Chapter 1

The Logic of Engineering Design

A scientific theory takes the form of an equation or a set of equations and inequation, usually expressed in the language of mathematics. We will use the general logical term *predicate* to stand for such mathematical descriptions. The purpose of the predicate is to describe and therefore predict all possible observations that can be made directly or indirectly of any system from a given reproducible class. The values obtained by physical measurement are denoted by free variables occurring in the formulae. An example is Einstein's famous equation

$$e = mc^2$$

where e is the energy of the system

m is its mass

and c is the speed of light.

Considerable familiarity with physics is needed to correlate the variables e and m and the constant c with the physical reality which they refer to. Nothing but confusion would result if the names of these variables were replaced by arbitrary alternatives, for example: $f = nd^2$.

The same physical system may be described at many different levels of abstraction and granularity; for example, physics can describe the material world as a collection of interacting quarks, or elementary particles, or atoms, or molecules, or crystal structures. All these descriptions lie well below the common-sense level of the physical objects which we see and touch in the everyday world. Science has discovered independent theories for reasoning at each of these levels of abstraction. But even more impressive is the demonstration that each theory is logically based on the more detailed theory below it. This is the strongest argument for the soundness of each separate theory, and also for the coherence of the entire intellectual structure of modern physics.

Mathematical theories expressed as predicates play an equally decisive role in engineering. A significant engineering project begins with a specification describing as directly as possible the observable properties and behaviour of the desired product. The design documents, formulated at various stages of the project, are indirect descriptions of the same behaviour. They are expressed in some restricted notation, at a level of abstraction appropriate to guide the physical implementation. This implementation is correct if its detailed description logically implies its specification; for then any observation of the product will be among those described and therefore permitted by the specification. The success of the whole project depends not only on correct reasoning at each level of design, but also on the soundness of the transition between the decreasing levels of abstraction appropriate to the successive stages of specification, design and implementation.

An engineering product is usually delivered as an assembly of components, which in general operate concurrently. The operation of each component can be described scientifically by a specific predicate, describing all its possible behaviours, including all its possible interactions with any other components which may be connected to it in an assembly. For this reason, the joint behaviour of many components in the assembly can often be described by the conjunction of these predicates describing the components separately. If some aspect of the component cannot be fully determined in advance, it may be described by the disjunction of predicates describing its alternative modes of behaviour. Finally, the links between predicates describing behaviour at different levels of granularity and abstraction can be formalised by quantification. In summary, the elementary operators of propositional and predicate logic provide all the basic concepts needed for a systematic engineering design methodology.

The goal of this book is to apply the philosophy and methods of modern science and engineering to the study and practice of computer programming. A computer program, and each of its components, is treated as a mathematical formula, describing the experimental observations which may be made by executing it in any variety of circumstances. The same phenomena may be described at several different levels of abstraction; the consistency of the various presentations can be assured by mathematical calculation and proof. The theory is developed first for a very simple programming language, which is then extended piecemeal; at each stage, it is hoped to preserve the validity and simplicity of what has gone before. The eventual goal is to develop theories which may be useful in the engineering of computer software, from specification of requirements through modular design to reliable implementation, installation and subsequent maintenance.

The full treatment of a simple programming language starts in Chapter 2. The remainder of this chapter develops the general philosophy of our approach to the science of programming and its application to software engineering.

1.1 Observations and alphabets

The first task of the scientist is to isolate some interesting class of reproducible system for detailed study. At the same time, a selection must be made of those properties which are regarded as observable or controllable or generally relevant to understanding and prediction of system behaviour. For each property, a name is chosen to denote its value, and instructions are given on how and when that property is to be observed, in what unit it is to be measured, etc. The list of names is usually accompanied by a declaration of the type of value over which each of them ranges, for example

$$x : \text{integer}, \quad y : \text{real}, \quad \ldots, \quad z : \text{Boolean}$$

This collection of names is known as an *alphabet*, and the names will occur as free variables, together with physical constants and other mathematical symbols, in any predicate describing the general properties of the system. They are called variables because their values vary from one experiment to another, conducted on a different system at a different place and time. Each theory in each branch of science is determined and delimited by its own choice of alphabet, and every formula and predicate of the theory has its free variables restricted to that alphabet.

We therefore require that every predicate P has associated with it (usually implicitly) a known set of free variables which it is allowed to contain. This is known as the *alphabet* of the predicate, abbreviated as αP. Often the predicate will contain all the variables of its alphabet, but there is no compulsion for it to do so. If the predicate has nothing to say about an observation x in its alphabet, that variable does not have to occur; or if preferred, it may be included vacuously by adding the trivial equation $(x = x)$. Of course, it is always forbidden for a predicate to contain free variables outside its alphabet.

An *observation* of a particular example of the chosen class of system can be expressed as a very limited kind of predicate. It consists only of a set of equations, ascribing particular constant values to each of the variables in the alphabet, for example

$$x = 4 \ \wedge \ y = 37.3 \ \wedge \ldots \wedge \ z = \text{false}$$

In logic, such observations are called *valuations* or *interpretations*; in computing, the instantaneous *state* of a machine executing a program is often recorded in this way as a "symbolic dump" of the values of all its variables at the time the observation is made.

Most of the dynamic variables treated in the physical sciences are functions which map a continuously varying time to a continuum of possible values. Consider for example a simple tank containing a liquid. The variables of its alphabet selected to describe the state of the system might be:

t (measured in seconds) is the time since the start of the apparatus ($t = 0$),
v_t stands for the measurable volume of liquid in the tank (in litres),
x_t is the total amount of liquid poured into the tank up to time t,
y_t is the total amount drained from the tank up to time t,
a_t is the setting of the input valve at time t (in degrees),
b_t is the setting of the output valve at time t.

The relationship between the symbolic variables and the real measurements that they denote is often illustrated by an annotated picture (Figure 1.1.1).

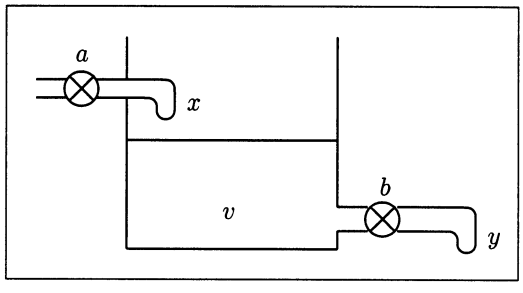

Figure 1.1.1 The water tank

In constructing a simple theory for a sequential programming language, we decide that the only relevant times at which observations can be made of its state are before the program starts and after it terminates; the decision to ignore all intermediate values of the variables is based on the reasonable hope that they will not be needed in the formulation of the theory. At the relevant times it is possible to record the values of all the global variables accessed or updated by the program. We will therefore need two names for each variable x: the final value will be recorded under the dashed name x', whereas the name x itself will stand for the initial value. An observation of a single completed execution of a particular program will give both the values observed, before and after; for example

$$x = 5 \wedge x' = 7 \wedge y = 2$$

may be an observation of a particular run of the program

$$x := x + y$$

The criterion of successful choice of an appropriate set of observations is nothing less than the success of the scientific theory that predicts their values. As Einstein remarked, it is the theory that determines what is observable. If there are too many observables, the theory gets too complicated; too few variables will make the theory inaccurate or restrict its range of application; and in general, the wrong choice will incur both penalties. In the following chapters, there will be examples of such failure, from which useful lessons will be drawn.

1.2 Behaviour and predicates

The normal discourse of scientists and engineers is wholly dependent on an agreed interpretation in reality of the alphabet of names used to describe observations or measurements. Only this makes it possible to perform experiments and record the observations against the relevant observation name. Scientific investigation often starts with a long, detailed and accurate record of particular observations of particular systems, and actual individual observations continue to play a decisive role throughout the later development of scientific knowledge. But the real purpose and goal of science are to replace this merely historical record by a theory of sufficient accuracy and power to predict the observations that will be made in experiments that remain to be performed in the future. A scientific theory is usually expressed as a mathematical *predicate* – an equation or an inequation or a collection of such formulae – which contain as *free* variables the names which have been selected to denote observations. Consider an observation

$$x = 12 \ \wedge \ y = 37.3 \ \ \wedge \ldots \wedge \ \ z = \text{ false}$$

A predicate $P(x, y, \ldots, z)$ correctly describes this particular observation if substitution of each variable by its observed value makes the predicate true (*satisfies* it)

$$P(12,\ 37.3,\ \ldots,\ \text{false})$$

But the purpose of the predicate is to describe in general a whole class of system, and it does so correctly if it describes every possible observation of every possible experiment made on any member of the class.

A useful scientific predicate is one that is as strong as possible, subject to the constraint of correctness: in general, it should be false exactly when its variables take combinations of values which in reality never occur together. This recommendation is violated by the weakest possible predicate, namely **true** (or equivalently, $x = x \wedge y = y$), which is satisfied by every conceivable observation. It therefore correctly describes every system, and it is useless because it does so. The strongest predicate **false** is equally useless for the opposite reason: there is no system which it describes. (If there were, it would necessarily have no observations, and therefore be inaccessible to science.) All of science is concerned with predicates that lie strictly between the two extremes of absolute logical truth and absolute falsity. Only such intermediate predicates convey useful information about what goes on in the world that we happen to live in.

Predicates describing the world are sometimes called Laws of Nature. They often take the form of a conservation principle or a differential equation. In the example of the water tank, the obvious law of conservation of liquid defines an invariant relationship between total inflow, total outflow and retained volume

$$x_t + v_0 = y_t + v_t \pm t \times \epsilon, \quad \text{for all } t \geq 0$$

where $\epsilon \leq \epsilon_{max}$, the maximum rate of accumulation of errors due to seepage, evaporation, precipitation, condensation, overflow, etc.

Other physical constraints may be imposed as inequations on the setting of the valves

$$0 \leq a_t \leq amax, \quad 0 \leq b_t \leq bmax$$

Finally, the relationship between the valve settings and the flow of liquid may be expressed by differential equations, say

$$\dot{x} = k \times a \quad \text{and} \quad \dot{y} = k \times b + \delta \times v$$

where δ accounts for extra outflow due to pressure of water in the tank, and $\dot{x}$ and $\dot{y}$ are the rates of change in x and y.

This collection of mathematical equations and inequations is strong enough to make a prediction about future water volumes, given sufficiently precise knowledge of the valve settings and the other variables and constants in the system.

The fundamental constituent of a sequential program is an assignment statement, for example

$$x := x + y$$

This causes the final value of x to be equal to the sum of the initial values of x and y. The effect is formally described by the predicate

$$x' = x + y$$

No restriction is placed on the initial or final values of any other global variables. Since no assignment is made to y, we choose to say nothing of the final value y'. Later, we will even exclude y' from the alphabet of the predicate. But still, the predicate can be used to help predict the final value of x on every single execution of the program.

A multiple assignment evaluates a list of expressions, and assigns their values to corresponding members of a list of variables; for example, the assignment

$$x, y := x + 3,\ y - x$$

is captured by the predicate

$$x' = x + 3 \ \wedge \ y' = y - x$$

This clearly states that all the expressions on the right of the assignment must be evaluated with the initial values of all the variables that they contain, and only

then is it allowed to assign the new values to the variables on the left of the assignment. In the interests of consistency, we require that all the assigned variables (on the left of :=) be distinct. The simplicity of this treatment of general assignment justifies the use of undashed variables to stand for initial values, and dashed variants (which are not allowed in the programming language itself) to stand for final values.

Scientists and engineers are entirely familiar with the practice of describing systems by predicates with an understood alphabet of free variables. They habitually transform, manipulate, differentiate and integrate textual formulae containing free variables, which have an external meaning independent of the formulae in which they occur. Such practices have been decried by pure mathematicians and logicians, who tend to use bound variables and closed mathematical abstractions like sets, functions and (less commonly) relations. But the conflict is only one of style, not of substance. Every predicate $P(x, y, \ldots, z)$ can be identified with the closed set of all tuples of observations that satisfy it

$$\{(x, y, \ldots, z) \mid P(x, y, \ldots, z)\}$$

Conversely, every formula S describing a set of observations can be rewritten as a predicate

$$(x, y, \ldots, z) \in S$$

A preference for the closed or open style of presentation of theories can influence a lifelong choice between specialisation as a pure or an applied mathematician. The preferences of pure mathematicians are explained by their main concern, which is the proof of mathematical theorems – formulae without free variables which are equivalent to the predicate **true**. Since our concern is primarily with descriptions of physical systems, we shall prefer to use predicates containing free variables selected from an alphabet whose existence, composition and meaning can only be explained informally by relating them to reality. In fact we have already begun to identify systems with descriptions of their behaviour, so that we can combine, manipulate and transform the descriptions in a manner which corresponds to the assembly and use of the corresponding systems in the real world. In this, we will see that the universal truth of abstract mathematical theorems, so useless for direct description of reality, plays an essential role in validating the transformations applied to such descriptions by scientists and engineers, and among them the engineers of software.

1.3 Conjunction

Propositional logic provides many ways of constructing complex predicates from simpler ones; the first and most important of these is undoubtedly *conjunction*, which we write as $\wedge$. If it needs definition, the following will suffice:

1. An observation satisfies a conjunction $(P \wedge Q)$ **iff** it satisfies both P and Q.
2. The alphabet of $(P \wedge Q)$ is the union of the separate alphabets of P and Q.

Conjunction is extremely useful in describing the behaviour of a product that is constructed from (say) two components with known behaviour, described individually by the two predicates P and Q. Consider first the simple but important case, when the alphabets of P and Q are *disjoint*, containing no variable in common, for example $\alpha P = \{x\}$ and $\alpha Q = \{z\}$. Then their conjunction $(P \wedge Q)$ describes the behaviour of two completely separate components, the first of which is described by $P(x)$ and the second by $Q(z)$. There is no connection between the components, and no synchronisation or coordination of their behaviour. Each observation (say, $x = 7 \wedge z = 3$) of their joint behaviour can be split in two: one part ($x = 7$) involves only variables from the alphabet of P, and this part satisfies $P(x)$; the rest of the observation ($z = 3$) similarly satisfies $Q(z)$. That is exactly the condition under which the whole observation satisfies $(P \wedge Q)$.

As an example, consider two programs operating on entirely disjoint collections of global variables. The two programs can be executed together in either order, or even in parallel. The combined effect of their execution is most simply described as the conjunction of the separate effects of each component. We use $\|$ to denote parallel execution of two assignments

$$x := x + z \ \| \ y := y - w$$

Their combined behaviour is precisely described by the conjunction of their separate behaviours

$$(x' = x + z) \wedge (y' = y - w)$$

It can be seen that the effect is the same as the multiple assignment

$$x, y := x + z, \ y - w$$

In most cases of interest, components of an engineering product are connected together in such a way that they can interact and thereby affect each other's behaviour. In principle, such an interaction can also be observed from the outside, and the observation can be recorded in some variable, say y. The interaction and its observation belong simultaneously to the behaviour of *both* the components which participate in it. The physical possibility of interaction is therefore represented by the fact that the variable y belongs to the alphabet of *both* of the predicates P and Q that describe them. In order for an observation (say, $x = 7 \wedge y = 12$) of one component P to be coupled with an observation (say, $y = 12 \wedge z = 3$) from the other component Q, it is essential that both observations give the *same* value to all the variables they share (in this case, just y), so that the coupling gives (say) $x = 7 \wedge y = 12 \wedge z = 3$. Such a coupled observation can still be split into two overlapping parts, one of which contains only the variables in the alphabet of

P, and this part satisfies $P(x, y)$; and the same for the other part, which satisfies $Q(y, z)$. That is the exact condition for the whole observation to satisfy the conjunction $(P(x, y) \wedge Q(y, z))$ of the component descriptions; its alphabet is clearly still the union of their separate alphabets.

A simple example of conjunction has already been given in the previous section. The three components of the water system are the tank, which obeys the conservation law, and the input valve and output valve, which regulate the flow in accordance with separate differential equations. The behaviour of the whole system is described exactly by the conjunction of the predicates describing the behaviour of its components. The variables in common to the three parts must obviously take the same value wherever they occur.

Conjunction is the general method of modelling connection and interaction in an assembly constructed from two or more components. But in practice, not all combinations of predicates are physically realisable in the available technology, for example by connection of wires in hardware, or by execution of programs in some software system. Any scientific theory which is to be useful in engineering practice must clearly state the general conditions under which assembly of components with non-disjoint alphabets will be physically realisable. If these conditions are violated, the resulting conjunction of specifications could be contradictory, yielding the predicate **false**, which is a logical impossibility and could never be implemented in practice. Avoidance of such inconsistency is a necessary goal of the more complex theories described later in this book.

A very effective way of achieving the necessary consistency is to distinguish which variables in the alphabet of each subsystem are "controlled" by that subsystem itself, rather than the environment in which it may be connected. These are called *dependent variables* or *outputs*, and form the *output alphabet* ($out\alpha P$) of the subsystem P. The necessary constraint is that each variable of a complete subsystem can be controlled by only one of its components. So the conjunction $(P \wedge Q)$ is forbidden unless the output alphabets are disjoint

$$out\alpha P \cap out\alpha Q = \{\}$$

A variable controlled by any subsystem is controlled in the combined system

$$out\alpha(P \wedge Q) = out\alpha P \cup out\alpha Q$$

The *input alphabet* is just defined as the rest of the variables, often called *independent variables*, controlled by the experimenter or the environment of the system

$$in\alpha P = \alpha P \setminus out\alpha P$$

The output alphabet of a sequential program consists of all its dashed variables; that is those that are allowed to appear on the left of an assignment within

it. The restriction on sharing the output variables is sufficient to ensure consistency of the conjunction of the predicates describing any two parallel programs. We can therefore relax the usual constraint against one component using variables updated by another parallel component. For example, we can allow

$$(x := x - y \parallel y := 2 \times y) \quad = \quad (x' = x - y) \wedge (y' = 2 \times y)$$

This again gives the same result as a multiple assignment. When one of the component expressions refers to a variable (e.g. y) updated by the other, our theory requires that it is the *initial* value of that variable that is obtained. In general, an implementation might have to make a private copy of such variables before executing the programs in parallel, as is standard when forking a process in UNIX^{TM} [157]. Consequently in this theory parallel components can never interact with each other by shared variables. We will see that such interactions can lead to highly non-deterministic effects. These may well be worth avoiding, even at the cost of extra copying (which is needed anyway on a distributed implementation, one with disjoint stores). A general theory for parallel programs which interact by updating the same variables is considerably more complicated, because it needs to take into account the intermediate values of program variables during execution.

1.4 Specifications

We have seen the role of predicates in describing the actual behaviour of individual components of an assembly. In suitable circumstances, the behaviour of the whole assembly is described by the conjunction of the predicates describing its components. The result can be used by the scientist to predict or even control the outcome of individual experiments on the assembly. But once the theory has been confirmed by experiment, it has an even more valuable role in reasoning about much more general properties of much wider classes of system: there is no longer any need to amass more data by individual experiment. In software engineering, lengthy experimentation is the usual method to determine the properties of a program – it is called program testing. As in natural science, the main reason for theoretical studies is to replace the bulk of tests by mathematical calculations based on the theory, and verified by just a few crucial experiments.

In engineering practice, predicates have an additional role as *specifications*, which define the purpose of a product by describing the desired properties of a system which does not yet exist in the real world, but some client, with money to pay, would like to see it brought into existence. A predicate used as a specification should describe the desired system as clearly and directly as possible, in terms of what behaviour is to be exhibited and what is to be avoided. The specification often defines part of a formal or informal contract between the client and the team engaged on implementation of the product.

In an industrial control system, a primary requirement is to hold some controlled variable within certain safety limits. In the example of the water tank, we impose a lower limit *minv* and an upper limit *maxv* on the volume of liquid held

$$minv \leq v_t \leq maxv, \qquad \text{for all } t$$

These express absolute limits on v, which must be maintained at all times.

Sometimes there are undesirable states which are permitted occasionally, but only for a relatively short proportion of the total time. For example, we may wish that the volume should not be above $(maxv - \delta)$ for more than 10% of any consecutive interval of ten seconds. The undesirable condition can be defined as a Boolean function of time (taking values 0,1)

$$\text{risk}_t \quad =_{df} \quad (v_t + \delta > maxv)$$

Now the requirement is expressed using integrals

$$\int_t^{t+10} \text{risk}_x dx \leq 1, \qquad \text{for all t}$$

The overall safety specification includes the conjunction of the two requirements displayed above. A complete specification will usually be a conjunction of a much larger number of separately expressible requirements.

Another lesson that may be drawn from this example is that, in the formalisation of a specification, one should not hesitate to use notations like those of real numbers and integrals, chosen from the entire conceptual armoury of mathematics: whatever will express the intention as clearly and directly as possible. Often, the notations cannot be executed or even represented on a computer, but they are well understood by the process physicists and control engineers, who must carry the ultimate responsibility for approving the statement of requirements before implementation begins.

An example more familiar to programmers is *sorting*. One of the requirements of a program that sorts data held in an array A is that the result A' should be sorted in ascending order of key. An array is regarded as a function from its indices to its elements. Let *key* be the function which maps each element to its key. The desired condition is

$$(key \circ A') \text{ is monotonic}$$

where $\circ$ denotes function composition. A second requirement on the program is that the result should be a permutation of the initial value

$$(\exists p : p \text{ is a permutation} \bullet A' = A \circ p)$$

The overall specification is the conjunction of these two requirements. Note that

the alphabet constraints described in the previous section prevent the program from being implemented as a conjunction of components which meet the two requirements separately.

Again, this example uses fairly sophisticated concepts from pure mathematics to achieve brevity at a high level of abstraction. For example, one must know the definition of a monotonic function, that it preserves the ordering of its arguments

$$x \leq y \;\Rightarrow\; A'(x) \leq A'(y), \qquad \text{for all } x, y \text{ in the domain of } A'$$

Other more diffuse formulations of the sorting concept may be shown to be equivalent to it. And they should be, if that increases confidence that the specification describes exactly what the customer has in mind. Even when safety is not involved, it is extremely wasteful and embarrassing to implement and deliver a product which turns out to do what was never wanted.

Knuth has claimed that every interesting concept in computing science can be illuminated by the example of the greatest common divisor [111]. The most direct way of specifying the greatest common divisor z of two numbers x and y is that it must be a divisor of both its operands, and the greatest such. This can be formalised as

$$\begin{aligned} x, y \geq 1 \;\Rightarrow\; & x \bmod z = 0 \wedge \\ & y \bmod z = 0 \wedge \\ & (\forall w : x \bmod w = y \bmod w = 0 \bullet w \leq z) \end{aligned}$$

This specification has the form of an implication, where the antecedent $(x, y \geq 1)$ states the general condition under which it is reasonable to ask for calculation of the greatest common divisor. The user of the product must undertake to make the antecedent true, because if it fails, there is no constraint whatsoever on the behaviour of the product.

In all the examples described above, individual requirements placed on the system have been formalised as separate predicates; like the components of an assembly, these are collected together by simple conjunction, but now unrestricted by the constraints of implementation technology. As a result, the conjunctive structure of a clear specification is usually orthogonal to the structure of its eventual implementation. The essential intellectual content of engineering design lies in correctly transforming the conjunctive structure of a specification to the orthogonally structured conjunction of components assembled to implement the specification. Engineering would be essentially trivial if a fast and economical product could be assembled from two components, one of which was fast and the other one economical.

1.5 Correctness

The previous two sections have shown both systems and specifications are (conjunctions of) predicates, describing all actual behaviour and all desired behaviour respectively. This gives a particularly convenient definition of the concept of correctness: it is just logical implication. Let S be a specification, composed perhaps as a conjunction of many individual requirements placed on the behaviour of a system yet to be delivered. Let P be a description of all the possible behaviours of the eventually delivered implementation, composed perhaps as the conjunction of the description of its many components. Assume that P and S have the same alphabet of variables, standing for the same observations. We want assurance that the delivered implementation meets its specification, in the sense that none of the possible observations of the implementation could ever violate the specification. In other words, every observation that satisfies P must also satisfy S. This is expressed formally by universally quantified implication

$$\forall v, w, \cdots \bullet P \Rightarrow S$$

where $v, w, \cdots$ are all the variables of the alphabet. Dijkstra and Scholten [52] abbreviate this using square brackets to denote universal quantification over all variables in the alphabet

$$[P \Rightarrow S]$$

Logical implication is the fundamental concept of all mathematical reasoning; it plays a crucial role in deducing testable consequences from scientific theories, so it should not be a matter of surprise or regret that it is the basis of all correct design and implementation in engineering practice.

As a trivial example, consider the specification

$$x' > x \ \wedge \ y' = y \qquad \ldots S$$

which specifies that the value of x is to be increased, and the value of y is to remain the same. No restriction is placed on changes to any other variable. There are many programs that satisfy this specification, including the assignment

$$x, y := x + 1, y \qquad \ldots P$$

Correctness of a program means that every possible observation of any run of the program will yield values which make the specification true; for example, the specification is satisfied by the observation $(x = 4 \wedge x' = 5 \wedge y' = y = 7)$, because the predicate S is true when its free variables are replaced by their observed values

$$5 > 4 \ \wedge \ 7 = 7$$

In fact, the specification is satisfied not just by this single observation but by every

possible observation of every possible run of the program

$$[(x, y := x + 1, y) \;\Rightarrow\; x' > x \;\wedge\; y' \;=\; y]$$

This mixture of programming with mathematical notations may seem unfamiliar; it is justified by the identification of each program with the predicate describing exactly its range of possible behaviours. Both programs and specifications are predicates over the same set of free variables, and that is why the concept of program correctness can be so simply explained as universally quantified logical implication between them.

Logical implication is equally interesting as a relation between two products or between two specifications. If S and T are specifications, $[S \Rightarrow T]$ means that T is a more general or abstract specification than S, and at least as easy to implement. Indeed, by transitivity of implication, any product that correctly implements S will serve as an implementation of T, though not necessarily the other way round. So a logically weaker specification is easier to implement, and the easiest of all is the predicate **true**, which can be implemented by anything.

Similarly, if P and Q are products, $[P \Rightarrow Q]$ means that P is a more specific or determinate product than Q, and it is (in general) more useful. Indeed, by transitivity of implication, any specification met by Q will be met by P, though not necessarily the other way round. So for any given purpose a logically weaker product is less likely to be any good, and the weakest product **true** is the most useless of all.

Explanation of correctness as implication gives a strangely simple treatment of the perplexing topic of non-determinism. Let P and Q be product descriptions with the same alphabet. Their disjunction $(P \vee Q)$ may behave like P or it may behave like Q, with no indication which it will be. In order to be sure that this is correct, both P and Q must be correct. Fortunately, this is also a sufficient condition; this is justified by appeal to the fundamental logical property of disjunction as the least upper bound of the implication ordering

$$[P \vee Q \Rightarrow S] \quad \textbf{iff} \quad [P \Rightarrow S] \text{ and } [Q \Rightarrow S]$$

The progress of a complex engineering project is often split into a number of design stages. The transition between successive stages is marked by signing off a document, produced in the earlier stage and used in the later stage. A design document D can also be regarded as a predicate: it describes directly or indirectly the general properties of all products conforming to the design. But before embarking on final implementation, it is advisable to ensure the correctness of the design by proving the implication

$$[D \;\Rightarrow\; S]$$

Now the implementation of the product itself reduces to the simpler task of finding a predicate P, expressed in technologically feasible notations, which satisfies the implication

$$[P \Rightarrow D]$$

Transitivity of implication then ensures the validity of the original goal that the product should meet the starting specification

$$[P \Rightarrow S]$$

This is a very simple justification of the widespread engineering practice of stepwise design. It is also a vindication of our philosophy of interpreting specifications, designs and implementations all as predicates describing the same kind of observable phenomena. However, these predicates are usually expressed in very different notations at each different stage of the engineering process. For example, the notations of a programming language are deliberately restricted in the interests of feasibility and efficiency of implementation. Satisfaction of notational constraints is an essential feature also for the solution of any mathematically defined problem: the answer must be expressed in notations essentially more primitive than the problem, for example as numerals rather than formulae, as explicit functions rather than differential equations. Otherwise a trivial restatement of the problem itself could be offered as a solution.

Stepwise design is even more effective if it is accompanied by decomposition of complex tasks into simpler subtasks. Let D and E be designs of components that will be assembled to meet specification S. The correctness of the designs can be checked before their implementation by proof of the implication

$$[D \wedge E \Rightarrow S]$$

The two designs can then be separately implemented as products P and Q that conform individually to the two design descriptions

$$[P \Rightarrow D] \quad \text{and} \quad [Q \Rightarrow E]$$

Their assembly will then necessarily satisfy the original specification

$$[P \wedge Q \Rightarrow S]$$

The correctness of the final step does not depend on lengthy integration testing after assembly of the components, but rather on a mathematical proof completed before starting to implement the components. The validity of the method of stepwise decomposition follows from a fundamental property of conjunction: that it is monotonic in the implication ordering.

The principle of monotonicity plays an essential role in engineering practice. Let X stand for a component, and let Y stand for a component that is claimed to be better than X in all relevant respects: as explained above, this can be expressed formally as an implication between their descriptions

$$[Y \Rightarrow X]$$

Now let $F(X)$ be a description of the behaviour of an assembly in which X has been inserted as a component. Replacement of component X by Y in the physical assembly corresponds to replacement of the description of X by the description of Y, and the resulting overall description is therefore $F(Y)$. If Y is really better than X, the engineer would expect the resulting assembly to be better too. This expectation is expressed by the implication

$$[Y \Rightarrow X] \Rightarrow [F(Y) \Rightarrow F(X)]$$

But this is exactly what is meant by the statement that F is monotonic.

For any theory to be useful in engineering, all methods of connection of components into an assembly should be monotonic in this sense. Of course, in practice, the principle will occasionally be violated, in cases when a supposed local improvement leads to worse global performance. This kind of failure is one of the most worrying problems to the engineer, because it is not just a failure of a single component or a single product, but rather a failure in the underlying theory on which its whole design has been based. Until the theory has been mended, there is no reason to suppose that the design can be.

A major problem in the account we have given of stepwise decomposition is that the designer must simultaneously formalise the designs of *both* the components D and E. That is like trying to split an integer s into two integer factors before the invention of division: it was necessary then to guess *both* the factors d and e, checking the guesses by multiplication

$$d \times e = s$$

But after the invention of long division, all that is needed is to guess only one of the numbers, say d; the other can be calculated by the formula

$$e = s \div d$$

Provided there is no remainder (which is also checked by calculation), this is guaranteed to give the other factor. Fortunately, a similar principle is in general available in engineering design.

The principle is especially helpful in planning the reuse of existing assemblies and designs. Suppose it is decided to use a known design or available component Q in the implementation of a specification S. So it remains only to design another

component X which will be connected to Q, adapting its behaviour to meet this particular requirement. More formally, X must satisfy the implication

$$[X \wedge Q \Rightarrow S]$$

There are many answers to such an inequation, of which (X = **false**) is the most trivial. It is also the most difficult to implement — in fact impossible! There can never be any way of expressing the universally false predicate in any notation that claims to be implementable. What we want is at the other extreme, the answer that is easiest to implement, the one which preserves all design options and choices, so that these can be made later, when it is possible to assess their implications.

That is why we ask: What is the *weakest* specification [92] that should be met by the designers of X? In a top-down design, it is much better to calculate X from Q and S, rather than attempting to find it by guesswork. Fortunately, propositional calculus gives a very simple answer

$$X = \neg Q \vee S$$

This is guaranteed by the law of propositional logic

$$[X \wedge Q \Rightarrow S] \quad \textbf{iff} \quad [X \Rightarrow (\neg Q \vee S)]$$

The specification $(\neg Q \vee S)$ is often written as an implication $(Q \Rightarrow S)$, and will be in general easier to implement than S. The formula permits calculation rather than guesswork to aid in the top-down search for an implementation of X that works with Q to achieve S. Such replacement of guesswork by calculation is the main practical goal of the development of a mathematical theory for engineering.

1.6 Abstraction

Our simple explanation of correctness assumes that the alphabets of specification, design and implementation are all the same. In many cases, the alphabets are different, and for good reason: they reflect different levels of abstraction, granularity and scale at which the observations are made. The whole task of design and implementation is to cross these levels of abstraction, and to do so without introducing error. A simple case of abstraction is when the alphabet of the specification is a subset of that of the implementation. For example, specifications will usually exclude mention of any variable introduced to describe only the internal interactions of the components of the implementation. Such a variable serves as a *local variable* in a program, similar to a bound variable in a mathematical formula. What is observable inside the assembly is of no concern to its customer, and is usually hidden by physical enclosure in its casing. The corresponding logical operation must remove the free variable from the predicate and from its alphabet.

For the implementation to work, such a hidden variable describing an internal property must indeed have *some* value, but we do not care what it is. It should therefore be hidden by existential quantification. The quantification is justified by the $\exists$-introduction rule of the predicate calculus

$$[P \Rightarrow S] \quad \textbf{iff} \quad [(\exists v \bullet P) \Rightarrow S], \qquad \text{if } v \text{ does not occur in } S$$

The variable is removed from the alphabet of $(\exists v \bullet P)$. Existential quantification over internal interactions of an assembly often leads to considerable simplification of the descriptions, without affecting the range of specifications that will be satisfied.

Universal quantification plays a complementary role to the existential: it assists in the top-down design of systems with reusable components. Recall the task described at the end of the previous section, to find the weakest X such that

$$[X \wedge Q \Rightarrow S]$$

But of course the condition of implementability of the conjunction $(X \wedge Q)$ requires that X must not mention any of the output variables (say x' and y') of Q. So the answer must hold for *all* values which Q may give to them, as in the universally quantified formula

$$X \quad = \quad (\forall x', y' \bullet Q \Rightarrow S)$$

where $out\alpha Q = \{x', y'\}$. This answer has been called the *residual* [15] of S by Q, because it describes what remains to be implemented in order to achieve S with the aid of Q. The answer is justified again by a simple law of the predicate calculus

$$[X \wedge Q \Rightarrow S] \quad \textbf{iff} \quad [X \Rightarrow (\forall x', y' \bullet Q \Rightarrow S)]$$

whenever x', y' do not occur in X.

As an example, suppose it is required to maintain a constant value for the expression $(x - y)$. This task is expressed in the specification that its value afterwards is the same as its value before

$$S \quad = \quad (x' - y' = x - y)$$

Suppose for other reasons it is desirable to double the value y by

$$Q \quad = \quad (y := 2 \times y)$$

What simultaneous change must be made to the value of x in order to re-establish the truth of S? The required answer is given by the residual.

$$
\begin{aligned}
& (\forall y' \bullet y' = 2 \times y \Rightarrow (x' - y' = x - y)) \\
= \quad & (x' - 2 \times y \;=\; x - y) \\
= \quad & x' \;=\; x + y \\
= \quad & x := x + y
\end{aligned}
$$

The answer, which is not totally obvious, has been derived by pure calculation, and has not required the dubious aid of intuition.

Of course, not all design decisions are sensible. An ill-conceived design may be expensive, difficult or even impossible to implement. Suppose the specification is to make the product of x and y into an odd number with the help of a program that doubles y. So the required residual is

$$
\begin{aligned}
& (\forall y' \bullet y' = 2 \times y \;\Rightarrow\; (x' \times y' \text{ is odd})) \\
= \quad & x' \times 2 \times y \text{ is odd} \\
= \quad & \textbf{false}
\end{aligned}
$$

This is, of course, unimplementable: there is no way that an odd product can be obtained by doubling one of the factors, and the situation cannot be remedied by changing the value of the other factor. It was clearly foolish to think that it would help to double y. Fortunately, the calculation of the residual as **false** gives clear warning of the impossibility of implementation.

In general, the transitions between engineering specifications and designs, or between various levels of design, present conceptual gaps far greater than can be bridged by just hiding some of the details of the interaction between components of the implementation. In fact, there is often an abrupt change in the nature, scale and granularity of all the observations involved and therefore of the alphabet used to denote them. We have a hierarchy of abstraction levels analogous to that found in the branches of a mature science, and there is an even more urgent practical need to demonstrate the soundness of the transitions between them.

A simple but quite general way of solving the problem is to describe mathematically the relationship between observations at the two levels, for example specification and design. At the design level, let $D(c)$ be a predicate with alphabet $\{c\}$ denoting a concrete observation relevant to implementation, and let $S(a)$ be a specification with alphabet $\{a\}$ denoting a more abstract observation relevant to the customer's use of the product (our reasoning will apply equally to much larger alphabets). The problem cannot be solved by plain universally quantified implication $[D(c) \;\Rightarrow\; S(a)]$ is equivalent to $[\exists c.D(c) \;\Rightarrow\; \forall a.S(a)]$; except in the most trivial cases this is just false, and most certainly does not define the proper notion of correctness, relating the design to the specification. The solution is to understand and formalise the relationship between an individual observation c of the design, and the corresponding values of a to which it may give rise at the

higher level. This relationship can be described in the usual way by some linking predicate $L(c, a)$. Now the set of all abstract observations that may be made of the design is described by a predicate in which all concrete observations are hidden

$$\exists c \bullet D(c) \wedge L(c, a)$$

This construction lifts the abstraction level of the design to that of the specification. More precisely, it gives the strongest specification with alphabet $\{a\}$ that is satisfied by the design D. Proceeding from the top down, the specification describing the higher level observation a can be converted into a design describing the more concrete observations

$$(\forall a \bullet L(c, a) \Rightarrow S(a))$$

This actually gives the weakest design with alphabet $\{c\}$ that is guaranteed to satisfy the specification. Either the existential or the universal transformation may be used to define correctness, as shown by the equivalence

$$[(\exists c \bullet D(c) \wedge L(c, a)) \Rightarrow S(a)] \quad \textbf{iff} \quad [D(c) \Rightarrow (\forall a \bullet L(c, a) \Rightarrow S(a))]$$

Equivalences of this form are very common in constructing links between theories: the two transformations are often known as *Galois connections*. They will play a central role in the unification of theories of programming.

1.7 The ideal and the reality of engineering

The preceding sections have painted an ideal picture of an engineering project, as an abstract exercise in pure logic. It is a picture that is as far from the day-to-day reality of engineering practice as could be expected of any other branch of pure speculative philosophy. The first idealisation is that the true requirements and qualities of a product can be accurately captured in a precise logical description of the way that it should behave. Requirements capture is in fact the most challenging of all the engineers' tasks, because there is no way of checking that they describe what the customer actually is going to want when the product is actually delivered. Even the best specifications are peppered with qualifications like "reasonably" and "normally" and "approximately" and "preferably", which cannot be made more precise until much later in the investigation of the design, or even after delivery.

Another bold idealisation is that the specification, once formalised, will remain constant. In fact specifications are subject to a constant series of changes, before, during, and even after delivery of the product. In principle, the slightest change can invalidate the entire structure and all the details of the whole design, but in practice the engineer usually finds some ingenious way of preserving and

maintaining the greater part of the work progressed so far. Indeed, the experienced engineer often has a way of anticipating the most likely changes from the start. Changes made even after initial delivery of the product are particularly significant in software engineering. The lifelong task of many programmers is to make a succession of adaptations to some existing program, to meet new or changing needs that were never envisaged in the original specification. In principle, the rigorous documentation of the design should be the most valuable aid in identifying where to make the necessary changes and how to make them correctly. Unfortunately, in practice the design documentation goes rapidly out of date, and even becomes too dangerous to use. The only safe way to find out what the program actually does is by testing and tracing example runs, by trial and more frequently by error.

A project that starts from scratch can suffer from even greater problems. The absence of a previous program to adapt usually means that this is the first application of some new and comparatively immature technology to a problem for which there is no current solution. These are just those cases where the standard calculations do not apply, and greatest reliance must be placed on guesswork and experimentation. In the top-down progression from specification through design, the earliest decisions on the structure of the product are the most irreversible, and yet they must be taken at the time of greatest ignorance of their consequences on the cost and performance and acceptability of the product. Mistakes are inevitable, and can be recovered only by judicious backtracking. If success is possible at all, it is only by the experience, judgement and ingenuity of the engineer. No amount of mathematical calculation or logical proof can ever be a substitute for that.

Finally, all other problems of engineering design must be subordinate to the overriding imperative to deliver the promised product at the due time, and at a cost within the allocated budget. All the ideals of philosophy and logic are of no avail if the engineer fails in this, the most important of all engineering duties. And in fact, this is where the true engineer finds his or her greatest intellectual and personal reward – not just the pursuit of an ideal of accuracy, but also a justified pride in the working product and the satisfied customer.

The stark contrast between ideals and reality has been noted and deplored by philosophers, moralists and theologians through the centuries; there is no general reconciliation, and each individual must continually find a resolution appropriate to the needs of the moment. In engineering, some will ignore theoretical ideals, and rely exclusively on experience of their craft, but others will on occasion find guidance from their understanding and pursuit of an ideal, which is shared by other members of a recognised profession. The ideal suggests an integrated approach to the overall task, and enables deviations to be isolated and controlled separately. In the longer term, a theoretical understanding provides a basis for the emergence of professional standards, methods and techniques for uniformly reliable solution of technical problems, independent of the area of application. These in turn provide

material for a sound education for new entrants to the profession, who respond favourably to the inspiration of a unifying ideal, even before understanding the many compromises necessary to put it into practice.

Finally, the privilege of the purest allegiance to an ideal is that of the researcher, seeking to build a scientific foundation which will contribute simultaneously to the advancement of knowledge and education, as well as the continuous improvement of professional practice of the accredited engineer. One final appeal to an analogy with the physical sciences: it is the pursuit of an ideal of truth that in the long run has led to the development of modern technology and engineering methods, and these have been of outstanding success in solving problems which continue to face the modern world.