Mathematical Models for Computing Science

Lecture Notes for Marktoberdorf Summer School August 1994

C.A.R. Hoare

Mathematical equations and other predicates are used in the physical sciences to formalise, describe, and predict the observable behaviour of some isolatable fragment of the real world. Phenomena of interest in computing science can with advantage be formalised as mathematical predicates in the same scientific way. As a result, concurrency can often be modelled by conjunction [5], non-determinism by disjunction, locality by existential quantification, and correctness by logical implication [7]. This thesis is illustrated by application at a variety of levels of granularity, scale and abstraction: in hardware, by switching, combinational and sequential circuitry [9, 6]; and in programming [4] by the procedural [2], parallel [1], and logical [8] paradigms.

The major achievement of modern science is to demonstrate the links between phenomena at different levels of abstraction and generality, from quarks, particles, atoms and molecules right through to stars, galaxies, and (more conjecturally) the entire universe. On a less grand scale, the computer scientist has to establish such links in every implementation of higher level concepts in terms of lower [10, 11]. Such links are also formalised as equations or more general predicates, describing the relationships between observations made at different levels of abstraction [3]. Their clarification assists in understanding the structure of an entire scientific discipline.

could I have a copy of my
Marktoberdorf lecture notes please.

Contents

Introduction

A scientific theory is a predicate, usually expressed in the notations of mathematics, which describes all possible observations that can be made directly or indirectly of any system from a given reproducible class. 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 with the physical reality which they refer to.

The same physical system may be described at many different levels of abstraction and granularity, for example as a collection of interacting quarks, or elementary particles, or atoms, or molecules, or crystal structures. 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 soundly based on the more detailed theory below it. This is the strongest argument for the soundness not only for each separate theory, but also for 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 levels of abstraction.

An engineering product is usually an assembly of components, which in general operate concurrently. The operation of each component can be described scientifically by a separate predicate. Their joint behaviour in the assembly can often be described by the conjunction of these predicates. A non-deterministic product is 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. Propositional and predicate logic provide all the basic concepts needed for a systematic engineering design methodology.

These lectures will illustrate the methodology by examples drawn from many branches of computing science. The first is a simple control system, a water tank that must not be too empty or too full. This application must be implemented partly with the aid of an electronic system built ultimately from transistors. The behaviour of C-mos transistors is described at a fine level of granularity by Boolean switching logic. At the gate level, combinational logic provides simpler designs and methods of reasoning about them. The introduction of storage elements (latches) requires a shift to a more complex sequential circuit theory. Formalisation of links between these theories is needed to show how components assembled at a coarser level of granularity can be implemented as sub-assemblies at the finer level. Such transitions between levels of abstraction are as important to reliable engineering as they are to the progress of science.

Software also provides a wide range of programming paradigms and their corresponding theories. For conventional sequential programs, I take the language and theory of E.W. Dijkstra's Discipline of Programming. A more complex theory is needed for parallel programs operating in shared store; these complexities are avoided in the declarative programming paradigms, both functional and logical, as well as in simple relational databases. Distributed computing is represented by Communicating Sequential Processes. Each of these theories has it own concepts and methods of reasoning, which may be happily studied in isolation. The links between the theories are also essential to the validity of systems assembled from programs written in different languages, and for avoidance of engineering errors of the worst kind, those which lurk in the interfaces between different materials and technologies.

Each of these topics will be treated only in the simplest possible fashion, ignoring many known problems and practical difficulties. This too accords with scientific principle, and certainly with educational practice. The world becomes comprehensible only by isolating a few significant features; disturbing factors are initially regarded secondary effects, controllable by careful design of experiment. And even in applying the theory, an engineer will often prefer to develop a sketch of a new design in a simple but inaccurate theory: experience gives confidence that errors revealed by a later more complex analysis can be avoided by later adjustments in the details of the implementation. The embedding of simple theories in more complex but realistic ones can help to make this transition reliably.

The simple and undeveloped theories expounded in these lectures are very far from practical application. The primary reason for their study can only be scientific curiosity, to answer questions on "How does it work, and why?". Experiments in computing are notoriously easy and cheap to conduct, by program execution or by hardware simulation; so there is little joy in predicting their individual results. But what the theory gives is prediction and control of general properties, not only of a particular program but of a general class of programs. Of greatest interest is the way in which the separate theories are linked, because this clarifies the inherent structure of computing science as an intellectual discipline. It forms the basis of a coherent general education in the subject and for subsequent mutual understanding and cooperation of specialist engineers and research scientists.

1 Observations and Alphabets

The first task of the scientist is to isolate some interesting class of system for detailed study. Then 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 variable 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 variables 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 variables is known as an *alphabet*; and the names will occur as free variables, together with physical constants, in any predicate describing the general properties of the system.

An *observation* of the system can be expressed as a set of equations, ascribing particular constant values to each of the variables in the alphabet, for example

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

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".

1.1 Water tank control system

Most of the dynamic variables treated in the physical sciences are assumed to be continuously varying with time over a continuum of possible values. Consider a simple tank containing a liquid (Fig. 1). The variables selected to describe the state of the system are as follows:

t	(measured in seconds) is 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.

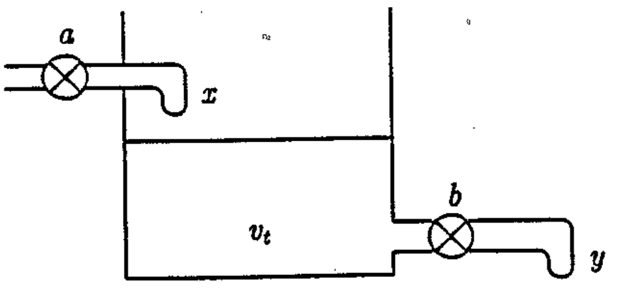


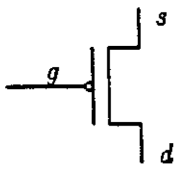
Fig: 1

1.2 Transistor networks

The ultimate components of most digital electronic circuits are transistors and wires. The transistors act as switches to connect (and disconnect) the wires, either to a source of power (High voltage), or to ground (Low voltage). Observations are assumed to be made only when the whole network is *stable*, in the sense that all switching activity has ceased, and all connections are permanently established. For simplicity, we will postpone consideration of circuits that never stabilise.

In design and documentation, each wire is given a name, say v, w, g, s, d. For each such name w, let Hw be a Boolean variable which is observed to be true when wire w is stably connected to High voltage, and false when it is disconnected from High voltage. Similarly, let Lw mean that wire w is connected to Low voltage. For example, particular observations of the stable values of the wires connected to the three terminals of a P-transistor might be

$$\begin{array}{ll} & Lg \wedge Hs \wedge Hd \\ \text{or} & Hg \wedge Hs \wedge Ld, \end{array}$$



where g is the wire connected to the gate
s is connected to the source
and d is connected to the drain.

The association of two Boolean variables to each wire permits representation of four possible states. The usually desirable states are those in which exactly one of Hw and Lw is true. The state in which both are false is called *floating* or *tristate*. In the fourth state, known as a short-circuit, the wire connects the sources of low and high voltage, sometimes in an oscillatory manner. This state must be avoided by the designer.

1.3 Combinational logic

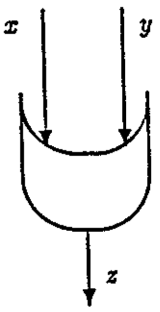
Avoidance of the floating and short-circuit states can be achieved by certain design conventions that will be detailed later. As a result, the design of combinational logic may use the wire name itself as a Boolean variable to distinguish the remaining two states.

w	means the wire is connected to High (and not to Low)
$\neg w$	means the wire is connected to Low (and not to High).

For example, if x and y are connected to the inputs of an OR-gate and z to its output, typical observations might be

$$z \wedge y \wedge \neg x$$

$$\neg z \wedge \neg y \wedge \neg x$$



1.4 Sequential circuits

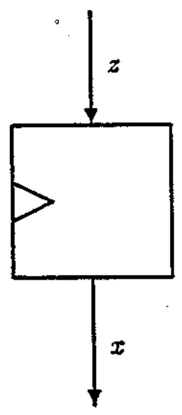
The previous examples treat just a single observation of each wire, made at the end of a cycle of operation, when the whole circuit has reached a stable state. Sequential circuit design deals with the whole sequence of successive cycles, so subscripted variables are needed to record the observations. Let v_t stand for the value of wire v recorded at the end of the t^{th} cycle of operation. Here, t ranges over the natural numbers

$$\mathcal{N} = \{0, 1, 2, \ldots\}.$$

A particular observation is recorded as an infinite sequence of Boolean values. For example, if z is the input and x is the output of a delay element, we might observe:

$$x = < 0,1,1,0,1,0,\ldots >$$

$$\wedge\ z = < 1,1,0,1,0,\ldots >$$



1.5 Sequential programs

The effect of executing a sequential program is to change the values of the global variables of the program. These can be observed before the program starts and after it terminates: the intermediate values are of little or no concern. Consider a global program variable with symbolic name x; we use a dashed variable x' to denote its final value, and the undecorated name x itself to denote the initial value. Obviously the type and range of values of these observational variables are the same as those declared in the program for the global variable with the same name. It is convenient to exclude from the alphabet of a program the dashed variant of the variables to which it makes no assignment. So a particular observation of the program $x := x + y$ might be $x = 5 \wedge x' = 7 \wedge y = 2$.

1.6 Parallel programs

In a computer with multiple processors, it is possible for two programs to share the same main store, and to update stored values in an almost arbitrarily interleaved fashion. A similar complexity is introduced by single-processor machines which timeshare a number of program threads with the aid of interrupts. Observations of initial and final values of the variables are no longer adequate to control the possibilities of interaction (even interference) between the programs. So the entire history or trajectory of the entire program from its start to its finish must be recorded as a set of timed observations of the current machine state. The observation variables are

start:	the start time
finish:	the finish time.

For each program variable, its name denotes a function from a time in the interval [start ... finish] onto the value of the variable at that time. An observation of a program that assigns to x and y in parallel could be

$$\begin{array}{rlcl} & \text{start} = 37 \ \& \ \text{finish} = 43 & & \\ \& \ \forall t. & 37 \leq t < 42 & \Rightarrow & y_t = 12 \\ & \& \ 42 \leq t \leq 43 & \Rightarrow & y_t = 20 \\ & \& \ 37 \leq t < 40 & \Rightarrow & x_t = 4 \\ & \& \ 40 \leq t \leq 43 & \Rightarrow & x_t = 16 \end{array}$$

Here the value of x has changed at time 40, and the value of y at time 42. Each value has changed only once.

1.7 Communicating processes

A communicating process is intended to interact with its environment at certain distinct points in time. Each individual interaction can be recorded as a value from a certain set A of event names (often called the alphabet of the process). An observation of the behaviour of the process up to a given moment of time can be recorded as the sequence of events in which it has engaged so far. This is known as a *trace*, usually abbreviated to *tr*; its type is A^*, the set of all finite sequences of events from A. At this level of detail, the exact timing of the occurrence of an event is ignored, and only their relative ordering is significant. Events that occur simultaneously would have to be recorded in some arbitrary order.

As an example, consider a device which alternately engages in the two events a and b. Perhaps it is a simple single buffering device, which alternately accepts a message and delivers it at the other end. For simplicity, we will ignore the content of the message. An observation of the behaviour of this device may occur after three events, when

$$\mathit{trace} \ = \ < a, b, a > .$$

Some time later (when the second message has been delivered), a subsequent observation may be

$$\mathit{trace} \ = \ < a, b, a, b > .$$

1.8 Functional programs

The alphabet of a functional program consists of the names of all the functions it declares and uses, together with their types, for example

$$bcd : \mathcal{N} \times \mathcal{N} \rightarrow \mathcal{N}.$$

Each execution of the program begins with input of a call to one of these functions, with a particular set of constant parameters. Each successful execution ends with output of a constant value computed by the program. Observation of all executions may be recorded in the usual way as an equation, ascribing a particular value to the variable

$$bcd = \text{ the greatest common divisor.}$$

Observation of the entire value of the function *bcd* would require an infinite set of runs, and take an infinite text to record its graph. But physical science long ago abandoned its fear of the infinite, when it took real numbers as its medium of measurement; and there is no need for computing science to hesitate from the same step.

1.9 Logic programs

In the declarative view of logic programming, a predicate is just a function that delivers a Boolean value. Like a functional program, its alphabet is the set of predicate names that it uses and defines, for example,

father, mother, grandfather.

Typical observations might include

father (henry, elizabeth)
¬mother (henry, elizabeth).

1.10 Databases

The alphabet of a relational database consists of the set of its attribute names (and their types), for example

title, theatre, author.

Its observations are just the records that can be retrieved from it, for example:

$$\text{title} = \text{Camelot} \wedge \text{theatre} = \text{Coliseum} \wedge \text{author} = \text{Lerner}.$$

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 is to replace a merely historical record of observations 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. A predicate $P(x, y, \ldots, z)$ correctly describes a particular observation, e.g.

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

if substitution of each variable by its observed value makes the predicate true (*satisfies* it):

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

The predicate correctly describes a particular system or subclass of system 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 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 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 have to have no observations, and therefore be inaccessible to science). All of science is concerned with predicates that lie strictly between the two extremes of truth and falsity.

In science and engineering, it is normal practice to reason, 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 strongly prefer 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

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

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 from an alphabet whose existence, composition and meaning can only be explained informally by relating them to reality. In fact we will begin 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.

2.1 The water tank

The behaviour of a physical process is usually described by a physical law, often in the form of a conservation principle or a differential equation. In the example of the water tank, the obvious law of conservation of liquid may be expressed

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

where ϵ is the maximum rate of accumulation of errors due to seepage, evaporation, precipitation, condensation, etc.

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

$$0 \leq a_t \leq amax, \qquad 0 \leq b_t \leq bmax.$$

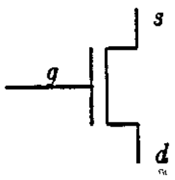
Finally, the relationship between the valve settings and the flow of liquid may be expressed by differential equations, say

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

where δ accounts for extra outflow due to water pressure. 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 setting and the other variables and constants in the system.

2.2 Cmos transistors

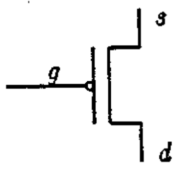
An N-transistor (implemented in C-mos) is drawn



where g is called the *gate*, and s and d are *source* and *drain* respectively. It acts as a simple switch, connecting the source to the drain when the gate is connected to High voltage. Thus if either of s or d is already connected to Low, the other will be too. This aspect of the transistor's behaviour is described by the predicate

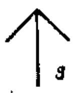
$$Hg \Rightarrow (Ls = Ld),$$

which holds in all the stable states of the three wires g, s and d, connected to the transistor. A connection through an N-transistor will also transmit High voltage, but unfortunately only with a degree of attenuation that makes its use unreliable in controlling other switches. For simplicity, we assume non-transmission of High voltage, even at the risk of ignoring the possibility that it leads to short circuits. A P-transistor is *complementary* to the N-transistor, in the sense that it makes its connection just when the gate is low, and then it will reliably conduct High voltage:



$$Lg \Rightarrow (Hs = Hd).$$

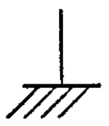
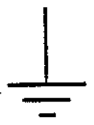
Two essential components of any switching network are the power rail (VDD) and the ground plane (GND). Any wire can be connected to either of these sources, to neither of them, or (erroneously) to both. Connection of wire s to power is indicated pictorially by



Its effect is to make Hs true, which is described by the simple assertion

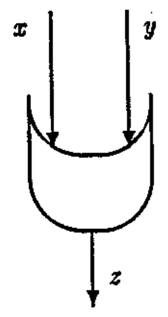
$$Hs.$$

Connection to ground is indicated

Ld  or 

2.3 Combinational logic

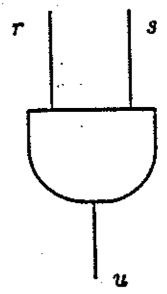
An OR-gate is usually drawn



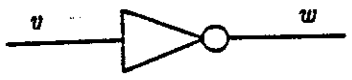
Its purpose is to present at wire z the higher of the two voltages at the wires x and y. This is described by the equation

$$z = x \vee y.$$

Similar definitions can be given for AND-gates and NOT-gates.



$$u = r \wedge s$$



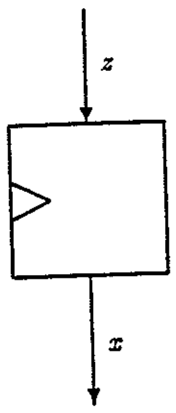
$w = \neg v$

2.4 Sequential circuits

Every combinational component is also a sequential circuit, whose behaviour is obtained by just subscripting all the wire names by time. For example, the behaviour of the OR-gate is described as a sequential circuit by the predicate

$$z_t = x_t + y_t, \qquad \text{for all } t \in \mathcal{N}.$$

The simplest component that distinguishes the sequential circuit from the combinational is the Delay, whose output on each cycle of operation is equal to its input at the end of the previous cycle.

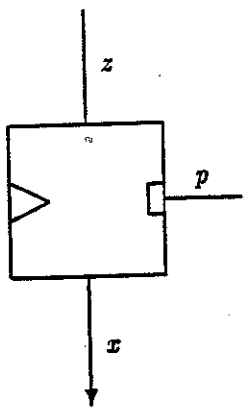


This behaviour is described by the predicate

$$x_{t+1} = z_t, \qquad \text{for all } t \in \mathcal{N}.$$

This enables the value of x to be predicted from the value of z at all times except $t = 0$, when it is in fact inherently unpredictable.

Another component of a sequential circuit provides useful storage. It is the D-type flip-flop, and has two input wires (z and p) and one output (x). The output remains constant on successive cycles, unless a control signal is received on p. In this case, the data value presented on the other input wire (z) is stored for output on the following cycle.



$$x_{t+1} = (z_t \text{ if } p_t, \text{ otherwise } x_t), \text{ for all } t \in \mathcal{N}.$$

In future, this will be written using a shorter conditional notation

$$x_{t+1} = (z_t \lhd p_t \rhd x_t).$$

2.5 Sequential programs

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:

$$x' = x + y.$$

No restriction is placed on the initial or final values of any other global variables. Since no assignment is made to y, y' is not even in the alphabet of the process.

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

$$x' = x + 3 \ \&\ y' = y - x.$$

In future we will *identify* the notations of a programming language with the predicates that describe exactly the behaviour of the program when executed.

The interpretation of programs as predicates allows a simple definition of the conditional construction, using only propositional logic. Let programs P and Q have the same alphabet, and let c be a Boolean expression written in the language (so it has only undashed variables, and can describe only the initial state). We then define

$$\text{if } b \text{ then } P \text{ else } Q = (P \lhd b \rhd Q).$$

If b is true (initially of course), all the observations of its execution are observations of P; otherwise they are observations of Q.

An even simpler definition can be given of the vexed concept of non-determinism. Let P and Q be predicates with the same alphabet. Then their disjunction $(P \vee Q)$ describes a system that may behave like P or like Q; so all its observations are described either by P or by Q or by both. It is impossible to know in advance which of them it will be — so the user of a non-deterministic product had better not care!

2.6 Parallel programs

To control the complexity of parallel programming in a multiprocessor with shared storage, let us assume that any variable which is the target of assignment in one program will not be assigned in any other concurrent program. Consider the single assignment statement $x := x + y$, with output variable x and input variables x and y. As a result of its execution, there is a time t' between its start and finish when the value of x is equal to the sum of the values of x and y, read at some earlier time t. At all times other than t', the value of x remains constant; no such assurance can be given for y, because its value can be assigned by some other concurrently executing process. A formal definition is:

$$\begin{aligned} x := x + y = \exists\, t, t' \mid \text{ start } &\le t \le t' \le \text{ finish} : \\ x_{t'} \;=\; & x_{\text{start}} + y_t \\ \&\ \text{changes}\ (x) &\le 1 \end{aligned}$$

where changes (x) is the number of changes in the value of x in the interval between start and finish.

2.7 Communicating processes

The simplest communicating process is one that never interacts at all with its environment. It is defined by the predicate describing its (lack of) behaviour:

$$\text{STOP} = (\mathit{trace} \;= <>)$$

where <> is the empty sequence. A slightly more interesting process is one which first engages in an event a from its alphabet and then stops

$$a \rightarrow \mathrm{STOP} = (trace \ \leq \ < a >)$$

where $< a >$	is the sequence containing only the event a
and $x \leq y$	means that y begins with (or consists of) a copy of x. This is known as *prefix* ordering.

Even more interesting is a process P that first does a and then b, and then starts again from the beginning. Such a process satisfies the equation

$$P = (a \rightarrow (b \rightarrow P)).$$

Its behaviour will always be a finite subsequence of the infinite alternating sequence

$$trace \ \leq \ abababa \ldots \ \ldots .$$

An equivalent finite description of this behaviour is obtained by counting the number of past occurrences of a and b at a given time:

$$0 \leq \ trace \ . \ a - \ trace \ . \ b \leq 1$$

where $trace \ . \ x$ is defined as the number of occurrences of the event x in the trace.

In the example above, the equation defining P has P on the right-hand side as well. Such recursive definitions are very useful in computing; and explanation of their exact meaning is one of the reasons for the study of computing theory.

2.8 Functional programming

The primitive component of a functional program is a (possibly conditional) equation, for example

$$bcd(x, y) \quad = \quad bcd(x, y - x), \qquad \text{if } y > x.$$

This is already an exact description of the behaviour of the program. It states that if the second of the parameters to *bcd* is greater than the first, the result will be the same as if the first parameter had been subtracted from the second. More formally, this behaviour is described by a predicate in which all free variables are quantified over the appropriate range

$$\forall x, y \mid y > x : bcd(x, y) \quad = \quad bcd(x, y - x).$$

2.9 Declarative logic programming

The simplest component of a logic program is a clause consisting solely of a head, for example father (henry, elizabeth):-

This states that father is a Boolean function, which, when applied to constants henry and elizabeth, gives the answer *true*. This is exactly what is stated by the predicate:

father (henry, elizabeth).

A more interesting clause in a PROLOG program is one that contains a body as well as a head:

grandfather (X, Z):- father (X, Y), father (Y, Z).

The corresponding predicate is obtained by direct notational substitution:

1. replace :- by $\Leftarrow$ (logical implication)
2. replace , by $\wedge$ (conjunction)
3. universally quantify over all free variables.

The example shown above becomes

$$\forall X, Y, Z \mid: \text{father } (X, Y) \wedge \text{ father } (Y, Z) \Rightarrow \text{ grandfather } (X, Z).$$

By the laws of predicate logic, this is equivalent to

$$\forall X, Z \mid: (\exists Y :: \text{ father } (X, Y) \wedge \text{ father } (Y, Z)) \Rightarrow \text{ grandfather } (X, Z).$$

2.10 Databases

A database is a predicate defined by complete enumeration of all its observations, for example

$$\begin{array}{ll} & (\text{title} = \text{Camelot} \wedge \text{theatre} = \text{Coliseum}) \\ \vee & (\text{title} = \text{Cats} \wedge \text{theatre} = \text{Haymarket}) \\ \vee & (\text{title} = \text{Hair} \wedge \text{theatre} = \text{Coliseum}). \end{array}$$

A database query is also a predicate, using a much more expressive language to select the desired observations from the database. An example query might be

$$\text{theatre} \in \{\text{Haymarket, Coliseum}\} \ \& \ \text{title} \neq \text{Hair}.$$

3 Concurrency and Conjunction

Propositional logic provides many ways of constructing complex predicates from simpler ones; for us, the most important is undoubtedly *conjunction*, which we will write as &, and pronounce as *and*. If it needs definition, the following will suffice:

1. An observation satisfies a conjunction $(P \;\&\; Q)$ if and only if it satisfies both P and Q.

2. The alphabet of $(P \;\&\; Q)$ 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. We will consider first the simple but important case, when the alphabets of P and Q are disjoint, containing no variable in common. Then their conjunction $(P \;\&\; Q)$ describes the behaviour of two completely separate components, the first of which is described by P and the second by Q. There is no connection between the components, and no synchronisation or coordination of their behaviour. Each observation of their joint behaviour can be split in two: one part involves only variables from the alphabet of P, and this part satisfies P; the rest of the observation similarly satisfies Q. That is exactly the condition under which the whole observation satisfies $(P \;\&\; Q)$.

In most cases of interest, components of an engineering product are assembled 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, $x = 7 \;\&\; y = 12$) of one component P to be coupled with an observation (say $y = 12 \;\&\; z = 3$) 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) $x = 7 \;\&\; y = 12 \;\&\; z = 3$. Such a coupled observation can still be split into two overlapping parts, one of which satisfies P and the other Q. That is the exact condition for the whole observation to satisfy the conjunction $(P \;\&\; Q)$ of the component descriptions; its alphabet is clearly still the union of their separate alphabets.

This is the general method of modelling connection and interaction in an assembly constructed from two or more components. But in practice, of course, we are mostly interested in connections that can physically be realised in some available technology, for example by connection of wires in hardware, or by juxtaposition 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 contradictory specifications could yield the predicate *false*, which is a logical impossibility and could never be implemented in practice.

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 *outputs*, and form the *output alphabet* $\text{out}\alpha P$ of the subsystem P. Each variable of a complete subsystem can be controlled by only one of its components. So the conjunction $(P \ \& \ Q)$ is forbidden unless the output alphabets are disjoint:

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

A controlled variable in any subsystem is controlled in the whole system

$$\text{out}\alpha(P \ \& \ Q) = \text{out}\alpha P \cup \text{out}\alpha Q.$$

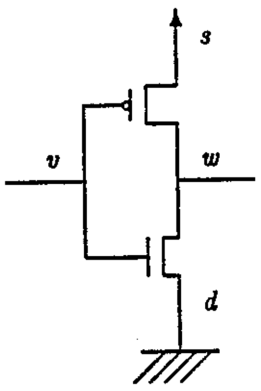
The input alphabet is just defined as the rest of the variables that are not outputs

$$\text{in}\alpha P = \alpha P - \text{out}\alpha P.$$

3.1 Transistor networks

A complex electronic network is constructed from simpler components by connection of wires. By general convention, wires that have been given the same name in two separate components of an assembly are joined together to form a single wire with the same name. Normal electrical conduction will ensure that both components see the same voltage on the wire at all times, at least when the circuit is stable and therefore observable. It is this that corresponds in physical reality to the logical principle that the variable denoting the wire can have only a single value.

By the definition of stability, a circuit is stable only when all its components are. As a result, the conjunction of predicates describing exactly the stable states of the components will describe exactly the stable states of the connected assembly. There is no need in a switching model to make any distinction between input and output wires. Consider, for example, an assembly made from two transistors, a P-transistor and an N-transistor, connected as shown



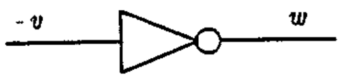
The conjunction of the four component descriptions is

$$\begin{array}{l} \ \ Hs \\ \&(Lv \Rightarrow (Hs \equiv Hw)) \\ \&(Hv \Rightarrow (Lw \equiv Ld)) \\ \&Ld. \end{array}$$

This can be rewritten as

$$(Lv \Rightarrow Hw) \ \& \ (Hv \Rightarrow Lw) \ \& \ Hs\&Ld$$

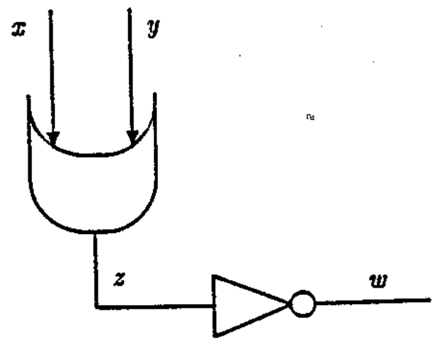
which describes with reasonable accuracy the actual and intended behaviour of a *negation* circuit with input v and output w. As a combinational circuit it is often abbreviated to



3.2 Combinational logic

The wires in a combinational circuit have the same effect of equalising values of variables in the alphabets of the components which they connect. But in this case, distinction between output and input wires is essential to avoid the kind of inconsistency that will lead to falsity of the describing predicate, and to short-circuit in the implemented product.

Consider the assembly of the three components introduced earlier:



This is described by

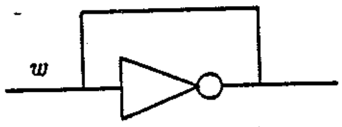
$$(z = x \vee y) \ \& \ (w = \neg z),$$

which may be rewritten as

$$(w = (\neg x \wedge \neg y)) \ \& \ (z = \neg w).$$

Here, the second conjunct describes the value of z, which may be of no subsequent interest in the use of the circuit. A wire whose only purpose is to carry a signal locally within a subassembly can be hidden (as described in the next lecture), with resulting simplification of the entire behavioural description.

The assembly of combinational circuits is subject to the constraint that the output wire of each circuit can be connected only to the input of some other gate, and that a chain of gates connected in this way must never form a cycle. The simplest circuit that violates this condition is

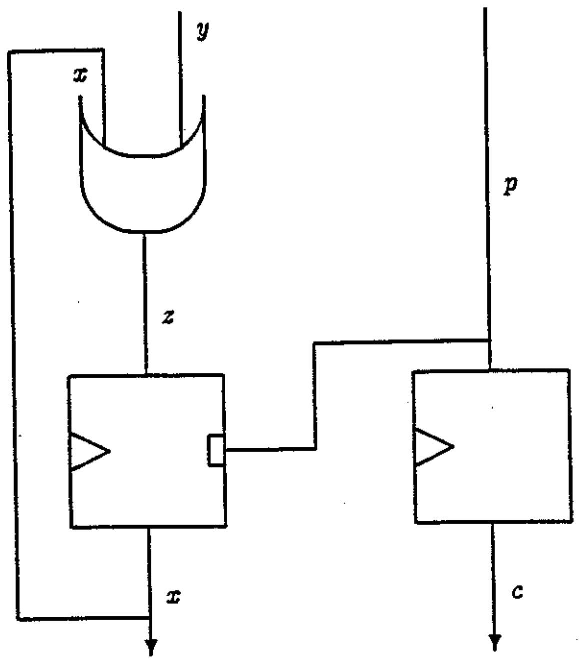


$$w = \neg w$$

The predicate describing the circuit is a contradiction, the same as *false*. In practice the voltage on w might oscillate around some intermediate value, consuming considerable power. These phenomena simply cannot be described in a theory with such a small and simple alphabet. That is why they must be excluded by syntactically checkable constraints on the predicates that are conjoined.

3.3 Sequential circuits

The connection of sequential circuits is subject to a similar restriction to that on combinational circuits; but a cycle of connected gates is allowed on condition that it contains a delay. For example, here is a circuit that combines three components that have been encountered earlier.



Its behaviour is described as the conjunction

$$(z_t = x_t \vee y_t)\&(x_{t+1} = z_t \lhd p_t \rhd x_t)\&(c_{t+1} = p_t).$$

Again, if z is never going to be used outside the assembly, this can be simplified to

$$(x_{t+1} = (x_t \vee y_t) \lhd p_t \rhd x_t)\&(c_{t+1} = p_t).$$

3.4 Sequential programs

Two sequential programs can be executed safely in parallel, provided that neither of them updates any global variables used by the other. Parallel execution then has exactly the same effect as sequential execution in either order. This effect is most simply described as the conjunction of the separate effects of each component. Using $\|$ to denote parallel execution of two assignments

$$x := x + z \parallel y := y - z$$

is precisely described by the conjunction of their separate behaviours:

$$(x' = x + z) \ \& \ (y' = y - z).$$

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

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

The output alphabet of a sequential program consists of all its dashed variables, i.e., those that appear to the left of an assignment within it. The restriction on sharing the output variables is sufficient to ensure consistency of the conjunction. We can therefore relax the usual constraint against one component using variables updated by another parallel component. For example, we can allow

$$(x := x - y \parallel y := 2 \times y) = (x' = x - y) \ \& \ (y' = 2 \times y).$$

When one of the components 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. An implementation might have to make a private copy of such variables before executing the programs in parallel. 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 with disjoint stores).

3.5 Communicating processes

Consider two completely separate processes with event alphabets A and B, containing no events in common ($A \cap B = \{\}$). A trace of their parallel execution is just an arbitrary interleaving of events from the traces of the individual processes. From this interleaved trace, omit all the events in B; the resulting trace (containing just the A-events) is denoted by $\mathit{trace} \upharpoonright A$. This will be a trace of the behaviour of the first process; and similarly $\mathit{trace} \upharpoonright B$ will be a trace of the second process. So if $P(\mathit{trace} \upharpoonright A)$ and $Q(\mathit{trace} \upharpoonright B)$ are descriptions of the behaviours of the two component processes, a description of their joint behaviour is given just by a conjunction of these two predicates. The event alphabet of the combined process is the union ($A \cup B$) of the two alphabets of the components.

Surprisingly, the same reasoning still works when there are events common to the alphabets of both the processes. The occurrence of such an event requires simultaneous participation by both processes, and therefore appears at the appropriate position in the trace of both of them. For example, let $A = \{a, b\}$ and $C = \{b, c\}$. The following table shows how the two restrictions work:

$$\begin{array}{lcl} \mathit{trace} & = & a\ b\ a\ c\ b\ c\ a\ b \\ \mathit{trace} \upharpoonright A & = & a\ b\ a\ \ b\ \ a\ b \\ \mathit{trace} \upharpoonright C & = & \ b\ \ c\ b\ c\ \ b. \end{array}$$

The example shows a typical trace of the connection of two buffering processes with alphabets A and C

$$P = a \to b \to P$$
$$Q = b \to c \to Q.$$

Their assembly is described by the conjunction of predicates describing their separate behaviour:

$$\begin{array}{ll} & 0 \le \mathit{trace}.a - \mathit{trace}.b \le 1 \\ \& & 0 \le \mathit{trace}.b - \mathit{trace}.c \le 1. \end{array}$$

From this it follows that

$$0 \le \mathit{trace}.a - \mathit{trace}.c \le 2.$$

If the only purpose of the event b is to convey information between these two components, the description that does not mention *trace.b* will be more convenient. The necessary hiding of the event b will be described in the next lecture.

3.6 Functional programming

The meaning of a functional program is the conjunction of the predicates describing its individual clauses. For example, a function *bcd* may be defined by three clauses, which are true for all x, y, z:

$$\begin{array}{llcllcl} & bcd(x,y) & = & x & \Leftarrow & x = y \\ \& & bcd(x,y) & = & bcd(x,\ y-x) & \Leftarrow & y > x \\ \& & bcd(x,y) & = & bcd(x-y,\ y) & \Leftarrow & x < y. \end{array}$$

Each individual clause describes a particular property of *bcd*, which is shared by many functions. For example, the first clause is satisfied by *min* and *max*, whereas the second clause is satisfied by the remainder function. All three clauses are satisfied by the greatest common divisor, a partial function whose domain is confined to strictly positive integers.

Note that the three conditions on these three lines are disjoint, in the sense that at most one of them can be true at a time. This ensures that no attempt will be made to ascribe inconsistent values to $bcd(x, y)$. If consistency is not obviously (even mechanically) checkable, a more complex theory will be needed to explain the consequences of this error.

3.7 Logic programming

In declarative logic programming, as in functional programming, the individual clauses are put together with conjunction

$$\begin{aligned} &\forall X, Z \mid: \text{grandfather}\,(X, Z) \Leftarrow \exists\, Y \mid: \text{father}\,(X, Y)\ \&\ \text{father}\,(Y, Z) \\ \&\ &\forall X, Z \mid: \text{grandfather}\,(X, Z) \Leftarrow \exists\, Y \mid: \text{father}\,(X, Y)\ \&\ \text{mother}\,(Y, Z). \end{aligned}$$

These clauses may be combined and simplified to

$$\begin{aligned} &\forall X, Z \mid: \text{grandfather}\,(X, Z) \Leftarrow \\ &\qquad (\exists\, Y \mid: (\text{father}\,(Y, Z) \vee \text{mother}\,(Y, Z))\ \&\ \text{father}\,(X, Y)). \end{aligned}$$

In declarative logic programming, there is no need for disjointness of the conditions contributed by the separate clauses.

3.8 Shared store parallelism

The parallel execution of sequential programs (described in 3.4) requires that each parallel component operates on a copy of the initial values of any variables assigned by other components; and run-time interference is thereby excluded. In general, shared-store parallelism does not require this initial copying; and its model is more complicated. The predicate describing program behaviour has been carefully designed to be consistent with any behaviour of any program running in parallel, provided that the two processes respect each other's output variables. As a result, parallel execution is again validly modelled by conjunction.

Consider the example

$$\begin{aligned} &x := x + y \,\|\, y := 2 \times y \\ = &(\exists\, u, u' \mid \text{ start } \leq u \leq u' \leq \text{finish} : y_{u'} = 2 \times y_{\text{start}}\ \&\ \text{changes}\,(y) \leq 1) \\ \&\ &(\exists\, t, t' \mid \text{ start } \leq t \leq t' \leq \text{finish} : x_{t'} = x_{\text{start}} + y_t\ \&\ \text{changes}\,(x) \leq 1). \end{aligned}$$

The conjunction of the two descriptions can be used to analyse the possible outcomes of parallel execution of the assignment. The deciding factor is whether y is updated before x reads it ($u' < t$) or the other way round. Let us use the suggestive abbreviations

$$\begin{aligned} x &= x_{\text{start}} & x' &= x_{\text{finish}} \\ y &= y_{\text{start}} & y' &= y_{\text{finish}}. \end{aligned}$$

Then the parallel execution is described by

$$\begin{array}{ll} & \text{changes }(x) \le 1 \;\&\; \text{changes }(y) \le 1 \\ \& & ((x' = x + y \;\&\; y' = 2 \times y) \\ & \lor(x' = x + 2 \times y \;\&\; y' = 2 \times y \;\&\; \text{ change }(x) \ge \text{ change }(y))) \end{array}$$

where change (x) is the time that x changes.

This example shows that the range of possible effects of parallel execution is described by the disjunction of several cases. As expected, they are the same as executing the two assignments in either order or even simultaneously. The disjunction represents genuine uncertainty or non-determinism of the outcome of parallel execution; it is a result of unavoidable or deliberate failure to control the relative timing of execution of the parallel components. Note that a valid implementation may always select the first alternative, for example by copying the initial states. This prevents even a deliberate plan for processes to interact through shared store. However, such interactions can be achieved on joint termination of the processes, or by means of a broadcast synchronisation signal, as in the Bulk Synchronous Paradigm (BSP) for parallel programming.

4 Specification and Correctness

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. In computing science, this is particularly important, since most particular experiments are very cheaply and reliably conducted by running a computer program, and working them out by theory is so complicated that it would require computer assistance to predict them anyway.

In engineering, predicates are also used in a complementary role, to describe the 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 may be part of a formal contract between the client and the team engaged to implement the product.

Individual requirements placed on the system can be 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. Engineering would be delightfully easy if a fast and economical product could be assembled from two components, one of which was fast and the other one economical.

In summary, both systems and specifications are (conjunctions of) predicates, describing all actual and all desired behaviour respectively. This gives a particularly convenient definition of the concept of correctness, as 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 as an universally quantified implication:

$$\forall\, v, w, \ldots :: P \Rightarrow S$$

where $v, w, \ldots$ are all the variables of the alphabet. E.W. Dijkstra abbreviates this using square brackets to denote universal quantification:

$$[P \Rightarrow S].$$

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 correct design and implementation in engineering practice. The remainder of these lectures will give many examples drawn from computing science.

The progress of a complex engineering project is often split into a number of design stages. The transition between each stage is marked by signing off a document, produced in the earlier stage and used in the later. 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

$$[D \Rightarrow S].$$

Now the proof of the product itself may be discharged by the supposedly simpler task

$$[P \Rightarrow D].$$

Transitivity of implication then ensures the validity of the original goal

$$[P \Rightarrow S].$$

This is a very simple justification of the widespread engineering practice of stepwise design. It is a vindication of our philosophy of interpreting specifications, designs and implementations all as predicates describing the same kind of observable phenomena; the next lecture will show how to deal with phenomena of different kinds.

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

$$[D \mathbin{\&} E \Rightarrow S].$$

The two designs can then be separately implemented as products P and Q such that

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

Their assembly will then necessarily satisfy the original specification:

$$[P \mathbin{\&} Q \Rightarrow S].$$

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.

Explanation of correctness as implication gives a strangely simple treatment of the perplexing topic of non-determinism. Let P and Q be programs with the same alphabet. Their disjunction $(P \vee Q)$ may behave like P or it may behave like Q. In order for this to be correct, both P and Q must be correct. Fortunately, this is also a sufficient condition, as justified by the fundamental logical property of disjunction as the least upper bound of the implication ordering:

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

The account given above 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 task of design and implementation is to cross these levels of abstraction, and to do so without introducing error. A general method will be treated fully in a later lecture.

A simple but common special 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 internal interactions of the components of the implementation. Such a variable serves as a local variable in a program or a bound variable in a mathematical formula. For the implementation to work, such variables must indeed have *some* value, but we do not care what it is. It may therefore be hidden by existential quantification. The quantification is justified by the $\exists$-introduction rule of the predicate calculus:

$$[P \Rightarrow S] \text{ iff } [(\exists\, v \mid: P) \Rightarrow S]$$

whenever the variable v does not occur in S. The variable is removed from the alphabet of P. Existential quantification of local wires or variables of a product can considerably simplify the descriptions, without affecting the range of specifications that will be satisfied.

An important engineering principle is 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. But it remains to design another component X which will be connected to Q, adapting its behaviour to meet the requirement S. More formally, X must satisfy the implication

$$[X \;\&\; Q \Rightarrow S].$$

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! What we want is at the other extreme, the answer that is easiest to implement.

That is why we ask: What is the weakest specification that should be met by the designers of X? In a topdown 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

$$X = (Q \Rightarrow S).$$

This is guaranteed by the law

$$[X \;\&\; Q \Rightarrow S] \text{ iff } [X \Rightarrow (Q \Rightarrow S)].$$

But of course X must not mention any of the output variables of Q. So the answer must hold for all values which Q may give to them, for example

$$X = (\forall\, x, y \mid: Q \Rightarrow S)$$

where out$\alpha Q = \{x, y\}$. This answer will be called the *residual* of S by Q, because it describes what remains to be implemented to achieve S with the aid of Q. The answer is justified again by a simple law

$$[X \;\&\; Q \Rightarrow S] \text{ iff } [X \Rightarrow (\forall\, x, y \mid: Q \Rightarrow S)]$$

whenever $x, y \notin \alpha X$.

4.1 Control of liquid in tank

The primary requirement on an industrial control system is usually 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 only for a relatively short proportion of time. For example, we may wish that the volume should not be above $(maxv - \delta)$ for more than 10 % of any consecutive interval longer than ten seconds. The undesirable condition can be defined as a Boolean function of time (taking values 0,1).

$$\text{risk}_t = (v_t + \delta > maxv).$$

Now the requirement is expressed using integrals.

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

The overall specification is a conjunction of the two requirements expounded above. The purpose of this example is to show that, in the formalisation of a specification, one should not hesitate to use notations chosen from the entire conceptual armoury of mathematics: whatever will express the intention as clearly and directly as possible. This is the only possible protection against the embarrassment and expense of implementing a product which turns out to be not what was wanted.

4.2 A sorting program

One of the requirements on 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

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

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

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

The specification is the conjunction of these two requirements. Alphabet constraints 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. 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 is wanted.

4.3 Greatest common divisor

The most direct way of specifying the greatest common divisor is that it must be a divisor of both its operands, and the greatest such. Such a function f is described by the predicate S:

$$\begin{aligned} S(f) = \forall x, y \mid x, y > 3 : \ & x \bmod f(x, y) = 0 \\ \& \ & y \bmod f(x, y) = 0 \\ \& \ & (\forall z \mid x \bmod z = y \bmod z = 0 : f(x, y) \bmod z = 0). \end{aligned}$$

The functional program $fp(bcd)$, given in section 3.6 meets this specification, in the sense that

$$(\forall\, bcd \mid: fp(bcd) \Rightarrow S(bcd)).$$

Here quantification is over the alphabet of the program, namely the free variable bcd. There are other functions too that meet this slightly unusual weakened specification.

If the functional program is incomplete, and for particular parameter values none of the conditions which guard the clauses is true, then there will usually be many different functions bcd which satisfy the predicate describing the program; and the correctness condition above will not be provable. If the program fails to terminate, this too will usually be signalled by failure of the proof. But there are some subtleties which are here ignored.

4.4 Local wires

In the design of the negation circuit (section 3.1), the sole purpose of the wires s and d is to connect the circuit to power and ground. Their values have no interest in the specification and use of the circuit, and they will never be connected to any other wire. Their existence should therefore be concealed by existential quantification over the relevant variables

$$\exists\, Hs, Ls, Hd, Ld \mid:$$

After this quantification, the predicate describing the circuit simplifies to

$$(Hv \Rightarrow Lw)\ \&\ (Lv \Rightarrow Hw).$$

This is clearly a reasonable external specification of the circuit, at the switching level of abstraction. The direction of the implication shows how information propagates (with reversed polarity) from wire v to w.

Exactly the same treatment is given to local wires in a combinational circuit. For example (3.2)

$$(\exists z \mid: z = x \vee y \;\&\; w = \neg z) = (w = \neg x \wedge \neg y).$$

In general, localisation should be confined to wires in the output alphabet of the predicate. This ensures that the value of the wire is controlled by exactly one of the components included in the assembly. Otherwise, there are problems like floating wires, which would undermine the correct working of the logic, or at least the ability of our simple theory to describe it.

4.5 Buffers in CSP

A communicating process can be specified by an arbitrary predicate describing its traces. For example, consider a simple message buffer, which stores the messages which it has input, and outputs them later on demand. One property of such a process is that it never outputs more messages than it inputs: and usually there is some limit N on the number of messages that it can store. Let the input of a message be denoted by the event a, and the output by b. Then an N-buffer can be specified by

$$0 \leq \mathit{trace.a} - \mathit{trace.b} \leq N.$$

So the example $(P = a \rightarrow b \rightarrow P)$ of section 2.7 is a 1-buffer.

When communicating processes are combined to operate in parallel, they interact with each other by simultaneous participation in events which are common to both their alphabets. Typically, such an event is the communication of a message on some channel which connects the processes. If that is the sole purpose of the channel, then the occurrence of the shared events, the values of the messages, and even the existence of the communication channel are of no interest to the surrounding environment; once the component processes have been composed, they should be hidden from external observation. The obvious way of doing this is simply to remove all record of the occurrence of the event to be hidden from the trace of the combined processes. For example, consider the pair of buffers defined in 3.5. Their joint behaviour is described by

$$\begin{aligned} & 0 \leq \mathit{trace.a} - \mathit{trace.b} \leq 1 \\ \& \quad & 0 \leq \mathit{trace.b} - \mathit{trace.c} \leq 1. \end{aligned}$$

After assembly of the components, the event b represents just internal communication between them. The effect of concealing the event b is to make the value of $\mathit{trace.b}$ unobservable; it must have some value, but we neither know nor care what it is:

$$\begin{aligned} \exists n \mid: \quad & 0 \leq \mathit{trace.a} - n \leq 1 \\ \& \quad & 0 \leq n - \mathit{trace.c} \leq 1. \end{aligned}$$

By simple arithmetic, this simplifies to

$$0 \leq \ trace.a - \ trace.c \leq 2,$$

confirming the obvious fact that a buffer of depth two can be constructed by connecting two single buffers — after the channel that they share has been hidden.

4.6 Residual in combinational circuits

Consider a specification

$$S = (z = (x \not\equiv y)).$$

Suppose that the design so far has progressed to a stage

$$Q = (z = (x \wedge \neg y) \vee w)$$

where out$\alpha Q = \{z\}$. What remains to be implemented is the residual

$$(\forall z \mid: Q \Rightarrow S).$$

By considering separately the two cases $z =$ true and $z =$ false, this can be simplified to a conjunction

$$D = (w \Rightarrow (x \not\equiv y))\&(\neg x \wedge y \Rightarrow w).$$

The task of implementing this design is fortunately simple, indeed obvious.

$$P = (w = \neg x \wedge y)$$

An alternative design decision

$$P = (w = x \not\equiv y)$$

is equally correct, but pointless, because it is no easier to implement than the original specification S. The use of a magic formula does not reduce the need for common sense judgements about the general direction of design. The primary role of the mathematics is to control complexity of the detail, where human judgement is less reliable in achieving correctness.

4.7 Residual in sequential programs

Suppose the task of a loop is to maintain a constant value for the expression $(x - y)$. This task is expressed in the specification

$$S = (x' - y' = x - y).$$

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

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

What change must be made to the value of x in order to establish S? The answer is given by the residual

$$\begin{array}{ll} & (\forall\, y' \mid: y' = 2 \times y \Rightarrow (x' - y' = x - y)) \\ = & (x' - 2 \times y = x - y) \\ = & x' = x + y \\ = & x := x + y. \end{array}$$

The answer, which is not totally obvious, has been derived by pure calculation.

But the technique is not magical, and sometimes it just can't work. Suppose the specification is to make the product and y odd, so the required residual is

$$\begin{array}{ll} & (\forall\, y' \mid: y' = 2 \times y \Rightarrow (x' \times y' \text{ is odd})) \\ = & x' \times 2 \times y \text{ is odd} \\ = & \text{false}. \end{array}$$

This is, of course, unimplementable: there is no way that an odd product can be obtained by doubling one of the factors. Fortunately, the calculation of the residual gives clear warning of the impossibility of implementation.

5 Algebra and Normal Forms

The interpretation of designs as predicates permits all questions of correctness to be resolved in principle by reasoning in the predicate calculus. But proofs conducted exclusively in the predicate calculus are laborious to the point of impracticality. In established branches of engineering, the problem is solved by division of labour. For example, applied mathematicians reason primarily by arithmetic and symbolic calculation, using an appropriate collection of algebraic laws; the proof of these laws is the proper contribution of pure mathematicians, who enjoy the necessary facility in predicate logic.

A telling example from classical mathematics is provided by the distinction between calculus and analysis. The differential calculus is defined by a collection of algebraic laws, for example

$$\frac{dx^2}{dx} = 2 \times x.$$

The definition of the differential operator expands this law to a predicate

$$(\forall\, \epsilon \mid \epsilon > 0 : \exists\, \delta \mid: \left| \frac{x^2 - (x - \delta)^2}{\delta} - 2 \times x \right| \leq \epsilon).$$

It is this predicate that is proved by the branch of pure mathematics known as analysis; all other mathematicians, including scientists and engineers, simply use the calculus, without any need (and certainly with no desire) ever to expand the abbreviations again.

The simplicity and power of algebraic reasoning is achieved by severely restricting the notations in which the reasoning is conducted. This is necessary because the validity of the algebraic laws often depends on powerful assumptions made about the range of values of the free variables appearing in them. For example, the simplest version of the differential calculus assumes that all functions involved are everywhere differentiable. Substitutions for the free variables must be restricted to terms whose syntactic form guarantees the assumptions. In particular, the operators appearing in all the terms must be confined to those which ensure their results will satisfy the assumptions, whenever their operands do. For example, simple versions of differential calculus have to exclude division from the list of permitted operators.

The presentation of an algebraic theory begins, like the definition of a new programming language, with a specification of the syntax of allowable terms. This is given by a *signature*, consisting of an enumeration of the atomic terms (or constants) of the language, together with a list of the operators, which may be used to combine them into larger terms. That defines the syntax: the substance of the algebra is presented by a collection of algebraic equations between terms containing free variables as well as the operators and constants of the signature. The laws are proved or claimed to be valid for all permitted values of the free variables, namely those that are expressible in the restricted notations of the signature. In our examples, the free variables are *second-order*, in that they range over *predicates*, which themselves contain other free variables denoting observations: these first-order variables are seldom mentioned explicitly in the laws, but they are assumed to satisfy the alphabetic constraints imposed in the definition of the operators.

For any given signature of atoms and operators, there is no end to the list of algebraic laws that might be postulated or proved by the pure mathematician. But the task may be considered accomplished when enough laws have been proved that all other equations (expressible within the notational limitations of the signature) can be proved from those already presented. These proofs should be conducted by algebraic reasoning alone, perhaps with a little induction, and certainly without expanding the definitions of the operators. The presentation of the collection of laws will then be accompanied by a proof of their completeness in this sense.

A good criterion for the completeness of a collection of laws is the existence of a *normal form*, and a proof that every term of the language can be reduced to this form by algebraic substitution alone. The test for equality or implication between normal forms should be easier than between arbitrary terms. Ideally, it should be just textual identity or containment of normal forms, but other mechanically checkable equivalences are also allowed.

A familiar example of an algebraic presentation is given by Boolean Algebra. Here the operators include conjunction, disjunction and negation; and there are enough laws to reduce each term to a normal form, for example, the *conjunctive* normal form CNF. This is written as a conjunction of a number of *clauses*: each clause is a disjunction of a number of distinct *literals*, each of which is either an *atom* or the *negation* of an atom. For example $(v \equiv \neg w)$ "reduces" to the normal form

$$(v \vee w) \wedge (\neg v \vee \neg w).$$

(As usual, it is actually an expansion in the size of the text.) Two normal forms are equal if they can be made textually identical by reordering the operands of the conjunctions and the disjunctions.

5.1 Logic programs

A clause in a conjunctive normal form is called a *Horn clause* if it contains exactly one positive literal, and all the other literals are negated atoms. A predicate has the Horn property if it is expressible as a conjunction of Horn clauses. A Horn clause

$$(h \vee \neg a \vee \neg b \vee \ldots)$$

is usually written as an implication backwards

$$(h \Leftarrow\ a \wedge b \wedge \ldots).$$

Here h is called the *head* of the clause, and the rest is called its *body*. All the clauses of a PROLOG program are Horn clauses, though the notation is different:

$$h\text{:- } a,\ b,\ \ldots$$

If the head is the only literal, this can be written

$$(h \Leftarrow \mathit{true}) \quad \text{or} \quad h\text{:-}$$

Boolean algebra can be conveniently extended by allowing quantification over a local Boolean variable, for example,

$$(\exists\, x \mid: B(x)) = B(\mathit{true}) \vee B(\mathit{false}).$$

The right hand side of this equation can be reduced to conjunctive normal form by standard techniques. More interesting, the normal form will have the Horn property if the original term had that property. This means that localisation of predicate names in a logic program can be eliminated, at least in the propositional case.

The proof that $\exists$ preserves the Horn property is given by displaying the laws of Boolean logic that can be used to effect the reduction of $\exists x \mid: B(x)$ to Horn clauses, on the assumption that $B(x)$ is expressed the same way. Firstly, separate out all clauses of $B(x)$ that contain x at their head (if there are none, add the vacuous clause $x \Leftarrow$ false). The result can be written:

$$\exists x \mid: (\& \ i \mid i \in S : (x \Leftarrow A_i)) \ \& \ (\& \ j \mid j \in T : h_j \Leftarrow B_j(x)).$$

By Boolean algebra, this reduces to

$$(\& \ i, j \mid i \in S \wedge j \in T : h_j \Leftarrow B_j(A_i)) \quad \text{if } S \neq \{\},$$

which is a conjunction of Horn clauses, (or just *true*, if T is empty).

The purpose of the restriction of logic programs to Horn clauses is to permit a particularly efficient method of implementing the answer to an arbitrary query. Each clause is implemented as the declaration of a Boolean function, with the given head and body; each atom in the body is implemented as a call on the function with a head that matches it. If there is no match, the answer is taken as *false*.

The fact that unmatched calls give the answer *false* is absolutely necessary for the logic program ever to give a negative answer to any question at all. Without it, any collection of Horn clauses can always be satisfied by taking all its atoms as true! Prolog in fact goes to the opposite extreme: it will give a negative answer whenever this is consistent with the program: positive answers will be given only when forced. To predict when false answers will be given, the predicate defined by the program must be strengthened.

The technique of strengthening is as follows. First collect together all clauses with x at their head:

$$(\& \ i \mid i \in S : x \Leftarrow A_i).$$

Logically, this is equivalent to a single clause whose body is the disjunction of all the A_i:

$$x \Leftarrow (\bigvee i \mid i \in S : A_i).$$

The value of x computed by the logic program is the falsest possible one, namely

$$x = (\bigvee i \mid i \in S : A_i).$$

If S is empty, this correctly reduces to $x =$ false. The actual behaviour of the logic program is described exactly by making this change for all heads of clauses appearing in the program. After this strengthening, it is not legitimate to add any further clauses which share the same heads. This restriction can be enforced by including all the strengthened heads in the output alphabet of the program.

In logic programming texts, the replacement of implication by equality is said to be justified by the *closed world assumption.* This is a rather mystical misnomer. It is not an assumption, but a correct scientific description of the actual answers given by execution of the logic program. The programmer's responsibility is to write enough clauses in the program to ensure that the computed answers correspond to what is wanted in the world outside the computer.

5.2 Transistor networks

The predicate describing a transistor can easily be rewritten in the form of a pair of Horn clauses; for example, an N-transistor is described by

$$\begin{array}{rcl} Hs & \Leftarrow & Lg \wedge Hd \\ \& \; Hd & \Leftarrow & Lg \wedge Hs. \end{array}$$

Connection to power and ground give Horn clauses with empty bodies. So any description of a switching circuit, being a conjunction of transistors, has the Horn property, which it retains even after hiding the local wires. The predicate can easily be rewritten as a logic program, whose execution gives the same result as etching the circuit on the surface of a silicon chip, and connecting its leads to a battery. The actual propagation of signals through the transistors causes the source and drain to become connected only if this is forced by the predicate describing the transistor; and the replacement of implication by equality is justified by the same reasoning as in the previous section. As a result, the familiar negation circuit

$$(Lv \Rightarrow Hw) \;\; \& \;\; (Hv \Rightarrow Lw)$$

is actually described by the stronger predicate

$$(Hv = Lw) \;\; \& \;\; (Lv = Hw).$$

Now at last it is possible for the designer to discharge the proof obligation that allows the network to be used as a combinational logic gate, namely that every wire is connected to High or Low. This is formally expressed by the equations

$$Hw = \neg Lw, \quad \text{for all wires } w.$$

We define the *output* wires of the network as those whose names appear in the head of any clause, and the input wires are the rest. Let POST be the assertion that all output wires are two-valued, and let PRE say the same about the input wires. The responsibility for ensuring PRE is placed upon the environment in which the circuit is to be used: so the obligation of circuit designer is to prove

$$\text{PRE} \Rightarrow \text{POST}.$$

This is a specification that must be satisfied by the strengthened version of the circuit description; for example, in the case of the negation circuit

$$(Lw = Hv \;\&\; Hw = Lv) \Rightarrow ((Hv \not\equiv Lv) \Rightarrow (Hw \not\equiv Lw)).$$

Note that this essential theorem cannot be proved without prior strengthening of the predicate describing the circuit.

Now we can see exactly the reasons for the alphabetic restrictions on the composition of combinational logic gates. The restriction on sharing wires in their output alphabets is needed to ensure that the separate strengthening of the predicates describing individual logic gates has the same effect as strengthening applied to the whole combinational network after assembly. The prohibition on combinational cycles is what ensures that the proof of the two-valued property of the whole circuit can be validly constructed from proofs that each individual gate propagates two-valuedness from its input wires to its output. Otherwise the proof would be circular — a mathematical error just as serious as the corresponding hardware design error.

5.3 Combinational logic

The description of a single logic gate is always a single equation with its output variable standing alone on the left hand side. The description of a combinational circuit is just a conjunction of a number of these equations; the left hand side is always a single output wire and the right hand side is a Boolean expression containing both input and output wires. Because of the restriction against connecting two output wires, each output wire name occurs on the left hand side of exactly one equation. Because of the restriction against cyclic connection, the equations can be sorted into an order ensuring that the first occurrence of each output wire name is on the left hand side of its defining equation. After this reordering, it is possible efficiently to simulate the behaviour of the circuit just by executing the equations as a sequence of assignments in a high level language.

The sorting of the equations is preserved by existential quantification over an output wire. The existential quantifier $\exists\, w.$ can be eliminated by the algebraic law

$$(\exists\, w \mid: P \;\&\; w = e \;\&\; Q(w)) \quad = \quad P \;\&\; Q(e).$$

Because the formula has been sorted, the equations P that preceded the equation for w do not contain w, and they are left unchanged; in all subsequent equations, the variable w is replaced by the right hand side of the defining equation, and the defining equation is omitted. The resulting sequence of equations is still sorted.

When two sets of sorted equations are combined by conjunction, it may be necessary to sort again the entire collection. In some cases, this will not be possible. These are just the cases in which there is a cycle of components and wires in the hardware realisation of the circuit; so the combination would be impossible to implement anyway. The normal form acts as a check against this mistake.

5.4 Sequential programs

In this section we will consider programs with only one global variable x. Its alphabet consists just of the variables $\{x, x'\}$, with x' as output. The only assignments that can be written take the form

$$x := f.x,$$

where f is a total function representable as an expression of the programming language. In fact, the restriction to a single variable is not serious; all our reasoning will apply equally to longer lists of variables. These lists will appear on the left hand side of multiple assignments, for example

$$x, y := e.(x, y), f.(x, y),$$

in place of the single assignment above.

Assignments will be chosen as the atoms of our algebraic theory. The operators will include the conditional (defined previously), disjunction (representing non-determinism), and a new operator for sequential composition defined:

$$P(x, x');\ Q(x, x') = (\exists\, v \mid: P(x, v)\ \&\ Q(v, x')).$$

The result of the composition has the same alphabet $\{x, x'\}$ as its operands. Implementation is easy: just execute P first in initial state x to obtain its final state v; then v is fed to Q as its initial state. The final state x' of Q is taken as the final state of the whole construction.

In reducing an expression to normal form, the most useful laws are those that eliminate occurrences of each operator. For example, the law

$$(x := e.x\ ;\ \ x := f.x) = (x := f.(e.x))$$

will eliminate semicolon between any list of consecutive assignments, since presumably $f.(e.x)$ is a valid expression of the language. Similarly, if the language contains conditional expressions, the following law will eliminate conditionals between assignments in favour of conditional expressions

$$(x := e \lhd b \rhd x := f) = (x := (e \lhd b \rhd f)).$$

If sequencing and conditional were the only operators of the algebra, we have already shown that the syntactically defined subclass of just assignments constitute a normal form! Equality then depends only on equality of the right hand sides.

To deal with disjunction, we need laws that show how the other operators distribute through it in both directions:

$$\begin{array}{lcl} (P \vee Q);\ R & = & (P;\ R) \vee (Q;\ R) \\ P;\ (Q \vee R) & = & (P;\ Q) \vee (P;\ R) \\ (P \vee Q) \lhd b \rhd R & = & (P \lhd b \rhd R) \vee (Q \lhd b \rhd R) \\ P \lhd b \rhd (Q \vee R) & = & (P \lhd b \rhd Q) \vee (P \lhd b \rhd R). \end{array}$$

All of these are provable in Boolean algebra — they can even be checked by truth tables. By induction, these laws can be extended to disjunctions with an arbitrary number of operands.

To accommodate disjunction, the concept of the normal form must be extended to a *nonempty set* A of assignments, separated by disjunction, which will be written

$$\bigvee A.$$

Laws of distribution through disjunction can then be used to bring it out as the outermost operator of the program:

$$\begin{array}{lcl} (\bigvee A);\ (\bigvee B) & = & \bigvee\{a;\ b | a \in A \ \&\ b \in B\} \\ (\bigvee A) \lhd d \rhd (\bigvee B) & = & \bigvee\{a \lhd d \rhd b | a \in A \ \&\ b \in B\} \\ (\bigvee A) \vee (\bigvee B) & = & \bigvee(A \cup B). \end{array}$$

The occurrences of $(a;\ b)$ and $(a \lhd d \rhd b)$ on the right hand side of these laws can be reduced to assignment by the two laws already given. Equality of normal forms is tested as equality of the sets enumerated by the right hand sides of the assignments.

Consideration of normal forms can reveal interesting properties shared by all programs; they are only a very small subset of all predicates with alphabet $\{x, x'\}$. For example, the predicate $(x < 3)$ can never be expressed in the normal form, because every assignment allows an arbitrary initial value (the functions are all total). The predicate $(x' > 3)$ is not expressible, because it allows an infinite range of results, whereas every normal form is only a finite disjunction. In fact, a necessary condition for expressibility of $P(x, x')$ as a program is

$$(\forall x | : \{x' | P(x, x')\} \text{ is finite and non-empty}).$$

Because of incomputability, this is not a sufficient condition.

In specifying and reasoning about programs, there is absolutely no need to restrict our expressive power to predicates which correspond to programs. Let us extend our consideration to all predicates with only two free variables, one in its input alphabet and one in the output. The operators on such predicates include all those of Boolean Algebra, including disjunction, conditional and even negation. Sequential composition is definable in exactly the same way as in the programming language. Finally, each predicate P can be converted to a predicate (denoted $\breve{P}$), simply by reversing the roles of the input and output alphabets

$$\alpha(\breve{P}) = \alpha(P), \text{ and out}\alpha(\breve{P}) = \text{ in}\alpha(P).$$

The composition and converse are identical to those defined for arbitrary relations; and so they obey all the beautiful laws of the relational calculus.

A good example of the power of this calculus is the definition of an approximate inverse for sequential composition. This is an answer to the question: What is the weakest specification X of a program which, when followed by Q, will achieve specification S

$$[X;\ Q \Rightarrow S].$$

The answer is

$$X = \neg(\neg S;\ \breve{Q}),$$

which we will denote $S\%Q$, and call the *residual* of sequential composition. The relevant law of the relational calculus is

$$[P;\ Q \Rightarrow S] \quad \text{iff} \quad [P \Rightarrow S\%Q].$$

Note the similarity of the law that makes positive integer division (div) an approximate inverse of integer multiplication:

$$p \times q \leq s \quad \text{iff} \quad p \leq s \text{ div } q.$$

The residual is intended to help in the topdown development of a program to meet a specification S with the aid of a known program Q. For example

$$\begin{aligned} (x' - y' = x - y)\%(y := 2 \times y) \quad &= \quad (x' - 2 \times y = x - y) \\ &= \quad x := x + y. \end{aligned}$$

The residual sometimes gives an unimplementable specification, for example

$$(x' \times y' \text{ is odd div } y := 2 \times y) = \textit{false}.$$

This indicates that nothing can be done before $y := 2 \times y$ to make $x \times y$ odd afterwards.

In more complicated cases, the check on implementability of the residual may be helpful in selecting components to help in the task, and avoiding those that hinder it.

6 Abstraction and Quantification

In previous sections, we have described how to formulate a scientific theory, and how to use it for reasoning about specifications, designs and implementations. The method has been illustrated by a number of separate theories, which describe computer software and hardware at varying levels of granularity and abstraction. Reasoning within a single one of these theories is somewhat simplified by the assumption that all the predicates involved have the same alphabet, and that in every law the variables of each predicate have the same meaning in terms of observations. In this lecture we will begin to build bridges between the separate theories, and show how to move securely between their differing levels of abstraction.

We have given one example of a mismatch of alphabet between an implementation and a specification. This occurs when an implementation needs a local variable (e.g. a wire) to model the interactions between its components. Such interactions are an internal feature of the assembly, and they are not intended to be observable from the outside: the variable can therefore be hidden by existential quantification, which reestablishes a perfect match of alphabets between the implementation and the specification. But this technique cannot begin to deal with the more general case, for example when the specification and implementation alphabets are completely disjoint. Quantification over the entire alphabet of a predicate leads to a constant value, either *true* or *false*. The first of these is vacuous and the second unimplementable; they are equally useless for all practical purposes.

We therefore need to extend the concept of correctness so that it applies even when the alphabets of the implementation and the specification are disjoint. For example, we will explain how the gates of combinational logic networks are actually implemented by transistor networks, even though their alphabets are disjoint. Sequential circuits are used to implement the functions of a computer, which must conform to a given architecture. Machine language programs produced by a compiler must faithfully implement algorithms expressed in higher level languages. In each of these cases there is a significant level of abstraction to cross; the observations at each level of abstraction are of a different nature from those at the next or any other level. The same mismatch is seen in the natural sciences, where the conceptual and observational framework changes (for example) between the quantum, the atomic and the molecular levels. In engineering too, design of a complex product passes through many levels, from capture and analysis of requirements through software design and implementation down to selection and assembly of electronic components. It is the crossing of conceptual levels that presents the greatest risk of subtle and persistent errors: their avoidance is a major goal of engineering methodology.

The general technique for crossing a level of abstraction is to define the way in which an observation at one level of abstraction corresponds to one or more observations at the other level. This relationship can itself be described by a predicate (often called a *linking invariant*)

$$L(c, a),$$

which relates an abstract observation a (in the alphabet of the specification) to a more concrete observation c (in the alphabet of the implementation). Let $P(c)$ be a product description describing concrete observations c. Then the quantified predicate

$$\exists\, c \mid L(c, a) : P(c)$$

describes all the ways in which the product P can give rise to an abstract observation a. Let $S(a)$ specify the desired properties of the abstract observations. Then the correctness of the product is expressed by the implication

$$[(\exists\, c \mid L(c, a) : P(c)) \Rightarrow S(a)]\,.$$

The linking invariant is what ensures complete match between the alphabets on both sides of this implication.

Note that the predicates P and S are allowed to share observational variables that are not mentioned in L. However we strictly maintain the restriction that no variable appears in the alphabet of all three predicates $P, L,$ and S. Such common variables would serve no purpose; they would merely invalidate a number of laws useful for calculation.

The linking invariant acts like a conceptual component of the concrete system, translating from concrete to abstract observations. The concrete variables, since they do not appear in the specification, are then existentially quantified in the usual way. Indeed, ordinary localisation is a special case, arising from a linking invariant that is everywhere true; its abstract alphabet is empty:

$$L(c) = (c = c).$$

In the top-down design of an implementation, we want to ask the converse question: Given a linking invariant $L(a, c)$, what is the weakest description $X(c)$ of the concrete observations c that will satisfy the specification $S(a)$ describing the corresponding abstract observations a. The answer is

$$X(c) = (\forall\, a \mid L(c, a) : \; S(a)).$$

In a commonly recommended special case, the abstract observation is fully determined by the concrete one, so that the linking invariant can be expressed as an equation using a function f:

$$a = f.c.$$

In this case, the concrete specification ("concretion") can be calculated simply by substitution

$$(\forall\, a \mid L(c,a) : S(a)) = (c \in \text{dom}.f \Rightarrow S(f.c)).$$

If f is a total function, the antecedent can be omitted. Similarly, if the abstract observation determines the concrete by the function g

$$(\exists\, c \mid : c = g.a : P(c)) = (a \in \text{dom}.g \;\&\; P(g.a)).$$

In the case of a total function, both abstraction and concretion have the same simple effect, namely substitution of a formula for a variable. In continuous mathematics, such a substitution is known as a change of coordinates.

The algebra of abstraction can be greatly simplified by use of the notations of sequential composition and its residual. The alphabet of a linking invariant L can be partitioned into its abstract variables absαL and its concrete variables conc$\alpha L (= \alpha L - \text{abs}\alpha L)$. Provided that no variable of P occurs in the abstract alphabet of L, we define

$$P;\ L = (\exists\, c_1, c_2, \ldots \mid L : P),$$

where quantification is over all variables $c_1, c_2, \ldots$ in the concrete alphabet of L. The residual operator is defined by

$$S\%L = (\forall\, a_1, a_2, \ldots \mid L : S),$$

provided that $\alpha S \cap \alpha L = \text{abs}\alpha L$. Here quantification is over all variables in absαL. The alphabets of $(P;\ L)$ and $(S\%L)$ are just the symmetric difference of the alphabets of their operands:

$$\alpha(P;\ L) = (\alpha P \not\equiv \alpha L) = (\alpha P \cap \neg\alpha L) \cup (\alpha L \cap \neg\alpha P).$$

The fundamental Galois connection between $(\,;\,)$ and $(\%)$ can now be written as before

$$[P;\ L \Rightarrow S] \quad \text{iff} \quad [P \Rightarrow S\%L].$$

The converse $\sim L$ of a linking invariant is obtained by just reversing the roles of the concrete and abstract alphabets, leaving unchanged the meaning of the predicate. Predicate calculus reveals that

$$\neg(S\%L) = (\neg S;\ \sim L).$$

Clearly also

$\sim\sim L = L$

$\sim$ distributes through all Boolean combinators

$P;\ (L1 \vee L2) = P;\ L1 \vee P;\ L2,$ *etc.*

Linking invariants can be composed by sequential composition, under the usual alphabetic constraints. $(L1;\ L2)$ is defined just when

$$\alpha L1 \cap \alpha L2 = \mathrm{abs}\alpha L1 = \mathrm{conc}\alpha L2.$$

In this case

$$\mathrm{abs}\alpha(L1;\ L2) = \mathrm{abs}\alpha L2.$$

Assuming these constraints, we get associativity

$$L1;\ (L2;\ L3) = (L1;\ L2);\ L3,$$

and all the other axioms of the relational calculus. These include a number of interesting distribution properties

$$\begin{aligned}
\sim(L;\ M) &= (\sim M);\ (\sim L)\\
\neg(L;\ M) &= (\neg L)\%(\sim M)\\
\neg(L\%M) &= (\neg L);\ (\sim M)\\
\sim(L\%M) &= (\neg M);\ (\sim L)\\
(L\%M)\%N &= L\%(N;\ M).
\end{aligned}$$

6.1 Transistor nets implement logic gates

The description of a combinational circuit uses the single wire name w as a Boolean variable, with just two states, High or Low. The switching circuit uses two variables Hw and Lw for each wire, and is capable of describing two additional states, when the wire is undriven and when it is short-circuited. If the switching circuit is intended for use as a combinational circuit, it is the designer's responsibility to ensure that these additional states never occur. In that case, we may code the linking invariant for each abstract wire w as

$$\mathrm{Link}_w \hat{=} (w = Hw\ \&\ \neg w = Lw).$$

When applied to the negation circuit, this gives the expected result:

$$((Hv \Rightarrow Lw)\ \&\ (Lv \Rightarrow Hw));\ (\mathrm{Link}_v\ \&\ \mathrm{Link}_w) = (w = \neg v).$$

As an example of top-down design of a transistor network, let us start with a logic gate specification $w = \neg v$. The weakest transistor network satisfying this specification is

$$(w = \neg v)\%(\mathrm{Link}_w\ \&\ \mathrm{Link}_v) = (Hw \neq Lw\ \&\ Hv \neq Lv \Rightarrow (Hv = Lw)).$$

This specification can be trivially satisfied by connecting w or v to both High and Low. But that is ruled out by the proof requirement described in 5.2.

6.2 Combinational logic implements natural numbers

A combinational circuit may be required to implement some function on natural numbers, say addition. The clearest specification is given in terms of abstract variables A, B, C ranging over natural numbers:

$$A = B + C.$$

To implement this specification by means of the voltages on the wires of a combinational circuit, it is necessary to specify how each natural number is represented. For practical reasons, the size of the numbers is usually bounded by some power of two, say L. The usual binary representation can be specified as a linking invariant:

$$\begin{aligned} &BIN_A^L = (A = \sum_{n=0}^{L} a_n \times 2^n) \\ &\text{absa}BIN_A = \{A\}. \end{aligned}$$

Here the subscripted variables a_n are names of wires in the combinational circuit. Their desired behaviour is specified by the residual

$$(A = B + C)\%(BIN_A^{L+1} \ \& \ BIN_B^L \ \& \ BIN_C^L).$$

The result can be calculated by substitution:

$$\sum_{n=0}^{L+1} a_n \times 2^n = \sum_{n=0}^{L} b_n \times 2^n + \sum_{n=0}^{L} c_n \times 2^n.$$

The usual implementation introduces an internal array of wires $d_0, d_1, \ldots, d_L$, which communicate the carry between components calculating the individual digits of the result:

$$\begin{aligned} d_0 &= 0 \\ d_{n+1} &= \text{majority } (b_n, c_n, d_n) \\ a_n &= \text{oddparity } (b_n, c_n, d_n) \end{aligned}$$

where majority $(x, y, z) = (x \wedge y) \vee (y \wedge z) \vee (z \wedge x)$ and oddparity $(x, y, z) = (x \not\equiv y \not\equiv z)$.

6.3 Sequential circuits implement combinational

Let V and W be wires of a sequential circuit described by $S(V, W)$; their values are functions from time to Boolean. Let v, w be Boolean variables naming the same wires when considered as a combinational circuit. Their values may be observed at the end of any clock cycle during the operation of the sequential circuit. The linking invariant describes all the ways that these combinational observations may arise,

$$LK = (\exists\, t \mid: v = V_t \;\&\; w = W_t)$$
$$\text{abs}\alpha LK = \{v, w\}.$$

The strongest combinational circuit implemented by S is described by

$$S;\; LK.$$

In the case of the negation circuit

$$((\forall\, t \mid:\; W_t = \neg\, V_t);\; LK) = (w = \neg v).$$

If this construction is applied to a sequential circuit with different behaviour on each cycle, its combinational specification becomes non-deterministic. For example

$$(\forall\, t \mid:\; W_{t+1} = \neg\, V_t);\; LK = \mathit{true}.$$

This says that within the limited conceptual framework of combinational circuits, it is impossible to predict or control the time-varying behaviour of a genuinely sequential circuit.

The converse question: what is the weakest sequential circuit that implements a combinational specification $C(v, w)$? The answer is

$$C(v, w)\%LK = \forall\, t :: C(V_t, W_t).$$

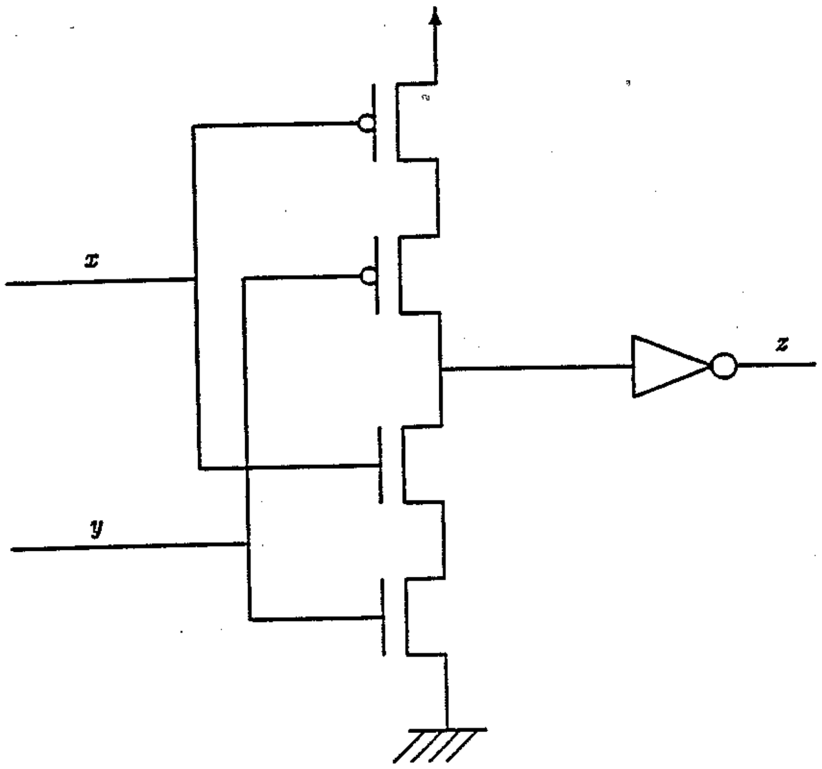
The residual operator has the required effect of decorating all the combinational wire names with a single universally quantified subscript, denoting the time of the observation. Furthermore any sequential circuit which displays purely combinational behaviour can be written in the form $(C(v, w)\%LK)$ for some combinational predicate $C(v, w)$.

6.4 From transistors to sequential circuits

In a transistor network, if at the end of a cycle a wire is left disconnected from both High and from Low, it will tend to retain the voltage which it had at the end of the previous cycle of operation. This tendency can be reinforced to any desired degree by electronic measures, which need not concern us here. It is the retention of voltage that permits sequential circuits to be constructed from transistor networks. The effect can be described by a linking invariant between the transistor alphabet $\{Hw, Lw\}$ and the subscripted variable w_t which describes the same wire in the alphabet of sequential circuits. The invariant is:

$$\begin{aligned} SEQ_w = \forall\, t \mid:\; & w_t = (Hw_t \vee (\neg Lw_t \wedge w_{t-1})) \\ & \&\; \neg w_t = (Lw_t \vee (\neg Hw_t \wedge \neg w_{t-1})). \end{aligned}$$

A simple component in a sequential circuit is the Muller C-element. In the diagram, unnamed wires are assumed to be hidden.



After hiding, simplification and strengthening, the transistor network is described by

$$(Lx \wedge Ly = Lz) \;\&\; (Hx \wedge Hy = Hz). \qquad (*)$$

As a sequential circuit, its behaviour is described by

$$(*);\ (\mathrm{SEQ}_z \;\&\; \mathrm{LINK}_x \;\&\; \mathrm{LINK}_y).$$

After simplification, this reduces to

$$z_t = \text{majority } (x_t, y_t, z_{t-1}).$$

The output takes the same value as its inputs, provided that they agree; otherwise it retains its previous value until they agree again.

6.5 Functions implement relations

Let x be a variable standing for the argument of the call of a function, and let y stand for its result. The function can often be conveniently specified by a predicate $R(x, y)$ which relates its argument x to its result y. A function f meets this specification just when its graph is contained in the relation:

$$(\forall x \mid x \in \text{dom}.f : R(x, f.x)).$$

This can be rewritten as $(R\%LK)$, where

$$LK = (x \in \text{dom}.f \ \&\ y = f.x)$$
$$\text{and abs}\alpha LK = \{x, y\}.$$

If $fp(f)$ describes a functional program with alphabet $\{f\}$, the strongest relational specification which it satisfies is therefore $(fp(f);\ LK)$, written in full as:

$$\exists f \mid (x \in \text{dom}.f \ \&\ y = f.x) :\ fp(f).$$

This describes the union of the graphs of all functions f satisfying $fp(f)$.

Working in the opposite direction, relations can also implement specifications described in terms of functions. Let $fp(f)$ be such a specification, with f now serving as an abstract observation. Let x, y be concrete observations of the initial and final states of a relational program. For these to satisfy the specification, they must be observations of one of the functions permitted by fp.

$$\exists f \mid x \in \text{dom}.f \ \&\ y = f.x :\ fp(f).$$

This can be abbreviated to

$$fp;\ \ LK.$$

If the relational program is run several times on the same initial values x, different values of y may be observed, provided these are consistent with different functions satisfying fp. The non-determinancy of the specification of the function allows non-determinism of its implementation.

The final question is: given a concrete sequential program described by $P(x, y)$, what is the strongest predicate that describes exactly the functions implemented by P? There is no general answer to this question. Consider the predicate

$$x = 0 \ \&\ y = 0.$$

This separately satisfies both the functional specifications

$$f = \text{ the identity function}$$
$$f = \text{ the squaring function.}$$

But the conjunction of these two specifications is *false*, which is certainly not satisfied by $(x = 0 \;\&\; y = 0)$, or anything else.

6.6 Sequential programs implement parallel

An observation of a sequential program is described by an alphabet (v, v') of initial and primed variables. The alphabet of a parallel program is a pair of times (start, finish) and a function V which maps each t (between start and stop) onto the values of the program variables v at that time. The predicate which links these observations states clearly the times at which the initial and final observations are made:

$$LK = (V_{\text{start}} = v \;\&\; V_{\text{finish}} = v').$$

This link may be used to convert a shared-store parallel assignment to a sequential one:

$$(V := f.V);\; LK.$$

Expanding the abbreviations:

$$\begin{array}{l} \exists\, V, \text{start}, \text{finish} \mid V_{\text{start}} = v \;\&\; V_{\text{finish}} = v' : \\ \qquad \exists\, t, t' \mid \text{start} \leq t \leq t' \leq \text{finish} : \\ \qquad\qquad V_{t'} = f.(V_t) \;\&\; \text{changes}\,(V) \leq 1 \end{array}$$

which reduces to the expected

$$v' = f.v.$$

In the simpler theory of sequential programs this predicate is written in the same assignment notation: $v := f.v$. The use of the same notation in two different theories is common in mathematics; it is justified by a clear mathematical link between their meanings.

7 Failures and Preconditions

The philosophical message of these lectures has been illustrated by a number of theories drawn from different branches of computing science. In each case, the simplest possible theory has been selected to make the relevant point. The time is now ripe to confess some of the problems that we have ignored so far, and to make suggestions on how they may be overcome. Usually this requires a more complicated theory; and as usual, it is important to explore the links between the simpler and the more complex theories, so as to formulate the precise conditions under which the simpler theory can safely continue to be used.

In a theory that is intended to help in engineering design, it is desirable to model accurately all the ways in which a product can fail. This often requires an assumption that failure is observable and denotable by a variable in the alphabet of the design. It is only this that enables engineers to prove absence of failure in a particular design project. If engineers discharge their primary responsibility to avoid failure, the theory does not have to predict accurately the other observations that may accompany the failure; it is often mathematically convenient to leave them wholly arbitrary.

A powerful general method of dealing with failure is to associate a second predicate with each design, known as its *precondition*. This precondition plays no direct part in describing the behaviour of the relevant component or system. Rather, it specifies the expected behaviour of the wider environment in which the system is embedded. If the environment satisfies the precondition, the behavioural predicate can be expected to predict the actual behaviour of the system. Otherwise it can't. Preconditions are an essential feature of any large scale engineering project, where they formalise the general assumptions that can be relied upon in each part of the design. It is the task of other parts of the design to validate the assumptions. In this way, responsibility for the correctness of the product can be shared among a large team of designers.

In this lecture, we replace the single-predicate model of system behaviour with a pair of predicates (P_0, P'); the first of these P_0 is a precondition, and the second P' is the behavioural predicate, known as the postcondition; it has exactly the same function that we have already described. We have seen the power of disjunction, conjunction and quantification in the construction of models of computing. We need to define analogues of these operations for the new predicate pairs.

Disjunction represents non-determinism; since we do not know what will happen, successful use of a non-deterministic product requires satisfaction of the preconditions of *both* alternatives:

$$(P_0, P') \vee (Q_0, Q') = (P_0 \ \& \ Q_0, P' \vee Q').$$

Conjunction is defined as the dual of disjunction

$$(P_0, P') \ \& \ (Q_0, Q') = (P_0 \vee Q_0, (P_0 \Rightarrow P_1) \ \& \ (Q_0 \Rightarrow Q')).$$

The post-condition states that the behaviour is described by one or both of the component behaviours, depending on the truth of the corresponding precondition. But only one of these preconditions needs to be true.

Successful use of parallelism requires satisfaction of the preconditions of *both* components. So we need to define a separate parallel operator:

$$(P_0, P')\|(Q_0, Q') = (P_0 \mathbin{\&} Q_0, P' \mathbin{\&} Q').$$

The hiding of a variable applies existential quantification both to the precondition and to the postcondition. Again, this operation has different properties from existential quantification, and needs a new symbol, say L:

$$Lx \mid: (P_0, P') = ((\exists\, x \mid: P_0), (\exists\, x \mid: P')).$$

Note that the behavioural part of all these definitions is exactly the same as in the simple case of a single predicate. Indeed, if all the preconditions in a formula are the same, (say *true*), then the two-predicate model is isomorphic with the single predicate.

In single-predicate models, universally quantified implication

$$[P \Rightarrow Q]$$

is used to signify that the product described by P is better than that described by Q, or at least as good. The reason is that the behaviour described by P is more predictable and more controllable than that of Q, and therefore useful for any purpose that Q may be used for. All the operators defined for an implementable theory are monotonic with respect to this ordering. This means that replacement of Q by P as a component of any assembly can only improve the quality of the assembly. For example,

$$\begin{array}{lll} \text{if } [P \Rightarrow Q] & \text{then} & [P \mathbin{\&} R \Rightarrow Q \mathbin{\&} R] \\ & \text{and} & [P \vee R \Rightarrow Q \vee R] \\ & \text{and} & [P;\ R \Rightarrow Q;\ R] \\ & \text{and} & \ldots \end{array}$$

In a two-predicate model, we introduce a new notation $P \sqsupseteq Q$ (or equivalently $Q \sqsubseteq P$) to stand for the improvement ordering, comparing a predicate pair $P = (P_0, P')$ with another $Q = (Q_0, Q')$. If P is to be an improvement on Q, its precondition must allow use of P in all circumstances allowed by Q, and maybe more, i.e.,

$$[Q_0 \Rightarrow P_0].$$

Furthermore, if the precondition for Q holds, then P's behaviour will be more predictable and more controllable than that of Q

$$[Q_0 \mathbin{\&} P' \Rightarrow Q'].$$

The new ordering $\sqsupseteq$ is defined as the conjunction of these two conditions

$$P \sqsupseteq Q \quad = \quad [Q_0 \Rightarrow P_0] \mathbin{\&} [Q_0 \mathbin{\&} P_0 \Rightarrow Q'].$$

This definition has the required properties of any improvement ordering:

it is a preorder, and all the operators defined so far are monotonic.

$$\begin{array}{lll} & P \sqsubseteq P & \\ \text{If} & P \sqsubseteq Q & \text{and} \quad Q \sqsubseteq R \quad \text{then} \quad P \sqsubseteq R. \\ \text{If} & P \sqsubseteq Q & \text{then} \quad (P \| R) \sqsubseteq (Q \| R) \\ & & \text{and} \quad (Lx \mid: P) \sqsubseteq (Lx \mid: Q) \\ & & \text{and} \quad \ldots \end{array}$$

In fact, $\sqsubseteq$ is a lattice ordering over pairs of predicates, with conjunction and disjunction as its meet and join.

$$\begin{array}{lll} R \sqsubseteq P \vee Q & \text{iff} & R \sqsubseteq P \text{ and } R \sqsubseteq Q \\ P \;\&\; Q \sqsubseteq R & \text{iff} & P \sqsubseteq R \text{ and } Q \sqsubseteq R. \end{array}$$

The bottom element is (*false*, *true*), which is always unusable, and the top element is (*true*, *false*) — clearly unimplementable.

Two descriptions may be defined as equivalent if each is always a valid replacement for the other:

$$P \equiv Q \text{ iff } P \sqsubseteq Q \text{ and } Q \sqsubseteq P.$$

Equivalent descriptions may be regarded as effectively equal. This formalises a previous informal claim that the description of the behaviour of the system is irrelevant when the precondition is not satisfied. As a result, we have general equivalences

$$(P_0, P') \equiv (P_0, P_0 \Rightarrow P')$$
$$(P_0, P') \equiv (P_0, P_0 \;\&\; P').$$

The introduction of a precondition into the description of a system is exactly the same as the introduction of a new Boolean variable into its alphabet. Let us name this variable "*ok*". An observation of its falsity means that the system has failed through violation of its precondition. So every description includes the fact that truth of the precondition ensures the truth of *ok*. It also ensures the postcondition. In summary, the pair notation can be defined simply as an abbreviation

$$(P_0, P') = (P_0 \Rightarrow (ok \;\&\; P'))$$

where *ok* does not occur in P_0 or P'. Now the equations defining conjunction, disjunction, and ordering among pairs can be proved as theorems about these single predicates.

Conversely, let $Q(ok)$ be any predicate. We can define its precondition and postcondition

$$Q_0 = \neg Q\,(\mathit{false}) \qquad \text{and} \qquad Q' = Q\,(\mathit{true}).$$

It follows that

$$(Q_0, Q')_0 = Q_0 \quad \text{and} \quad (Q_0, Q')' = (Q_0 \Rightarrow Q').$$

Indeed, if $Q(ok)$ is monotonic in ok, the translation can be inverted:

$$Q(ok) = (Q_0 \Rightarrow ok \ \& \ Q'), \quad \text{if } [Q(\mathit{false}) \Rightarrow Q(\mathit{true})].$$

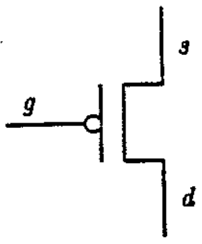
This establishes an isomorphism between predicate pairs and single predicates which are monotonic with respect to a single new Boolean variable in their alphabet.

7.1 Short circuit in a transistor network

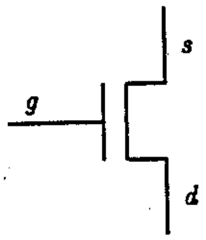
A short-circuit is a path of conducting transistors and wires which leads between the sources of High voltage and Low voltage. Short-circuits dissipate both heat and power, which can be injurious to the long term reliability of the whole network. These effects cannot be modelled in a theory as simple as ours. Designers are fortunately willing to accept an absolute obligation to avoid short-circuits. Such an obligation can often be coded in a simple theory by means of a precondition.

In the simple theory described so far, *Hw* means that the wire *w* is connected to High through a series of *P*-transistors with Low gates; and *Lw* means that it is connected to Low through a series of *N*-transistors with High gates. The restriction on the kind of transistor is needed for reliable use of the wire to control the gates of other transistors. Unfortunately a short-circuit can be conducted through an arbitrary interleaving of transistors of both types.

To control this phenomenon, we need to observe it, and record the observations in two new variables *WHw* and *WLw* associated wire *w*: they mean that the wire is "Weakly" High or Low. This is the result of connection to either source by either kind of transistor, as described by the preconditions of the transistors given below:

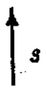


$WLg \Rightarrow (WHs \equiv WHd)\ \&\ (WLs \equiv WLd).$

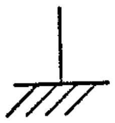


$WHg \Rightarrow (WLs \equiv WLd)\ \&\ (WHs \equiv WHd).$

The precondition for connection of a wire to High or Low is that it is not also connected to the other.

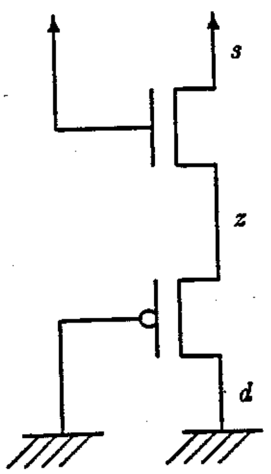


$WHs\ \&\ \neg WLs$



$WLd\ \&\ \neg WHs$

The precondition of a network is the conjunction of the preconditions of all its components. Consider the example:



$$WHs \mathbin{\&} \neg WLs$$

$$\mathbin{\&} (WLz \equiv WLs) \mathbin{\&} (WHs \equiv WHz)$$

$$\mathbin{\&} (WHz \equiv WHd) \mathbin{\&} (WLz \equiv WLd)$$

$$\mathbin{\&} (WLd \mathbin{\&} \neg WHd).$$

The precondition reduces to *false*, thereby revealing the short-circuit. The postcondition is just

$$\neg Hz \mathbin{\&} \neg Lz.$$

If z is hidden or left floating, this predicate does not clearly predict a short-circuit. Since the precondition does this job, the content of the postcondition is fortunately irrelevant.

The more realistic treatment of short circuits is a significant increase in the complexity of the model. Since strong connection implies weak, there are now six states for each wire:

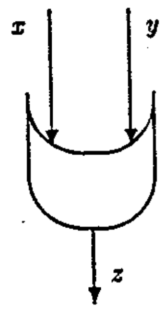
$WHw \mathbin{\&} WLw$	short-circuit
Hw	strongly High
$WHw \mathbin{\&} \neg Hw$	weakly High
$\neg WHw \mathbin{\&} \neg WLw$	floating
$WLw \mathbin{\&} \neg Lw$	weakly Low
Lw	strongly Low.

There are many other ways in which transistor network designs can go wrong. For example, there is an unavoidable resistance and delay to conduction of voltage through transistors and along wires. This could be modelled by introducing yet more states for each wire. An alternative practical expedient is just to count the number of transistors, connected source to drain, which separate an output wire from High or Low. If this never exceeds four, there is probably some way of sizing and positioning the transistors to make the whole network reliable and reasonably fast.

7.2 Cycles in combinational circuits

The correctness of a combinational circuit depends critically on the avoidance of cycles in the connections from input wires to output wires. Hitherto, we have assumed that a check against this error could be made by a syntactic inspection of the form of the equations or of the diagram picturing the connections. But the same check can be incorporated in the precondition for the circuit, after introduction of variables to indicate roughly the relative depth of each wire within the circuit.

If a connection graph is free of cycles, it is possible to ascribe to each wire w a numeric value Tw, in such a way that the output wire of any gate has a value strictly greater than the values of all its input wires. For example, Tw may stand for the time (measured from the start of the cycle) at which the wire reaches its final value, and remains stable thereafter. Now the precondition of a simple combinational gate simply describes this constraint upon its input and output wires, for example:



$$Tx < Tz \ \& \ Ty < Tz$$

The precondition of the whole circuit is just the conjunction of the preconditions for all of its gates. If this contradicts the basic property that $<$ is a transitive irreflexive relation, the precondition reduces to false, indicating that the circuit is in all circumstances totally useless.

The main reason for this encoding of the acyclic property of circuits is to permit the easy localisation of wire w by just existential quantification over the variable Tw. The variable can then be eliminated by the laws:

$$(\exists\, Tw \mid: Tv < Tw \ \& \ Tw < Tx) = (Tv < Tx)$$
$$(\exists\, Tw \mid: Tw < Tw) = \textit{false}, \text{ etc.}$$

7.3 Undefined expressions in assignments

In sequential programming language, consider the assignment

$$x := 1/0.$$

The expression on the right hand side is meaningless, and the effect of its evaluation by computer may be either non-termination or an interrupt, or even perhaps termination with x taking an arbitrary value. None of these effects can be described in terms of a model at the high level of abstraction at which we prefer to work. The complexities must be avoided by observance of an explicit precondition associated with each assignment, namely that the expression is defined; for example, the assignment $x := f(x,y)$ has precondition $((x,y) \in \text{dom}.f)$.

The same technique deals effectively with infinite looping or recursion. The precondition of the loop

$$\textbf{while } x \neq 3 \textbf{ do } x := x$$

is obviously that the initial value of x must be three. A general theory of recursion which achieves this effect is beyond the scope of these lectures.

7.4 Inconsistency in functional programs

Consider the functional program

$$\begin{array}{ll} f(x) = 0 & \text{if } x \leq 3 \\ f(x) = 1 & \text{if } x \geq 3. \end{array}$$

These two clauses are universally quantified over x. Consequently, their conjunction is actually *false*: there is no function f that satisfies both of them (what value would it give to $f.3$?). To guard against this paradox, for each function name f, we must introduce another variable Ef. This stands for a subset of the domain of the function which needs to be allocated to a given clause of the function definition; any attempt to define the function outside its allocated domain is an error.

The precondition of the clause

$$f(x) = \ldots \text{ if } b$$

is that the allocated domain of f is sufficiently large to include the whole of b; that is expressed by the predicate

$$(b \Rightarrow Ef).$$

When two clauses of a functional program are put together, it is essential that their allocated domains do not overlap. All values of x which are allocated to one clause must not be allocated to the other. Consider the parallel composition of two functions with preconditions $f1(Ef)$ and $f2(Ef)$. The resulting precondition is given by

$$f1(Ef)\ \&\ f2(\mathit{false}) \vee f1(\mathit{false})\ \&\ f2(Ef).$$

As an example

$$(b1 \Rightarrow Ef) \| (b2 \Rightarrow Ef) = (b1 \vee b2 \Rightarrow Ef)\ \&\ (\neg b1 \vee \neg b2).$$

This correctly allocates to the pair of clauses the union of their individual allocations; it also states that these allocations must not overlap.

Many functional languages take a less strict view about overlapping of the conditions on the individual clauses. Clearly, it can be allowed if the two clauses ascribe the same values to the function throughout the area of overlap. Some languages are even more generous, and specify that in case of conflict, the first clause will dominate. The construction of models for these languages is an interesting exercise.

7.5 Non-deterministic deadlock in communicating processes

Consider two processes with alphabet $\{a, b\}$

$$P = a \rightarrow P$$
$$Q = b \rightarrow Q.$$

These have behavioural descriptions:

$$\mathrm{P} = (\mathit{trace}.\mathrm{b} = 0)$$
$$\mathrm{Q} = (\mathit{trace}.\mathrm{a} = 0).$$

Their parallel composition has behaviour defined by the conjunction of these descriptions, which is equivalent to

$$\mathit{trace} = <> .$$

The calculations show correctly that the pair of processes will deadlock from the beginning, and never engage in any observable action.

Now consider a process that is non-deterministic, and may behave either like P or like Q:

$$(P \vee Q) = (\mathit{trace}.b = 0 \vee \mathit{trace}.a = 0).$$

When this is run in parallel with Q, the conjunction of the specifications indicates that the behaviour will be the same as that of Q

$$(P \vee Q)\ \&\ Q = (P\ \&\ Q) \vee Q = Q.$$

All possibility of deadlock has disappeared! The clause $(P\ \&\ Q)$ in the phrase $(P\ \&\ Q \vee Q)$ appears to allow an implementation to deadlock; yet this choice has been ruled out, by adding the alternative Q. This is quite wrong. Disjunction with a non-deterministic alternative should make the behaviour worse, not better.

The standard method of rescuing a scientific theory that is falsified by particular experiments is to postulate the existence of some neglected factor, whose observation or measurement would explain the discrepant phenomena. In the case of Communicating Processes, this factor has to record all the circumstances in which a process may deadlock. Deadlock occurs when the environment of the process is prepared to engage in one or more events in the alphabet of the process, but the process itself refuses to do so. We therefore need a new variable, called *ref* for *refusal*, in the alphabet of every process: its value is the set of events refused by a process in some stable state.

The descriptions of the two processes are now extended to include a description of what they are not allowed to refuse:

$$\begin{aligned} P &= (\mathit{trace}.b = 0 \;\&\; a \bar{\in} \mathit{ref}) \\ Q &= (\mathit{trace}.a = 0 \;\&\; b \bar{\in} \mathit{ref}). \end{aligned}$$

The definition of parallel composition of two processes with the same alphabet states that an event is refused if either of them refuse it

$$P(\mathit{ref}) \| Q(\mathit{ref}) = (\exists\, r, s | \mathit{ref} = r \cup s : P(r) \;\&\; Q(s)).$$

In our example, we still get

$$P \| Q = (\mathit{trace} = <> \;\&\; \mathit{ref} \subseteq \{a, b\}).$$

The possibility that this process stably refuses both a and b simultaneously is an observable phenomenon ($\mathit{ref} = \{a, b\}$), and does not disappear when further non-determinism is added:

$$(P \vee Q) \| Q = ((P \| Q) \vee Q) = (\mathit{trace}.a = 0 \;\&\; (\mathit{trace}.b \neq 0 \Rightarrow b \bar{\in} \mathit{ref})).$$

This illustrates our general account of the solution of scientific problems by introduction of new "observables". Many of the most important concepts of physics have been introduced in this way — mass, energy, force, etc. But of course, postulation of hidden factors introduces extra complexity into a theory, as well as removing it one stage further from observable reality. Such innovations are rightly resisted by the community of scientists, until all alternatives have been explored and found even less attractive.

References

[1] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31:560–599, 1984.

[2] E.W.Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[3] C.A.R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[4] C.A.R. Hoare. The varieties of programming language. In *Proc TAPSOFT*, 351 in LNCS, pages 1–18. Springer, 1989.

[5] C.A.R. Hoare. Conjunction and concurrency. PARBASE 90, 1990.

[6] C.A.R. Hoare. A theory for derivation of combinational C-mos circuit designs. *Theoretical Computer Science*, (90):209–251, 1991.

[7] C.A.R. Hoare. *Algebra and Models*, series *F* in Program Design Calculi, volume 118, pages 161–195. Springer, 1993.

[8] C.A.R. Hoare. Programs are predicates. *ICOT Journal* (38):2–15, 1993.

[9] C.A.R. Hoare and M.J.C. Gordon. *Partial correctness of C-mos switching circuits: an exercise in applied logic*, in Proc. 3rd Ann. Symp. on Logic in Computer Science, Edinburgh 28–36, 1988.

[10] C.A.R. Hoare, He Jifeng, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, (30):701–739, 1993.

[11] C.A.R. Hoare and I. Page. Hardware and software: the closing gap. Accepted for publication in *Transputer Communications*, 1994.