} \mathsf{correct\_code\_entered\_within\_three\_minutes} \\ \vee \left(\begin{array}{c} \mathsf{safe\_opened\_in\_the\_past} \\ \wedge \\ \mathsf{stay\_open\_issued\_since\_last\_opening} \end{array} \right) \end{array} \right). \tag{2.4}$$

■

The semantics of propositional logic defines the truth value of a composite formula from the truth value of its simpler components, down to the propositional letters. "Deduction systems" leverage this feature to derive the truth value of every

formula from a set of simpler ones, in a calculational fashion. The basic formulae are called "axioms" and define some a priori knowledge about the system modeled: every axiom is a formula assumed to be true. The deduction system's *inference rules* describe how to derive new true formulae from the axioms. The inference rules are typically independent of the modeled system, which only affects the axioms, and are universal for the chosen logic. True formulae derived by applying some inference rules are called "theorems" in mathematical logic jargon.

Deduction systems can also provide an alternative definition of the semantics of propositional logic, where we can "calculate" properties of systems specified in propositional logic by applying inference rules to a set of standard axioms in addition to those specific to the system description. It is sufficient to use the universal inference rule of "Modus Ponens",

> If a formula F holds and the implication $F \Rightarrow G$ holds, then the formula G holds,

and consider for any formulae F, G, and H the axioms

(AX1) $F \Rightarrow (G \Rightarrow F)$;
(AX2) $(F \Rightarrow (G \Rightarrow H)) \Rightarrow ((F \Rightarrow G) \Rightarrow (F \Rightarrow H))$;
(AX3) $(\neg F \Rightarrow \neg G) \Rightarrow (G \Rightarrow F)$

in addition to the system description axioms. In the case of the safe of Example 2.7, the formula

$$\neg\, \mathsf{correct\_code\_entered\_within\_three\_minutes} \Rightarrow \mathsf{closed} \tag{2.5}$$

specifying that the safe is closed if the correct code has not been entered in the last three minutes is a system property. We derive (2.5) from the axioms of propositional logic and the system-specific axiom (2.3), with repeated applications of Modus Ponens (we abbreviate open, closed, and correct_code_entered_within_three_minutes with O, C, and E).

(D1)	$O \Rightarrow E$	special case of (2.3)
(D2)	$\neg(\neg O) \Rightarrow \neg(\neg E)$	equivalent form[2] of (D1)
(D3)	$\neg O \Rightarrow C$	special case of (2.3)
(D4)	$(\neg(\neg O) \Rightarrow \neg(\neg E)) \Rightarrow (\neg E \Rightarrow \neg O)$	instance of (AX3)
(D5)	$\neg E \Rightarrow \neg O$	Modus Ponens with (D2), (D4)
(D6)	$(\neg E \Rightarrow (\neg O \Rightarrow C)) \Rightarrow ((\neg E \Rightarrow \neg O) \Rightarrow (\neg E \Rightarrow C))$	instance of (AX2)
(D7)	$(\neg O \Rightarrow C) \Rightarrow (\neg E \Rightarrow (\neg O \Rightarrow C))$	instance of (AX1)
(D8)	$\neg E \Rightarrow (\neg O \Rightarrow C)$	Modus Ponens with (D3), (D7)
(D9)	$(\neg E \Rightarrow \neg O) \Rightarrow (\neg E \Rightarrow C)$	Modus Ponens with (D8), (D6)
(D10)	$\neg E \Rightarrow C$	Modus Ponens with (D5), (D9)
		QED

[2]The equivalence is formally derivable from the same axioms.

The simplicity of propositional logic stems from the usage of atomic items (the propositional letters) to indicate arbitrarily complex facts and actions through their truth values. This very abstract view, however, restricts the expressiveness of the logic, and makes it often too limited to model and reason about entities with complex structures and behavior. Even in the simple Example 2.7, we had to coalesce different types of information into the simple propositional letters: for example, the proposition correct_code_entered_within_three_minutes conflates information about the entering of the code with the fact that the code was correct, and with timing information about when it last happened. This is not very flexible and generalizes poorly. What happens, for example, if we want to formalize the fact that the safe is open at a generic time? If we associate instants of time with the natural numbers $0, 1, 2, \ldots$, we can introduce a denumerable sequence of propositions open_0, open_1, open_2, ... Nonetheless, it is impossible to write formulae that mention all such propositions, and hence explicitly define the state of the safe at every instant. Things are even worse if we assume a continuum of time instants, for example, corresponding to the nonnegative real numbers; in this case, the required number of propositions is not available even in principle.

These observations call for an extension of propositional logic that supports atomic elements more complex than propositions. In particular, it should support *parametric* elements, and the ability to express properties about elements evaluated for infinitely many different values of the parameters. Such parametric elements are called "predicates"; predicate logic extends propositional logics to accommodate them. The rest of this chapter describes the basics of predicate logic, which underpins many other formal languages discussed in the rest of the book.

2.4.2.2 Predicate Logic

Predicate logic (also called "predicate calculus", or "first-order logic") extends propositional logic with variables, functions, Boolean predicates, and quantifiers. More precisely, the alphabet of predicate logic includes:

- Symbols for *constants* $a, b, c, \ldots$;
- Symbols for *variables* $t, u, v, w, x, y, z, \ldots$;
- Symbols for *functions* $f, g, h, \ldots$;
- Symbols for *predicates* $P, Q, R, \ldots$;
- The logic quantifiers $\forall$ ("for all", universal quantifier) and $\exists$ ("there exists", existential quantifier);
- The same logic connectives as propositional logic.

For notational convenience, the constant, function, and predicate symbols can include standard mathematical symbols such as $+$, $-$, sin, $<$, $=$, $>$, and π, with their usual syntax (e.g., infix binary operators) whenever useful.

Predicate logic builds well-formed formulae incrementally from the alphabet as follows. First, constants, variables, and functions are combined into *terms*:

(i) A constant c and a variable x are terms;
(ii) If f is an n-ary function and $t_1, t_2, \ldots, t_n$ are n terms, then the function application $f(t_1, t_2, \ldots, t_n)$ is a term.

Second, terms and predicates are combined into *atomic formulae*:

(iii) If P is an n-ary predicate and $t_1, t_2, \ldots, t_n$ are n terms, then $P(t_1, t_2, \ldots, t_n)$ is an atomic formula.

Finally, atomic formulae are combined with quantifiers and logic connectives into well-formed formulae of arbitrary complexity:

(iv) Every atomic formula is a well-formed formula;
(v) If F is a well-formed formula and x is a variable, then both $\forall x(F)$ and $\exists x(F)$ are well-formed formulae;
(vi) If F is a well-formed formula, then $\neg F$ is a well-formed formula;
(vii) If F and G are well-formed formulae and $\star$ is any binary connective of propositional logic ($\wedge$, $\vee$, $\Rightarrow$, $\Longleftrightarrow$), then $F \star G$ is a well-formed formula.

Example 2.8. Let us show a few examples of well-formed formulae of predicate calculus.

- Every well-formed propositional formula is also a well-formed predicate formula, because propositional letters correspond to argumentless predicates, and hence to atomic formulae.
- Every mathematical equality or inequality is a well-formed formula, because equality and other relational operators are binary predicates (usually written in the infix form). For example:
  - $x = x + 1$,
  - $\sin^2(x) + \cos^2(x) = 1$,
  - Bernoulli's inequality: $(1 + x)^r \geq 1 + rx$.
- More generally, mathematical statements are expressible in the language of predicate logic. For example:

  No natural number equals its successor: $\forall n(n \neq n + 1)$,
  The sine of π is 4 or 2: $\sin(\pi) = 4 \ \vee \ \sin(\pi) = 2$,
  Bernoulli's inequality holds for every nonnegative integer r and nonnegative real x: $\forall r(\forall x(r \in \mathbb{Z} \wedge r \geq 0 \wedge x \in \mathbb{R} \wedge r \geq 0 \Rightarrow (1+x)^r \geq 1 + rx))$. ■

While reading the examples above, you have probably tried to figure out which formulae express true facts and which are false. Intuition based on mathematical knowledge is sufficient for these simple examples, but being able to do it systematically for every formula requires the precise *semantics* of predicate logic. Given a well-formed first-order formula, an "interpretation" associates a value of a suitable domain with every variable and constant symbol, and a concrete function and relation with every function and predicate symbol. In practice, mathematical symbols and relations will be interpreted to convey the usual meaning adopted in mathematics. From the interpretation of the basic symbols, we know the value of

every term, and the truth value of every atomic formula. Finally, the truth value of a generic formula follows compositionally from the same rules of the propositional calculus semantics. The only new operators are the quantifiers, whose treatment requires a few more details:

- In every quantified formula $\forall x(F)$ or $\exists x(F)$, F is the quantifier's "scope".
- Every occurrence of a variable x within the scope of a quantifier $\forall x$ or $\exists x$ is called "bound"; a variable occurrence that is not bound is "free".[3]
- A universally quantified formula $\forall x(F)$ evaluates to **True** if F evaluates to true for *every* possible interpretation of x.
- An existentially quantified formula $\exists x(F)$ evaluates to **True** if F evaluates to true for *some* possible interpretation of x.

In general, the truth value of a formula depends on the interpretation given to constants, variables, functions, and predicate symbols. With the same definition as in propositional logic, a well-formed predicate formula is "valid" if it evaluates to **True** for *every* possible interpretation of constants, variables, predicates, and functions; it is "satisfiable" if it evaluates to **True** for *some* interpretation.

Example 2.9. Let us consider the truth value of some of the formulae in Example 2.8.

- Under the standard interpretation of mathematical functions and predicates over numerical domains, logic truth coincides with mathematical truth (of course!),
  - $x = x + 1$ evaluates to **False**, regardless of the value assigned to x by the interpretation;
  - Correspondingly, $\forall n(n \neq n + 1)$ evaluates to **True** under every interpretation of n.
- However, under nonstandard interpretations of constants, functions, and predicates, the semantics may become counterintuitive:
  - $x = x + 1$ evaluates to **True** under the interpretation where $+$ is not an ordinary sum, but is the function that returns its first argument: $x + 1$ evaluates to x, and hence the equality holds;
  - $\sin(\pi) = 4 \vee \sin(\pi) = 2$ is **False** with the standard interpretation for 2, 4, π, and sin, but it is **True** in other interpretations, such as the one when sin denotes a constant function with value 4.
- A well-formed formula F is "closed" if no variable appearing in F is free. The truth value of a closed formula is independent of the interpretation of its variables (but it may still depend on the domain of variables and on the interpretation of constants, functions, and predicates). For example, $\forall x(\exists y(y = x + 2))$ is **True**

[3] For simplicity, assume that different quantifier instances bind variables with different names. This is without loss of generality, and it helps keep the presentation plain.

over the domain of natural numbers for the standard interpretation of equality and sum, regardless of the values chosen for x, y. ■

If functions and predicates can have arbitrary interpretations, the semantics of predicate formulae may seem detached from the practice of mathematics, and of limited practical utility. In fact, predicate calculus is a very powerful modeling language, mainly when used in combination with some "first-order theories" that enforce the intended semantics of functions and predicates. A first-order theory supplements predicate calculus with a set of *axioms*. As in the deduction systems mentioned in the context of propositional logic, axioms are well-formed formulae that are assumed to be **True**; in other words, in a first-order theory we only consider interpretations that satisfy the axioms. Validity and satisfiability are redefined for a theory T accordingly: a formula is "T-valid" if it evaluates to **True** in every interpretation where T's axioms also evaluate to **True**; it is "T-satisfiable" if it evaluates to **True** in some interpretation where T's axioms evaluate to **True**. The details of how to describe the characteristic properties of arithmetic and other mathematics are quite involved and outside the book's scope. The rest of the book will simply assume they are available whenever needed to reconcile intuition with strictly formal reasoning.

Example 2.10 (Predicate logic model of the safe). With predicate calculus, we can make the descriptive model of the safe more detailed than in Example 2.7, which used only propositional logic. To this end, we introduce the two variables t, u to denote time; for simplicity, we can assume that they vary over the integers, but other options could be accommodated along the same lines. The propositions for the safe being open or closed and correct code being entered – used in the propositional model of Example 2.7 – become predicates with t or u as a parameter, so that their truth is time-dependent. A predicate formula that specifies the behavior of the safe is the following:

$$\forall t \left(\begin{array}{l} (\mathsf{open}(t) \iff \neg \mathsf{closed}(t)) \\ \quad \wedge \\ (\mathsf{open}(t) \iff \exists u \, ((t - 3 \leq u < t) \ \wedge \ \mathsf{correct\_code\_entered}(u))) \end{array} \right). \tag{2.6}$$

■

Exercise 2.11. Provide a predicate logic extension of the propositional formula (2.4), along the lines of Example 2.10. ■

The various examples have demonstrated the versatility of logic as a descriptive formalism: the formulae describe systems through their characterizing *properties* of interest, rather than as explicit sequences of transitions and reached states. Anyway, as remarked at the beginning of the current section, the distinction between operational and descriptive is largely a matter of style. The following example provides evidence of this fact by sketching an operational model of the safe formalized with predicate calculus. Similarly, we will discuss the converse

correspondence between state-based formalisms as models of logic formulae in the context of temporal logic (Chap. 9) and dual-language approaches (Chap. 11).

Example 2.12 (An operational model with logic). Using the predicates of Example 2.10, the following formula translates the state-based operational model of Fig. 2.4; the first line is the usual mutual exclusion between states (implicit in the model of Fig. 2.4); the second line describes the transition from closed to open and the permanence of open for three minutes; the last line specifies that the state closed does not change unless a correct code is entered (also implicit in the model of Fig. 2.4):

$$\forall t \left(\begin{array}{l} (\mathsf{open}(t) \iff \neg \mathsf{closed}(t)) \\ \wedge\ (\mathsf{correct\_code\_entered}(t) \Rightarrow \forall u((t+1 \leq u \leq t+3) \Rightarrow \mathsf{open}(u))) \\ \wedge\ \left(\forall u \left(\begin{array}{l} (t \leq u \leq t+2) \\ \quad \Rightarrow (\mathsf{open}(u) \wedge \neg \mathsf{correct\_code\_entered}(u)) \end{array} \right) \Rightarrow \mathsf{closed}(t+3) \right) \\ \wedge\ (\mathsf{closed}(t) \wedge \neg \mathsf{correct\_code\_entered}(t) \Rightarrow \mathsf{closed}(t+1)) \end{array} \right). \tag{2.7}$$

The rest of the book will provide plenty of examples of operational and descriptive formalisms; their strong connections with the basic "universal" languages introduced in the present chapter will be apparent. ■

2.5 Bibliographic Remarks

Pinker wrote several fascinating books about the origins, role, and evolution of natural languages [14–16].

Version 2 of the Unified Modeling Language (UML) is a standard of the Object Management Group [4, 8, 21]. Stevens discusses TCP and other Internet protocols in depth [20]. Every compiler construction book discusses the Backus-Naur form or extensions thereof [1,2,6,22]; Knuth traces back the origins of the notation [10]. The formalization of the semantics of programming languages now has a rich history and many comprehensive texts [12, 13, 17, 18]. These books typically abstract away some of the low-level details of real programming languages, but others have tried to formalize a specific language in its entirety, as Stärk et al. [19] did for Java.

For more specific references on operational and logic formalisms, see the bibliographic remarks at the end of other chapters – in particular, Chaps. 4, 5, 7, 8, and 11 for state-based notations and Chaps. 9 and 11 for logic-based formalisms. Mendelson [11], Enderton [7], and Kleene [9] are classic general introductions to mathematical logic; Ben-Ari [3] and Bradley and Manna [5] present the same basic topics from a computer science perspective.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley, Boston/London (2006)
2. Appel, A.W.: Modern Compiler Implementation in ML. Cambridge University Press, Cambridge/New York (2004)
3. Ben-Ari, M.: Mathematical Logic for Computer Science, 2nd edn. Springer, London/New York (2003)
4. Booch, G., Rumbaugh, J., Jacobson, I.: Unified Modeling Language User Guide, 2nd edn. Addison-Wesley, Upper Saddle River/Boston (2005)
5. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer, Berlin (2007)
6. Cooper, K., Torczon, L.: Engineering a Compiler, 2nd edn. Morgan Kaufmann, Burlington (2011)
7. Enderton, H.B.: A Mathematical Introduction to Logic, 2nd edn. Academic, San Diego (2001)
8. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd edn. Addison-Wesley, Boston (2003)
9. Kleene, S.C.: Mathematical Logic. Dover, Mineola (2002)
10. Knuth, D.E.: Backus normal form vs. Backus Naur form. Commun. ACM **7**, 735–736 (1964)
11. Mendelson, E.: Introduction to Mathematical Logic, 5th edn. Chapman and Hall, London (2009)
12. Meyer, B.: Introduction to the Theory of Programming Languages. Prentice Hall, New York (1990)
13. Mitchell, J.C.: Concepts in Programming Languages. Cambridge University Press, New York (2002)
14. Pinker, S.: The Language Instinct: How the Mind Creates Language. William Morrow, New York (1994)
15. Pinker, S.: Words and Rules: The Ingredients of Language. Basic Books, New York (1999)
16. Pinker, S.: The Stuff of Thought: Language as a Window into Human Nature. Viking Adult, New York (2007)
17. Reynolds, J.C.: Theories of Programming Languages. Cambridge University Press, Cambridge/New York (1998)
18. Riis Nielson, H., Nielson, F.: Semantics with Applications: An Appetizer. Springer, New York/London (2007)
19. Stärk, R.F., Schmid, J., Börger, E.: Java and the Java Virtual Machine: Definition, Verification, Validation. Springer, Berlin/New York (2001)
20. Stevens, W.R.: TCP/IP Illustrated, vol. 1–3. Addison-Wesley, Reading (1994)
21. UML 2.0. http://www.omg.org/spec/UML/2.0/ (2005)
22. Wirth, N.: Compiler Construction. Addison-Wesley, Harlow/Reading (1996)

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