July 1993

MODELS AND ALGEBRA

C.A.R. Hoare
Computing Laboratory
University of Oxford
11 Keble Road
Oxford
OX1 3QD

Summary

Science makes progress by constructing mathematical models, deducing their observable consequences, and testing them by experiment. Successful theoretical models are later taken as the basis for engineering methods and codes of practice for design of reliable and useful products. Models can play a similar central role in the progress and practical application of Computing Science.

A model of a computational paradigm starts with choice of a carrier set of potential direct or indirect observations that can be made of a computational process. A particular process is modelled as the subset of observations to which it can give rise. Process composition is modelled by relating observations of a composite process to those of its components. Indirect observations play an essential role in such compositions. Algebraic properties of the composition operators are derived with the aid of the simple theory of sets and relations. Feasibility is checked by a mapping from a more operational model.

A model constructed as a family of sets is easily adapted as a calculus of design for total correctness. A specification is given by an arbitrary set containing all observations permitted in the required product. It should be expressed as clearly as posssible with the aid of the full power of mathematics and logic. A product meets a specification if its potential observations form a subset of its permitted observations. This principle requires that all envisaged failure modes of a product are modelled, as indirect observations, so that their avoidance can be proved. Specifications of components can be composed mathematically by the same operators as the components themselves. This permits top-down proof of correctness of designs even before their implementation begins. Algebraic properties and reasoning are helpful throughout development. Non-determinism is seen as no problem, but rather as a part of the solution.

1. Introduction

A scientific theory is formalised as a mathematical model of reality, from which can be deduced or calculated the observable properties and behaviour of a well-defined class of processes in the physical world. It is the task of theoretical scientists to develop a wide range of plausible but competing theories; experimental scientists will then refute or confirm the theories by observation and experiment. The engineer then applies a confirmed theory in the reverse direction: the starting point is a specification of the observable properties and behaviour of some system that does not yet exist in

the physical world; and the goal is to design and implement a product which can be predicted by the theory to exhibit the specified properties. Mathematical methods of calculation and proof are used throughout the design task.

This paper suggests a similar fruitful division of labour between theoretical and experimental Computing Science, leading to eventual application in the engineering of computer software and hardware. Theoretical computing scientists develop a wide range of plausible theories covering a variety of computational paradigms. The mathematical consequences of each theory are explored, and also relationships with other competing or complementary theories. The experimental computing scientist can then select some combination of related theories as the basis for the design of a software system or language, or an architecture for a computing device. The efficiency and effectiveness of this design is then tested by simulation or experimental implementation and application in representative case studies. Reliable use of the system or device or language will be further assisted by mathematical theorems, methods and heuristics derived from the original theoretical model.

Computing Science is primarily concerned with discrete phenomena; and therefore cannot take advantage of the large body of established knowledge of continuous mathematics, developed by applied mathematicians for the enormous benefit of physical science and engineering. It is rather pure mathematics that supplies the concepts, notations, methods, theorems and proofs that are most relevant for computing. But in contrast to pure mathematics, potential relevance to computing is taken as a goal and a guide in the selection of directions for our research. Its achievement depends on a good general (but informal) understanding of practical Computing Science. This should cover

1. a range of problems which may be solved by application of some computing device, and the terminology in which they are described.

2. the methods by which solutions to complex problems can be found by decomposition into simpler subproblems, so that these can then be solved by similar or even simpler methods.

3. the methods by which complex systems can be constructed by connecting subassemblies and components implemented in a similar or lower level technology.

4. the comparative cost and efficiency of alternative methods of design and implementation in hardware or in software.

Understanding of this wide range of topics should relate not just to the current computing scene (for which any new theory will come too late) but to some possible future evolution of it. As in all branches of physical science, success depends on a large element of intuition, insight, guess-work and just plain luck. That is why the community of theoretical scientists must be prepared to develop a large number of alternative theories, most of which will never achieve experimental confirmation or find practical application. This apparent profligacy is justified by simple economics. It is less expensive and risky to invent and develop ten new theories than it is to make even a prototype implementation of just one of them. And it is less risky and less

onerous to design and implement ten prototypes than it is to invest in development of a new market for a genuinely innovative product. And not all new products that come to market will remain there. So theories are as numerous as the seeds scattered by the winds; only very few will settle and germinate and take root and reach maturity and propagate more seed to populate the forests of the future.

So let us postulate the wisdom or courage to select some general line of enquiry for a new theory. In its detailed development, the researcher would be well advised to lay aside all hope of future relevance, and adopt the attitude of pure mathematicians, engaged in the pursuit of truth wherever their curiosity may lead them. Avoid competitive promotion of one line of enquiry against another. Otherwise you lose the spirit of dispassionate scientific objectivity, so necessary for the health of science. Whenever choice arises between directions of pursuit, choose first the path of greater simplicity and of greater elegance. And that should be your second and third choice too, and especially your last one. An elegant theory will attract attention of other theorists, and a simple one will attract the interest of teachers and students. This is the only way to reduce the risk that the theory will be forgotten before the time is ripe for its development and practical application. And finally, when the theory achieves widespread use or standardisation, the quality of elegance is the only hope we have of rescue from the quagmire of arbitrary complexity which is so pervasive, particularly in software of the present day. And elegance is a property that is needed not only for a mathematical model but also for the theorems and algebraic laws derivable from it.

It is the purpose of this paper to encourage the development of new and simple theories with the aid of set-theoretic models. Such models permit easy derivation of algebraic laws, which in turn assist in derivation of efficient solutions to practical problems. Models also readily support a general method for deriving designs from specifications by top-down decomposition; in crossing levels of abstraction, we exploit a helpful correspondence between generality of specification at higher levels, and non-determinism at lower levels of design and implementation.

2. Observations

The first task of the theoretician is to decide on what kind of system to explore, and to characterise which of its properties are to be regarded as observable or controllable or otherwise relevant to the description and understanding of system behaviour. For each property, an appropriate name is chosen: for example in a mechanical assembly the name x may denote the distance of a joint along one axis, and x may denote its velocity. In mechanics, the observed values vary continuously with time, and they are often called measurements. In computing, observations usually yield discrete values, and they are made only at discrete points of time. For example, in the case of a fragment of program, x may denote the initial value of an integer variable before execution starts, and x' might denote its final value on termination. The fact that these observations are not continuous measurements in no way detracts from the mathematical and scientific quality of the theories which describe them.

Once the relevant observations have been named, the behaviour and properties of a general system or a particular one can be described or specified by mathematical formulae, equations, inequations or other predicates which contain these names as

free variables. Each predicate describes those systems in which the observed values of all its variables make the predicate true. In science, a general class of system is often described by differential equations; and a specific member of the class by adding particular boundary conditions. For example, the predicate

$$x < k\dot{x}$$

describes the behaviour of any joint which moves sufficiently slowly in the vicinity of the origin of the x axis. Similarly in the case of programs,

$$x' > x$$

describes the behaviour of any piece of code which does not decrease the value of x. Such predicates may serve either as scientific predictions about the behaviour of known systems, or as engineering specifications of systems yet to be designed and implemented.

Sometimes the validity of a prediction R depends on validity of some other condition P. This condition usually mentions variables whose values can be controlled by the experimenter, or the user, or in general by the environment within which the described system is embedded; and so it is often called a precondition. If the environment fails to make P true, no prediction at all can be made of the behaviour of the system; and, in the case of a specification, no constraint whatsoever is placed on the design of the product.

Observations which are described in user specifications are usually those which can be made directly, as it were with the naked eye. But in a mature branch of science the most important observations are those which can be made only indirectly by some more or less elaborate experiment. An experiment involves connection of its subject in some well understood manner with other processes whose behaviour is well understood, so that a more direct observation can be made of the behaviour of the combined system. Very often, the presumed understanding of the experimental apparatus itself depends on the very theory that is being tested. There is clearly a danger of circularity, a risk which attends research in all branches of science. But a successful choice of the right kind of indirect observation (for example, energy in physics) can provide a remarkably coherent and general explanation of wide range of diverse phenomena. Such an indirect observation is often accepted as if it were a direct observation in some theory at a lower level of detail.

A familiar example of the distinction between direct and indirect observations can be drawn from models of sequential and parallel programming. For sequential programming it is adequate to make direct observations of the values of variables before and after execution; and specification of a complete program can be formulated entirely in terms of initial and final values. But suppose in the interests of efficiency we wish to implement the specification with the aid of component programs executing concurrently in a multiprocessor with a single shared store. These processes can interact during execution in ways that cannot be understood in terms of direct observations before and after execution. So the presumed observations of each process must include complete sequences (called trajectories) of state pairs, representing changes

due to atomic actions of that process. These sequences are in practice not directly observable, except by running the process concurrently with some other process designed to test it. Indirect observations are usually more complex and abstruse (and even controversial) than direct observations, and they are not intended to appear in the user's specification of the completed product; but they are vital to the engineering soundness of the design, because they permit accurate specifications of interfaces and components that may then be designed and implemented separately by separate teams of engineers at separate times.

In a theory intended for engineering design, it is also important to include among the potential indirect observations all the possible ways in which a physical implementation may break or fail. It is only in a theory which includes such failures that it is possible to prove that a particular design or product will avoid them. Since all kinds of failure are to be avoided, there is no need to make fine distinctions between them, or to give accurate predictions of the behaviour after failure. For the same reason, there is no need to mention avoidance of failure explicitly in a user specification of a complete product. Let us describe all such universally undesirable observations by a predicate named FAIL.

A familiar example of failure in software is non-termination of a sequential program due to infinite iteration or recursion. This can be represented by introducing a special variable "terminated", which is true when the program has terminated, and remains false if it never terminates. It is understood that the final values of the variables $x', y', \ldots$ are observable only when "terminated" is true; this understanding can be coded in the mathematical theory by allowing these variables to take arbitrary values when "terminated" is false. A specification never needs to talk about termination: one can take for granted that it is desirable. But implementations need to avoid it. So the first step in moving from specification to design notation is to introduce this extra variable.

Of course, in practice we can never wait the infinite time required to make an observation of a false value of the variable "terminated". This leads to philosophical objections against introducing a value which is so essentially unobservable; but they are the same kind of objection that can be made to zero as a number or empty as a set. Projective geometers never expect to observe their line at infinity, but their theory would not work without it. And in our case, the explicit introduction of non-termination and similar failures gives a similar advantage: it enables us to deal automatically with failure to meet "liveness" conditions in the same simple way as we deal with "safety" properties. To deal with "fairness" conditions one must accept an even greater variety of indirect observations, which would take an infinite time to observe.

The danger of failing to introduce enough indirect observations is that a product designed with the aid of the theory will break in some way unpredictable by the theory; so the theory establishes only conditional or partial correctness. A bad example is the Brock-Ackerman anomaly in non-interleaving models of non-deterministic data flow.

The converse danger of introducing too many indirect observations is that the theory will be more complicated than necessary. It is possible to prove that this danger has been avoided by showing that the theory is fully abstract in the sense of Milner and Plotkin. But this proof can be done only after the theory has been more

fully developed.

In summary, a theory intended for engineering design works with observations at two (or more) levels of abstraction. The direct observations are those which are described in a user specification S, and in a precondition P, placing constraints on the method and circumstances of use. The indirect observations are those mentioned in a description D of the actual behaviour of a delivered product, and in the description FAIL of all the undesirable ways in which a product may fail if the precondition is violated. The fact that the product meets its specification in now encapsulated in a single mathematical theorem

$$D \Rightarrow (P \Rightarrow \neg FAIL \wedge S).$$

This means that if the precondition P is satisfied, then every observation of the behaviour of the delivered product D will be a non-failing observation, and will also satisfy the specification S.

The last important message of this section is that an engineer never just delivers a product, but rather a product together with its specification, including operating instructions and preconditions for safe and successful use. Clarity and precision of specification are included among the most important qualities of a product; and mathematics provides excellent assistance in achieving them. Failure to realise this in software specification is notorious; and leads to many of the other problems encountered in current software engineering practice.

3. Implementable processes.

The implication displayed at the end of the last section formalises a proof obligation which may be discharged after the design is complete. But it is far better to regard it as a mathematical statement of the designers task, namely to find some design D which satisfies the implication and so meets the specification. Eventually, the whole design D must be expressed wholly within some limited set of design notations, which are known to be directly implementable in the available technology. The task is in principle no different from that of solving any other engineering problem which has been precisely formulated in mathematics.

An essential quality of the solution of a mathematical problem (for example a differential equation) is that it is expressed in more restricted notations than those used to formulate the original problem; otherwise the solution could be just a trivial restatement of the problem. It is the notational restriction that makes the solution useful. In Computing Science, such notations can be designed to be translated automatically for direct implementation either in hardware or in the machine code of a computer. And, as in mathematics, it is very important that the notation of the solution should just be a subset of the notations used for specification. So the theorist must undertake to select from the class of all specifications those which are more or less directly implementable.

In the previous section we have assumed that specifications are written as mathematical predicates with free variables standing for observable values in a fashion generally understood by the educated professional. This is a style preferred by practicing engineers and scientists, who tend to manipulate, differentiate or integrate the

text of formulae rather than abstract functions: it is also the style adopted by the Z school of specification, and in the specification-oriented semantics of programming languages. Pure mathematicians, on the other hand, tend to prefer closed mathematical abstractions like sets and functions and (more occasionally) relations. This is evident in the study of Analysis and even more in Topology. It is the style preferred in the denotational semantics of programming languages. Each style is more appropriate for the use to which it is put, and there is no conflict between them. Every predicate can be identified with a set, namely the set of those assignments of values to its free variables which makes the predicate true. And the sets and functions of the pure mathematician can and should be translated into predicates and formulae before use by engineers and programmers. The important relation of set inclusion then translates to logical implication, defining precisely the designer's proof obligation. In the remainder of this talk, it is more convenient to adopt the style of pure mathematics, dealing with sets and relations rather than variables and predicates.

Let us give the name OBS to the set containing mathematical representatives for all possible direct and indirect observations of all possible processes of interest. We can now represent a particular process P as that subset of OBS which contains all observations which could in any circumstances be made of that process. The set of all such processes, implementable in a particular envisaged language or technology, constitutes a family of subsets of OBS, to which we give the name of PROC. So the first two components of our model are similar to those of a topology — a carrier set OBS and a particular family PROC of its subsets. It is already possible to formulate interesting questions about the family, for example does it contain the empty set, or the universal set OBS itself? Is it closed with respect to union or intersection? We will later give reasons for answering these questions by no, yes, yes, and no.

The family PROC may be defined by describing the mathematical properties of each of its members. These commonly take the form of closure conditions, which force the sets to be "large enough".

1. Sometimes the reason for such conditions are obvious. If a process contains an observation "it weighs less than g" then it must also contain all observations that "it weighs less than $g + h$", for all positive h.

2. Sometimes the motive is one of realism: a conventional program can be started in any initial state whatsoever; so for all s it must contain a state pair with s as its initial component — an important condition which we will call totality. The possibility of nontermination therefore has to be represented by a special "bottom" value of the final state, usually written $\perp$. This represents falsity of the "terminated" condition, described in the previous section.

3. Sometimes a condition arises from some intended property of the operations provided in the implementation. In a constraint language, the states of the machine are predicates, and no operation is provided to weaken the state. So the sequence of predicates in any trajectory must form a strengthening chain.

4. Sometimes the goal is to permit efficient implementation. For example, in concurrent execution of a process, it is more efficient for a processor to execute many

consecutive transitions from one process while some other process is stationary. The validity of this implementation method can be assured by the closure condition: for any trajectory containing a pair of consecutive transitions $(s,t),(t,u)$, there is also a possible trajectory similar to the first, except that this pair is replaced by the single atomic transition (s,u), going straight from the first to the final state.

5. Sometimes the aim is to avoid making unnecessary distinctions. For example, a process should never be started in an initial state for which it has non-termination as one of its outcomes. So we may not care what other possible outcomes it may have in this case. Our deliberate indifference is formalised by a closure condition that whenever $-$ is a possible final state, so is every other state whatsoever. This is just a set-theoretic statement of the fact that before termination the final value of the state is unobservable and therefore arbitrary.

6. And finally, the motive may be simply mathematical convenience, to avoid reasoning about special cases, or to satisfy some desirable algebraic law. But of course this should not detract from realism or range of applicability of the theory; so concessions to convenience should usually be confined to circumstances involving failure; since failure is going to be avoided anyway, the details of its treatment in the model are more open to arbitrary decision.

A closure condition can often be expressed in terms of a function or relation which maps members of the set to other members. A set S is closed with respect to a relation c if it contains its own image through c

$$cS \subseteq S$$

$$\text{where } cS =_{df} \{y | \exists x. x \in S \wedge x(c)y\}.$$

For example, condition (1) states closure with respect to the ordering relation on weights; and (4) and (5) can also be expressed as relational closures. Closure conditions expressed in this way are easier to treat mathematically.

The conditions defining membership of PROC are intended to ensure physical implementability. Like the laws of physics, they describe general properties such as conservation of energy that must be preserved in any physical system. So it is not surprising that their discovery and formalisation is the first and most serious difficulty in the construction of realistic models; what is worse, their sufficiency and validity can be established only at the very last step in the evaluation of the model by practical use. That is why Dana Scott once characterised formalisation as an experimental science. So when the experiment succeeds, when all aspects of the theory link together harmoniously, then great satisfaction can be derived from the achievement, in addition to the possibility of more practical benefits.

The sets in PROC are intended to represent exactly the implementable processes of the theory. But a specification of such a process does not have to be a member of PROC. Any other subset S, defined by any desired combination of mathematical predicates, can serve as a specification of requirements placed on a particular member

P from PROC, which is yet to be designed. The design will be correct if and only if the eventually delivered P is a subset of S, i.e., all possible observations of the process (including even the undesirable ones) are permitted by the specification. So the subset relation between a process and a specification captures exactly the concept of satisfaction, as described in the previous section. Of course, it may be that there does not exist any P in PROC which satisfies the specification. It is then logically impossible to meet the specification within the given technology. The theory may help the engineer in avoiding the danger of promising to deliver such a product.

Consider a specification T, and let S be subset of T. Then S is a stronger specification than T: it places more constraints on the product and may therefore be more difficult to implement. Indeed, because set inclusion is transitive, every product that meets specification S will serve also as an implementation of T, so implementation of T cannot possibly be more difficult than S.

The subset relation may also be used to define an ordering among the members of PROC. By transitivity of inclusion, $P \subseteq Q$ means that P satisfies every specification satisfied by Q, and maybe more. Consequently for all relevant purposes and in all relevant respects P is better than Q (or at least as good). Thus if Q is a simple design or prototype which clearly meets its specification, then Q can be validly transformed to (or replaced by) P, without jeopardising correctness; the motive for doing so may be a reduction in cost or increase in efficiency. One of the main objectives of a mathematical theory is to provide a comprehensive collection of such correctness-preserving, but efficiency-increasing transformations. Notice that the interpretation of the relation $\subseteq$ as "better than" depends on the fact that OBS contains all relevant ways in which a process may fail. It is this that ensures that the better process is the one that fails less often; and furthermore, because it gives rise to fewer non-failing observations, it is easier to predict and control what it is going to do. In this way "better" also implies "more deterministic".

We can now single out from PROC those processes which are the best of their kind, in the sense that none of them can be further improved. This subfamily will be called DET, because it contains those processes which are as deterministic and as free from failure as possible. For each process D in DET there exists a specification (namely D itself) which is met by D and by no other process

$$P \subseteq D \Rightarrow P = D,$$

$$\text{for all } P \in \text{ PROC, and all } D \in \text{ DET.}$$

The size of DET is therefore indicative of the range of solutions provided by the theory, and therefore of the range of essentially distinct problems that can be solved by it. So a theory in which DET has only a few members is not likely to be widely applicable.

It is unusual for a general theory to include cost or speed among is observables, because these factors are highly variable between one project and another. However if they are included in a more specific theory, it is important to ensure that the observations take the form "it costs less than n" or "it goes faster than m". Then $P \subseteq Q$ means that Q can cost more and go slower, so the interpretation of inclusion as a merit ordering can be maintained. But such a theory can deal only with uniform

improvement with respect to all criteria simultaneously; it becomes wholly inapplicable in the more frequent case when one criterion must be traded against the other. That is another reason why these considerations are usually omitted from a general theory, and left to the good judgement of the engineer. No amount of mathematical theorising can ever replace that!

4. Some useless processes.

Certain individual members of PROC can be simply defined as sets of observations; and the simplest example would be the empty set of observations. But there are two devastating arguments against including the empty set in PROC. Firstly the philosophical one: it is wholly unrealistic to design and deliver a product which could never give rise to any observation whatsoever, either direct or indirect. Secondly the practical objection: the empty set would by definition satisfy every specification expressible in the theory. It would be the only member of DET, and even if there were other members of PROC, there would never be any need to use them. The empty set would be a miracle or panacea, and a mathematical theory which contains it can only be applied to a problem domain in which a panacea exists. But in such a domain, there is hardly any need for a mathematical theory. For this reason, it is essential to introduce enough indirect observations to ensure that no process is represented by the empty set.

Another easily defined process is the universal set OBS itself. This is called ABORT in Dijkstra's sequential programming language, and CHAOS in CSP. It is the easiest of all processes to implement — in fact any process whatsoever will serve as an implementation. But it is the worst possible process to use; its behaviour is maximally uncontrollable and unpredictable, and it may go wrong in any or all possible ways. It is difficult to imagine a computational device that really behaves as badly as this; but perhaps one example would be a program which on receipt of an external interrupt executes a wild jump to a floating point number. But a true understanding of CHAOS comes from a recognition that a specification is an integral part of the delivered product. If somehow the specification becomes detached, say from a bottle of medicine, then the only safe thing to do with it is to throw it away unused. So also must one treat a product, say processed food, which has been stored in a manner which violates the stated preconditions for safe consumption. These are the most useless of products; and they are modelled in our theory by the weakest of all processes, namely the universal set OBS. These are good enough reasons for including OBS as a process in PROC; since OBS satisfies all possible closure conditions, it is mathematically convenient to do so. And since the responsible engineer should do anything to avoid such a dreadful process, mathematical convenience is a sufficient excuse for including it in the theory.

The following examples are more useful than OBS, but more specific to a particular computational paradigm. In a sequential programming language there is usually a process that terminates immediately without making any change to its initial state. It is called SKIP in Dijkstra's language and CONTINUE in FORTRAN. If the states before and after execution are observed, they will be found to be the same. So SKIP is modelled as the identity relation

$$\texttt{SKIP} =_{df} \{(s,t) | s = t\}.$$

In a constraint language, the states of the machine are represented as predicates. Such a language provides a method for strengthening the current state predicate by addition of some proposition b The process is named "affirm (b)", and its observations are those in which the final state is a conjunction of b with the initial state

$$\{(s,t)|t = s \wedge b\}.$$

We can already prove the algebraic law

$$\text{affirm(true)} = \texttt{SKIP}.$$

In a programming language that allows concurrent computation, it is usual to provide some means for synchronisation, whereby one process can wait until other concurrent processes bring the machine state into a condition satisfying some desired predicate c. This is achieved by a process which we will call "wait(c)". In a conventional procedural language, a state s satisfies c when evaluation of c in state s yields true. In a constraint language, satisfaction means that the state s logically implies the truth of c. If c is satisfied in the initial state, the wait has no effect, so observations of "wait (c)" will include

$$\{(s,s)|s \text{ satisfies } c\}.$$

In the more interesting case when c is not satisfied by the initial state, the "wait(c)" process simply waits. We therefore need to introduce a special "wait state" denoted by the symbol ?. The pair $(s,?)$ means that the process is observed to be waiting in state s; it corresponds to a refusal in CSP. The definition of "wait(c)" is

$$\{(s,s)|s \text{ satisfies } c\} \cup \{(s,?)|s \text{ does not satisfy } c\}.$$

If s does not satisfy c, then all states weaker than s also fail to satisfy c. So we have another closure condition for processes

$$\text{If } (s,?) \in P \text{ then } (s \vee t,?) \in P.$$

Since every predicate satisfies the predicate "true" we have the simple algebraic law

$$\text{wait(true)} = \texttt{SKIP}.$$

The wait state is useful even in a non-parallel language: Dijkstra's concept of a guarded command can be simply modelled:

$$b \rightarrow P =_{df} \text{ wait}(b); P.$$

A process which waits for false to be satisfied is likely to wait forever. Such a phenomenon is known as deadlock, and is denoted by STOP in CSP or NIL in CCS

$$\begin{aligned} \text{STOP} &=_{df} \text{ wait(false)} \\ &= \{(s,t)|t =?\}. \end{aligned}$$

The behaviour of STOP is certainly highly undesirable, almost certainly the result of a design error or violation of a precondition. It is therefore tempting to make no

distinction between STOP and the worst possible process OBS. Experience shows that it is wise to delay giving way to this temptation. Later we may wish to define a combinator that permits recovery from deadlock, but which cannot recover from other kinds of failure like non-termination. Premature identification of STOP with OBS would prevent introduction of this useful combinator.

The processes affirm and wait are both in DET. They are both deterministic (functional) because each initial state is paired with exactly one final state. Any process with less final states than that would violate the totality condition.

5. Boolean combinations of processes.

The single processes like those defined in the previous section are too simple to solve a real problem, even of the most trivial kind. They can only serve as primitive components, which need to be connected together and used in combination to exhibit more complex and useful behaviour. In order to prove in advance that such a combination will work as intended, we need to formalise by mathematical definition the various ways in which components and subassemblies can be combined into larger assemblies or can be adapted for new purposes. These combinators are usually denoted by mathematical operators, whose operands are written representations of the processes which they combine. The combinators are selected and designed to ensure that they map implementable processes to implementable ones; furthermore that they themselves are implementable by some kind of interconnection of components described by their operands. In this section we will introduce some combinators which can be defined with the aid of simple boolean operators on sets.

Let P be a process and let S be some suitable subset of OBS, such that $P \cap S$ (P restricted to S) is also a process. Then $P \cap S$ is like P, except that its capability of giving rise to observations outside S has been removed. For example suppose P has a two-position switch, and S contains only observations taken when the switch is off. Then $(P \cap S)$ describes an object in which the switch (or at least the capability of turning it on) has been removed. This kind of restriction is used in CCS and ACP to prevent external observation or interference with interactions occurring on internal communicating links in a network. It is immediately obvious that restriction distributes through set union. So do all the other combinators defined in the remainder of this paper.

In a concurrent program it is sometimes desirable to force the execution of a sequence of transitions as if it were only a single atomic transition, uninterrupted by any concurrent process. Let S be the set of all atomic transitions, i.e. all trajectories of length one. Then $(P \cap S)$ eliminates from P all transitions that are not atomic. We have assumed here that P already satisfies the closure condition that consecutive transitions of a process are permitted to be amalgamated; in $(P \cap S)$ such amalgamation is forced.

Conjunction might seem to be a useful operator for removing mistakes from a program, for example if S is the set of all observations that do not involve non-termination. Unfortunately the conjunction $(P \cap S)$ will usually fail the closure conditions (like totality), except for very special sets S. Or perhaps this is fortunate, because in practice such a process would be impossible, or impossibly inefficient, to implement. For the

same reason, a modeller should always take great care not to exclude accidentally the ways in which a combination of processes can in practice go wrong.

If P and Q are processes represented by sets, then their set union $(P \cup Q)$ is the strongest specification satisfied by both P and Q

$$(P \cup Q) \subseteq S \text{ iff } (P \subseteq S \text{ and } Q \subseteq S), \text{ for all } S \subseteq OBS.$$

If the closure conditions for PROC are expressed in terms of a relational image, $P \cup Q$ will be just the set union of the observations of P and of Q. This is extremely convenient, and explains why the standard semantics of CSP is given in terms of refusal sets, which are closed with respect to the subset relation, rather than acceptance sets, whose closure condition is not relationally definable.

This specification $(P \cup Q)$ presents no great difficulty of implementation. For example it can be implemented either by P or by Q, whichever is the easier; though the conscientious engineer will use his judgement to choose the one that is most cost-effective for the eventual user of the product. $(P \cup Q)$ can be thought of as a product delivered with two different operating instructions, one for P and one for Q, and no indication which is right. Of course it may be possible to decide between them by subsequent experiment; but until then the only safe thing is to use the product in a manner consistent with both its manuals. Or maybe there is a single manual, which is ambiguous, and does not state whether it will behave like P or like Q. Sometimes the supplier has good reason for the ambiguity — think of a restaurant which has fresh vegetable soup on its menu. Similarly, a designer may wish to keep options open for later decision, or a manufacturer may wish to retain freedom to deliver later variations of a product. In any case, the formula describing $P \cup Q$ may be simpler, more abstract, and easier to manipulate than either of the separate formulae for P or Q. So it is not unreasonable to regard $(P \cup Q)$ as a member of PROC, provided it satisfies the relevant mathematical closure conditions. And we do not have to decide the vexed question whether it is allowable (or even possible) to construct a genuinely non-deterministic process, whose behaviour is not determined even at time of delivery. We will return to this point in a later section.

If it is possible to take the union of two processes P and Q, what about their intersection $(P \cap Q)$? Such a process would engage only in observations on which both P and Q agree. If this is a member of PROC, it is better than both P and Q, although it is the worst process with this property. What is more, intersection can be very useful in meeting a specification expressed as a conjunction of a requirement S with a requirement T. For example suppose S requires that the final value of an array must be sorted in ascending order, and T requires that the final value must be a permutation of the initial value. With the aid of non-determinism it is easy to write a program P which assigns to the array an arbitrary ascending sequence of numbers, thereby meeting requirement S. Requirement T can be met similarly by a program Q that assigns to the array an arbitrary permutation of its initial value. Elementary propositional logic now guarantees that the intersection $(P \cap Q)$ will meet the conjunction $(S \cap T)$ of the requirements.

Unfortunately, in a conventional sequential language, the implementation of $P \cap Q$ would involve repeated execution of both P and Q, looking for an outcome possible

for both of them. And if there were no such outcome, it would be impossible to implement, because even non-termination would be disallowed. This impossibility is reflected in the theory, which insists that all members of PROC are total. Intersection is such a useful and common combinator for specifications that it is highly desirable to explore special cases in which a conjunction of specifications can be efficiently implemented by some combination of processes, each of which meets only part of the specification. That is the main driving force behind research into non-conventional and non-sequential programming languages, as well as modular structures for more conventional ones.

If union is easy to implement but intersection infeasible, let us explore some further process combinators that lie between these two extremes. Let S be some suitable subset of OBS. Then the conditional process $P\langle S\rangle Q$ (P if S else Q) is defined to exhibit an observation of P just when that is an observation in S; but each observation from outside S is an observation of Q:

$$P\langle S\rangle Q =_{df} (P \cap S) \cup (\neg S \cap Q).$$

Simple Boolean algebra gives

$$P \cap Q \subseteq P\langle S\rangle Q \subseteq P \cup Q$$

In the case of a conventional sequential language, the set S usually takes the form of a computable test b to be applied to the initial state, independent of the final state:

$$B =_{df} \{(s,t) | s \text{ satisfies } b\},$$

so its complement also takes the same form:

$$\neg B = \{(s,t) | s \text{ does not satisfy } b\}.$$

This means that $P\langle B\rangle Q$ can be executed by a test on the initial state, before choosing between execution of P or of Q. Such a combinator is included in all general-purpose procedural programming languages. But for other choices of S, it is very unlikely that the result will be a process. For example consider the relational converse of B, which tests the final state rather than the initial state. Such a conditional will not in general satisfy the totality condition; and even when it does, it could hardly be implemented without systematic backtracking.

The algebraic properties of $\langle S\rangle$, considered as an infix operator, are easily derivable by Boolean algebra. It is idempotent and associative; furthermore, it distributes through union and union distributes through it. Finally, $\langle S\rangle$ distributes through $\langle T\rangle$ for all S and T. Boolean algebra provides an extraordinary variety of mutually distributive operators: indeed, it seems that any operator which makes a selection between its operands, involving execution of exactly one of them, will distribute through itself and every other operator of the same kind.

I have encountered some resistance among algebraists to the idea of mutually distributive operators. To help overcome this prejudice, here is a whole new class of them. Let S be some suitable subset of OBS. Then the process $(P[S]Q)$ is defined to exhibit

an observation in S just when both P and Q agree to do so; but each observation outside S may be either from P or from Q

$$P[S]Q =_{df} (P \cap Q)\langle S\rangle(P \cup Q).$$

This is called the negmajority, because it gives the majority vote of P and Q and the negation of S. Simple Boolean Algebra again gives

$$P \cap Q \subseteq P[S]Q \subseteq P \cup Q.$$

A useful special case is when S is STOP, i.e. the set $\{(s,t)|t =?\}$. So P[STOP]Q will wait only in those states in which both P and Q are waiting; if the initial state is a wait state for P but not for Q, then the next and all subsequent transitions of P[STOP]Q will be determined by Q and P will be ignored; and symmetrically, interchanging P with Q. In a state in which both P and Q can proceed, either of them may be selected for execution, thereby giving rise to non-determinism. In CCS this operator is written $+$ and in CSP it is $[]$; it is called (external) choice because it allows the environment a degree of choice between its operands. Because the relation STOP is a constant function, [STOP] preserves the totality condition of its operands.

By Boolean algebra $[S]$ clearly shares all algebraic properties common to both union and intersection (which are actually just the special cases $[OBS]$ and $[\{\}]$ respectively). Furthermore, $[S]$ has S itself as unit; this provides a method for averting deadlock in CCS or CSP by giving a better alternative

$$\text{STOP}[]P = P.$$

Finally, $[S]$ is highly distributive. Indeed, each of the operators $[S], [T], \langle U\rangle$ and $\langle V\rangle$ distribute through all of them (including itself).

Associativity is a useful property of $[]$; for example, it justifies writing Dijkstra's guarded command set without brackets

$$b \to P[]c \to Q[]d \to R.$$

The definition of $[]$ and $\to$ ensures that the whole set deadlocks if and only if all the guards are false. In Dijkstra's language, the deadlock is converted to abortion by surrounding the guarded command by **if** ... **fi**, which can be defined

$$\textbf{if } P \textbf{ fi} =_{df} P \cup \{(s,t)|(s,?) \in P\}.$$

As our last Boolean combinator, let us consider negation or complementation. Suppose we wish to define a process which behaves like P, except that it can never do anything that Q can do. Such a process could be very useful as a safety-critical control program. Let Q be a process whose observations include all erroneous or dangerous ones; thus $(P - Q)$ is guaranteed to be safe. Unfortunately, there is no general way to implement such a complementation operator: and mathematically it is very unlikely to satisfy the closure conditions which define processes. If physical and mathematical impossibility are not strong enough arguments, we will later find yet another reason for not admitting complementation as a combinator on processes. But of course, like

all the Boolean operators, it remains extremely useful for specification.

6. Relational combinations of processes

We now shift attention to relations between observations of two or more processes; such relations are effectively subsets of the cartesian product space $OBS \times OBS$. If r is such a relation and P is a process, we define rP as the relational image of P through r. Each observation of rP is related by r to some observation of P

$$rP =_{df} \{y | \exists x. x \in P \wedge (x, y) \in r\}.$$

It follows that r, considered as an operator on PROC, distributes through union

$$r(P \cup Q) = (rP) \cup (rQ).$$

A simple example of a relation is the identity relation on some subset S of observations:

$$\{(x, x) | x \in S\}$$

The image of P through this is nothing but the familiar restriction operator $(P \cap S)$.

The relation r must of course be selected so that its image has the same desired closure properties as its operand. If the closure conditions are expressed in terms of a relation c, it is sufficient to prove

$$r; c \subseteq c; r,$$

where ; denotes relational composition. It follows that rP satisfies the closure condition:

$$c(rP) \subseteq r(cP) \subseteq r(P), \text{ provided } cP \subseteq P.$$

Another example of an image is the operation in CCS which changes the names of the events in which a process engages. In CSP, this is represented by a function f from the alphabet of the operand to the alphabet of the result, which is applied to each event in each trace. If f is a bijection, fP is structurally identical (isomorphic) to P; but if it maps several different events onto the same one, it introduces an element of non-determinism: when fP engages in an event fe, it is not known which of the events in $f^{-1}(fe)$ actually took place; and so its future behaviour will be more difficult to predict and control. There is no problem in generalising f to an arbitrary relation, at the expense of even greater non-determinism.

In a concurrent programming language, observations are sequences of events. If e is an event and s is a sequence, the sequence $\langle e \rangle^\frown s$ is defined as one that starts with e and continues with s. The prefixing relation (actually a total injection) is defined simply as

$$(e \rightarrow) =_{df} \{(s, t) | t = \langle e \rangle^\frown s\}$$

The image of a process P through this relation is a process which first engages in the event e and then behaves like P. It is denoted $(e.P)$ in CCS and $(e \rightarrow P)$ in CSP; though in CSP some additional observations (refusals) are needed to preserve closure conditions.

Now suppose a process P has already engaged in an initial event e, and we wish to predict what its future behaviour will be. This is known as "P after e" (P/e) in

CSP, or the derivative of P by b in the terminology of regular languages. Each of the traces of P/e must be such that restoration of e by prefixing will give back a trace of the original process P; so "after" is just the relational converse of prefixing. Since prefixing is a total injection, we obtain immediately

$$(e \rightarrow P)/e = P.$$

However, if the after operator is applied to a process which could not have started with an occurrence of e, the result will be empty, which we have decided should not be considered as a process. So "after" is only a partial operator on executable processes, which is why it does not feature in CCS or other process algebras. However, there is no reason why the operator cannot be used in specifications and in reasoning about the design of processes.

Now suppose that a known process P is known to have engaged in not more than one event, but it is not known what the event was, or even whether it has happened. Then the future behaviour of P is just the union of P with its image through the truncation operator, defined as the removal of the first event from any non-empty sequence. This is a projection operator, which conceals some part of the behaviour of a process; it therefore tends to introduce or increase non-determinism. Think of what is known of the contents of a box of assorted chocolates, after their hiding place has been discovered by children, who may have eaten some of them. A similar projection or hiding operator operation is used in CSP to prevent observation or participation by the environment in communications along internal channels of a network. In an actual implementation the events do actually occur, but they are unknowable and uncontrollable from outside. In an actual machine, advantage is taken of this non-determinism by allowing the events to occur as fast as they possibly can.

To define combinations of two or more operands, we need relations between three or more observations: for simplicity we shall confine attention to just binary combinations requiring only ternary relations:

$$r \subseteq ((OBS \times OBS) \times OBS).$$

Wherever possible, more complex combinators (with three or more arguments) should be defined in terms of simpler binary combinators, preferably associative ones; my treatment of Dijkstra's guarded command shows the way.

The relational image will be written by an infix notation

$$(PrQ) =_{df} r(P \times Q)$$

$$= \{z | \exists x, y . x \in P \wedge y \in Q \wedge ((x, y), z) \in r\}$$

The simplest examples are provided by the familiar Boolean operations. $(P \cup Q)$ is just the image of the relation

$$\{((x, y), z) | z = x \vee z = y\},$$

and $(P \cap Q)$ is the image of

$$\{((x, y), z) | x = y = z\}.$$

Relational definitions of $\langle S \rangle$ and $[T]$ are hardly more complicated. Of course, complementation cannot be defined as a relational image; but there are good reasons for excluding this anyway as a combinator for processes.

In a sequential programming language, each program is itself a relation, containing pairs (s,t) of initial and corresponding final states. Sequential composition of processes is defined as the image of the following relation between these pairs:

$$\{(((s,t),(t,u)),(s,u))|s,t,u \text{ are states}\}.$$

In fact, this yields the familiar composition of relations, which is easily implemented as sequential composition in a conventional programming language: the final state of the first process is taken as the initial state of the second process. This intermediate state is then concealed

$$P;Q =_{df} \{(s,u)|\exists t.(s,t) \in P \wedge (t,u) \in Q\}.$$

As with other concealment operators, interests of efficient implementation require explicit representation (and certainly not the concealment) of all the ways in which a combination can go wrong, either as a result of failure of the components or some mismatch between them. For example, what happens in sequential composition if the final state of the first operand is not in the domain of the second? Avoidance of this problem is the main motive for the closure condition that all processes must be total relations. Similarly, if the first operand of composition fails to terminate, the mathematics must ensure that the whole composition will fail. This can be achieved by an additional closure condition, namely that $(\bot,\bot)$ and $(?,?)$ should be observations of every process, thereby ensuring that the possibility of non-termination of the first operand is preserved by sequential composition. Of course, this is something of an artificial coding trick; but this does not matter, since $-$ is anyway an artificial state, which is going to be deliberately avoided in practice. An alternative solution is to complicate the definition of composition to treat non-termination as a special case. Choice between such alternatives is one that must often be made by a modeller, and often requires detailed exploration of both of them. In the end, mathematical convenience may be the only deciding factor.

In a constraint language, sequential composition $P;Q$ involves appending each trajectory of Q to every trajectory of P, except those ending in ?, which remain unchanged. No intermediate states are hidden; but any trajectory that violates the strengthening chain condition is removed. Parallel execution of two processes can be modelled by a form of interleaving of complete traces from each operand, subject to the strengthening chain condition, and the terminality of ?. In addition, any pair of transitions $(s,t),(s,t')$ of the operands may be replaced by the single resulting transition $(s,t \wedge t')$. Similarly, two final wait states (s,?) and $(s',?)$ should be merged to $(s \cup s',?)$.

When all the implementable operators on processes have been defined, it is at last possible to check that the design of the whole theory is fully abstract. Full abstraction means that the difference between any two processes P and Q, represented by different sets in PROC, may be detected by a direct observation of some experiment to which

they are both subjected. The experiment is conducted by connecting each process into an environment C, consisting wholly of implementable operators of the theory, giving $C(P)$ and $C(Q)$ respectively. The experiment is designed to ensure that the direct observations of $C(P)$ differ from those of $C(Q)$. If such an experiment is always possible, then the theory is said to be fully abstract; and this shows that the particular choices of indirect observation have not introduced any unnecessary complexity.

The characteristic feature of computational processes is that they involve many more steps than could ever be described explicitly in a program or other written representation of a design. This problem is solved by repeated use of parts of a program in some form of iteration, or more generally by recursion. We describe a simple but general form of recursion without parameters. Let X be a variable standing for an arbitrary subset of OBS. Let $F(X)$ be an expression of the programming language built up from X (and perhaps other process variables and explicitly defined processes) by means of combinators of the language. Consider the equation

$$X = F(X)$$

which states that X is a fixed point of F. Now subsets of OBS form a complete lattice under inclusion ordering, and F, being defined solely by relational images, is a monotonic function. A famous theorem by Knaster and Tarski proves existence of a solution to the equation.

In fact, there is a complete lattice of solutions: which one do we want? Since we want to be able to implement the solution, we want the easiest one, namely the greatest of them. Such a solution always exists as a specification; but in a general purpose programming language we would like it also to exist as a process. A general way of achieving this is due to Scott and Smyth.

1. Allow OBS to be a process.

2. Suppose $\{X_i | i \in N\}$ is any descending chain of processes. Ensure that $\cap_i X_i$ is also a process (in particular, it must not be empty). This usually requires exclusion of infinite non-determinism.

3. Ensure that all combinators are continuous in the sense that $F(\cap_i X_i) = \cap_i (FX_i)$ for all descending chains. This is guaranteed if all combinators are defined in terms of relations that are finitary, in the sense that the inverse image of any finite set is also finite (or universal). Sometimes a non-finitary relation is allowable, as in the case of hiding in CSP.

Now the fixed point of F is just

$$\cap_i F^i(OBS),$$

where F^i is the ith iterate of F. This can be readily computed by unfolding the definition of F as many times as required. If no finite unfolding is adequate, the implementation will fail to terminate; but this is exactly what the theory also predicts. The existence of this simple general way of defining iterations or recursions is a great

simplification of the task of the modeller, who can concentrate attention on the mathematical properties of finite processes, i.e., those that are defined without recursion. Another equally valid general method of explaining recursion is by metric spaces and contraction mappings: these always give an unique fixed point.

7. Calculus of design.

The previous sections have shown how to give a mathematical model of various ways in which processes may be combined into larger assemblies. The theory may be used predictively, as in science, to calculate the observable properties of a system whose components have known properties. But engineers have to work in the reverse direction. A specification is a description of the desired properties of an assembly that does not yet exist. The engineering task is to design and implement the components, and assemble them in a manner which meets the specification. For this we need a calculus of design. The calculus is based on the idea that a combinator defined on processes $P, Q, \ldots$ can be equally well applied to specifications, that is to arbitrary subsets $S, T, \ldots$ of OBS. This fact is heavily exploited (using predicates in the place of sets) by the schema calculus of Z.

Let us suppose that a designer is faced with a specification U. Judgement based on experience perhaps suggests an implementation in which (say) two components are connected by some combinator r. The further design and implementation of the components is to be delegated as separate tasks to two teams, or two persons, or even to one person working on the two tasks at separate times. Successful delegation requires careful formalisation of the correct specifications S and T of the two components. Their correctness should be proved before the first step of implementation, because detection and correction of design errors after delivery and assembly of the components may give rise to arbitrary unbounded expense and delay. The theorem that needs proof is formalised by using r to combine the specifications of S and T, showing this implies the original overall specification U:

$$SrT \subseteq U.$$

Because r is monotonic, any implementation of S, when combined by r with any implementation of T, will assuredly satisfy U. This method of rigorous decomposition can be applied equally well to the design of the subassemblies too, and can be repeated until each component can be implemented by a primitive process or assembly already known to work. It is particularly effective in Computing Science, where the combinators available for connecting large assemblies are logically indistinguishable from those available for much smaller subassemblies, right down to the level of primitive components.

This design method can be adapted to assist in reuse of subassemblies that have already been designed, for example as a module in a library. Suppose T is a specification of such a module, which is to be connected into a system with specification U by means of a combinator r. The designer has to find and implement the specification S of the rest of the system, in a manner which satisfies the inequation

$$SrT \subseteq U.$$

The method described above requires discovery and formalisation of S followed by proof. If the proof fails, the formalisation of S must be repeated — a frustration familiar to many an engineer who does indefinite integration by guessing a formula and checking its derivative. Far better to calculate the correct result directly and immediately from T and U. Provided r is defined as a relational image, this can be done by the formula

$$S = \cup\{X | XrT \subseteq U\},$$

or better, by some simpler formula which has been proved equivalent to it. This gives the weakest specification of any product X which has the required property, namely

$$XrT \subseteq U \text{ iff } X \subseteq S.$$

So the choice of this particular S as a specification for the rest of the system involves no additional design commitment or loss of generality. The technical term for this situation is a Galois connection, a simple case of a categorical adjunction. In general, let r be any relation. Then the weak inverse of r is defined as a relation s with the property that

$$XrT \subseteq U \text{ iff } X \subseteq UsT.$$

The weak inverse of sequential composition has been called the weakest prespecification; and in CSP the weak inverse of parallel composition has been called the weakest environment. But in practice weak inverses give rise to complications that belie the simplicity of the general theory:

1. S may in fact in unimplementable. In fact an early proof of this may save a lot of wasted effort, because it shows that either the choice of T or of r has been mistaken.

2. In practice, the size of the formulae derived by this method can be excessive. Nevertheless, the stepwise simplification of the formulae, with the aid of strengthening, may be a good guide to the further design decisions needed at this stage.

Further theoretical research to solve these problems is to be strongly recommended.

8. Non-determinism

The method of modelling a computational process as a set of observations is intended to deal in a uniform fashion with both deterministic and non-deterministic processes. It is possible to single out deterministic processes as a special case; it is possible to note informally when a combinator may fail to preserve determinism of its operands. But once non-determinism is accepted and taken for granted, there is no necessity to make these distinctions; and the mathematical theory develops most smoothly without them.

Many practicing engineers are very reluctant to accept non-determinism, and rightly so. The only way that they have been taught to assess reliability of a product is to test it. But a non-deterministic product may very well pass every test, yet later fail in practical use, just when most reliance is placed upon it. The only known solution to this problem lies in mathematical design methods that inhibit the intrusion of error.

Indeed, this is already coming to be accepted by some engineers as more effective than testing, even for determinate products. Hence the slogan "Design right - First time".

Many mathematical computing scientists are also reluctant to accept non-determinism, and rightly so. They have been educated in a tradition that mathematics is about functions, and that its concepts are expressed primarily in functional notation. The use of functional notation for non-deterministic operations leads to immediate confusion: for example, one has to question the validity of the absolutely fundamental equation of mathematics, namely

$$fx = fx.$$

Similar difficulties arise for partial functions; and the solution is the same: go back to the foundation offered by set theory, and use relational notations wherever they are appropriate. This is a solution which is already being applied by abstract functional programmers, of the schools of FP and squiggol; they have found it more effective to calculate with function composition rather than function application.

One important characteristic of our treatment of non-determinism is that it is not possible to specify that a delivered product must be non-deterministic at the time of delivery. Any satisfiable specification is satisfied by some member of DET. It is this that makes it possible to use the same mathematical model for non-determinism in products as for under-determination in designs and specifications; and full advantage is taken of this in the calculus of design.

Finally there is an interesting technical question, with strong philosophical overtones. Is it possible to make an observation of a delivered product that will reveal that the product was non-deterministic at time of delivery? In technical terms, could there be an observation of a non-deterministic process P which is not possible for any of the deterministic processes contained in P ? Or is each process P equal to the union of all deterministic processes contained in it

$$P = \cup\{D | D \in DET \wedge D \subseteq P\}?$$

One of the major differences between CCS and CSP is that in CSP conforms to this principle of invisibility of non-determinism.

Of course, it is always possible to detect non-determinism if one can observe the internal structure of the implementation. For example a process known to have the structure $(P; P)$ could be observed to be non-deterministic if each instance of P were observed to behave differently. In the theory of testing which underlies the equivalence of CCS, it is permitted at any time to take a copy of the current state of the process, and conduct the same (or different) tests on each copy; and it is this that may make non-determinism visible. In many applications, of course, such wholesale copying is physically impossible — for example if the system to be copied is the whole universe, or some large or inextricable part of it, like you or me. Even at the level of a single particle, quantum theory tells us that no copy can be made: otherwise it would be possible to test momentum of one copy and position of the other. It is really only in the abstract world of mathematics that copying is possible, and so easy that it can be taken for free.

But enough of philosophy: there are also interesting technical and practical considerations. For example, if non-determinism is unobservable, it is possible to prove

that a resource allocating process that chooses among free resources at the time of the request is just as good as one that gains efficiency by preselecting the next allocation at the time of the previous request. In CCS, these two processes can be distinguished, because one of them resolves its non-determinism later than the other; and this can be detected by cloning the whole allocator before the next request. But of course, a resource allocator is a prime example of a process that should never never be copied. So in this case at least there are good reasons for using the slightly more powerful proof methods available in CSP.

There is reason to suppose that non-determinism will come to play an increasing role in Computing Science, both practical and theoretical. In practice, continuing miniaturisation of circuits will continue to favour highly parallel hardware, and provide increasing incentives to use it efficiently. But if parallel processes are to cooperate on the solution of a single problem, possibly sharing mechanical resources such as disc storage, they will certainly need on occasion to synchronise with each other. Each synchronisation will in principle delay at least one of the processes involved. Examples of the most significant delays are those arising from paging faults in a virtual memory, or from scheduling of arithmetic units in a data flow architecture. Increase of processing speed can only increase the significance of these delays. The only solution is to allow the existence and duration of the delay to influence the course of the computation. For example the programmer can use the $[]$ combinator of CSP to allow the earliest possible selection of the first possible event that can occur. Since delays are essentially unpredictable, we are almost forced to accept non-determinism as an inherent property of programs and algorithms, and even of computer hardware. Learning how to cope with non-determinism is one of the most significant challenges and achievements of theoretical computing science, and still offers exciting challenges for the future.

9. Operational semantics.

The main danger in constructing mathematical theories about technological products is that they may in the end be unrealistic, impossible, or impossibly expensive to implement. The ultimate test of feasibility is widespread use by practicing engineers and programmers, supported by mechanical design tools and compilers. For example, theoretical investigations into communication and concurrency have been validated by efficient implementation of the occam programming language on the transputer, and by use of its algebraic laws in the design of the T800 floating point unit. But the installation of theory in practice requires an enormous investment and usually takes more than fifteen years. So it is duty of the theoretician to take every possible step to reduce the risk of unpleasant surprises at at later stage. Fortunately mathematics again provides methods of discharging this responsibility.

For a theory of hardware design, the justification for an abstract mathematical model, expressed say in Boolean algebra, is found by relating it to some more detailed model of implementation of the hardware components, say in terms of voltages on wires. And this too can be validated at an even lower level of abstraction by appeal to the relevant laws of physics. Only at this point, if there is any doubt, the mathematical theorist is entitled to hand over responsibility to the experimental scientist to confirm

the physical accuracy of the theory.

In the case of software designs and programming languages, the mathematician can discharge more of the responsibility. For example consider the risk that a theory of processes is in principle unimplementable; perhaps because one of the operators requires a test whether its operand will fail to terminate; or because it is more subtly incomputable in the sense of Turing and Church. This danger can be averted by giving the language a denotational semantics, expressed usually in a functional notation which is known to be Turing-computable. At one time this was the only known way of presenting an abstract formal semantics for a programming notation.

Apart from total incomputability, there is an equally serious danger, particularly for theories that include non-determinism: the language may contain an operator like conjunction whose implementation requires exploring all the possible non-deterministic behaviours of its operand, looking perhaps for one that terminates in some desirable state. Such an operator can indeed be implemented, but only by backtracking or similar technique, which introduces an exponential increase in time taken or resources consumed. The way to check against this risk is to construct a mathematical model of a step-by-step implementation, for example as a Petri net, or by formalisation in Plotkin's structured operational semantics. Both of these techniques ensure that that the permissible next steps in the evolution of a composite process can be determined by considering just the first possible step in the evolution of each of its components. Often at any given time only a small subset of the components need to be considered, which increases the possibilities for parallel execution of many steps, and reduces the need for synchronisation. But once feasibility has been checked by an operational model, operational reasoning should be immediately abandoned; it is essential that all subsequent reasoning, calculation and design should be conducted in each case at the highest possible level of abstraction.

10. Process Algebra.

The practical use of a model to assist in engineering design requires a significant use of mathematical reasoning of one kind or another. In principle, this reasoning can be based on the raw definitions of the operators involved; but the labour involved would be totally unacceptable — like solving partial differential equations by expanding the primitive definitions in terms of the epsilons and deltas of analysis. The only way that a branch of mathematics can be applied by engineers in practice is when it offers a range of useful theorems, symbolic manipulations and calculations, together with heuristics for their application. It is reasonable to expect a modeller to formulate and prove an initial collection of such theorems, because their proof may require changes in some aspect of the model, its operators, its processes, or even its observations.

For example, in a sequential programming language, it is reasonable to expect that sequential composition will be associative, and that it will have `SKIP` as its left and right unit (identity). These properties need proof, which can be given in terms of the definitions of the operators. If the definitions do not have these properites, it is the definitions that should usually be changed.

For example, it is reasonable to expect that wholly erroneous program `ABORT` cannot be reliably rescued by a program which will not start until the `ABORT` has successfully

terminated — because there is a strong possibility that it never will. This expectation can be most clearly expressed in a proposed algebraic law

$$\texttt{ABORT} ; P = \texttt{ABORT} \text{ , for all } P \in \text{PROC}.$$

However, this equation is false in the simple relational model of sequential composition, unless the image of P is total, like that of ABORT. One way to establish this is to insist that each process P contains the whole set $\{(s,t)|s = \perp\}$. But now it is necessary to check that all the other operators of the theory preserve this property. If not, they too may have to be changed ...

The easiest kind of theorem to use is one that is expressed as a general equation, which is true of all values of the variables it contains. An equation which is needed in a particular application can often be deduced by a process of calculation: starting at either side of the desired equation, a series of substitutions is made until the other side is reached; each substitution is justified by a known equation of theory. Each step is relatively easy to check, even with the assistance of a computer; and long sequences of steps may be carried out almost automatically by a term rewriting system. Such transformations are most frequently required to increase the efficiency of implementation by breaking down the more elaborate structure resulting from the top-down development. A very similar advantage can be taken of theorems (inequalities) using inclusion in place of equality, since all the operators involved in our theories are monotonic, and an engineer needs to exercise freedom to take design decisions which reduce non-determinism.

Of course, there is no limit to the number of theorems that may be derived from a model, and the mathematician needs good judgement in selecting the ones which are likely to be useful. Equally important, the chosen theorems should be reasonably memorable; for this, brevity and elegance are an important aid, as well as self-evidence to the operational understanding of engineers. Again, it is helpful if the theorems express familiar algebraic properties of combinators, for example, associativity, commutivity, idempotence, or distribution of one operator through another. In fact the best possible way of educating an engineer in a new computational paradigm is by an elegant collection of algebraic laws, together with examples and exercises combining theory with practice. This is the way in which pupils at school are taught to reason about the various kinds of number — integers, fractions, reals, complex numbers. The study of the sophisticated and widely differing models for these number systems is more the province of theoretical pure mathematics, and is a topic of specialist study in Universities.

An important goal in the derivation of algebraic properties of a model is to find enough laws to decide whether one finite process (defined without recursion) is equal to another, or below it in the relevant ordering. In some cases there is a decision procedure which applies the laws in a particular direction to eliminate the more complex operators, and produce a simple normal form. This procedure is applied to both sides of an equation or inequation, and a simple comparison is then made of the two normal forms. The symbolic calculations are easily mechanised by a term rewriting system, though in many cases the normal form (or some intermediate expression) is so much larger than the original formula that it may exhaust the available resources

of a machine, of at least the patience of its user. Except in the case of rather small finite universes, there is rarely any hope of an effective decision procedure for processes defined by recursion: in general, an inductive proof is necessary to reason about them.

The practical benefit of deriving laws strong enough for a decision procedure is that thenceforth it is known that all necessary equations (or inequations) can be proved from the laws alone, without expanding any of the definitions or even thinking about any of the observations. This is so valuable that it does not matter if the normal form contains notations which are not in fact implemented, or perhaps not even implementable. But even then the task of the mathematical modeller is far from over. In each particular application area for a computational paradigm, there are likely to be more specialised theorems, which can help in reliable use of the paradigm; and sometimes the theorems will be of more general utility, and so deserve a place in the central core of the theory. The constant illumination of practice by theory, and the constant enrichment of theory by practice over many years and centuries has led to the current maturity of modern mathematics and its applications in science and engineering; and it shows the direction of future advance for the comparatively immature discipline of theoretical Computing Science.

Algebraic laws have proved their value particularly in the design and implementation of general-purpose programming languages. They are most valuable in transforming a program from a structure which clearly mirrors that of its specification to one which most efficiently matches the architecture of the machine which will execute it. Such transformations may be carried out automatically, by an optimising compiler. Sometimes the motive is to transform a program into some smaller subset of a language, so that it may be implemented in some more restricted technology, for example by silicon compilation. Finally, algebraic transformations seem quite effective in verifying aspects of the design of the compiler itself.

The value of algebraic laws and equations is so great that there is a great temptation to avoid the laborious task of modelling, and simply to postulate them without proof. As Bertrand Russell has remarked: "The method of postulation has many advantages: they are the same as the advantages of theft over honest toil". In the case of a computational paradigm, the honest toil of linking algebra with an operational model is required to help implementation of the paradigm; and the link with a more abstract observational theory of specifications is essential for an effective calculus of design.

But of course, in spite of Russell's remark, the study of abstract algebra, independent of all its models and applications, has a most important role. A complete and attractive algebra can stimulate the search for applications and models to match it. A cramped and awkward algebra can give warnings about problems that are best avoided. When two models obey exactly the same complete set of algebraic laws, there is no need to choose between them; each can be used for the purpose it suits best. But the most important role of algebra is to organise our understanding of a range of different models, capturing clearly those properties which they share, and those which distinguish between them. The various number systems share many familiar algebraic properties — a useful fact that is totally concealed by the radical differences in the structure of their standard models. The variety of programming languages is subject to a similar algebraic classification.

My discussion of the relationship between models and algebra suggests future directions of research for pure algebraists.

1. An algebra usually starts with a collection of primitive constants and operators (the signature), in terms of which other useful concepts and notations can be defined.

2. The derived notations are often as useful as the primitive ones; and their algebraic properties should be explored with at least as much enthusiasm.

3. As in other branches of algebra, alternative but equivalent choices of primitive signature and axioms should also be explored.

4. Preference should be given to operators which preserve interesting properties of their operands. For example, the parallel combinator of CSP preserves determinism, and the more complex chaining operator preserves responsiveness; both of these properties help in avoiding deadlock.

5. Many useful laws can be expressed as inequations using some preorder representing improvement or implementation of specifications. All operators must be assumed monotonic in this order.

6. The most useful advances in pure and applied mathematics have been made by postulating inverses for more primitive operators, especially when this is counter to engineering intuition.

7. Where exact inverses are impossible or otherwise undesirable, weak inverses may be a very useful substitute, even if they are not directly implementable.

8. The existence of normal forms is a good test of the completeness and consistency of an algebra. There is no need for the normal form to consist wholly of implementable notations.

9. Limits and recursion can be introduced by standard techniques, using the same ordering suggested in (5), or some suitable complete metric space.

10. If possible, the algebra of implementable processes should be embedded into a complete Boolean algebra, which can serve as a specification language.

11. The eventual goal of research is the development of a family of related algebras, suited to a wide range of application areas and implementation technologies. The more powerful and expressive members of the family will be more useful for specification and design; and methods of symbolic calculation will be available to transform designs to more directly implementable notations.

But the most important message of this paper is one that we know already: that the interplay between models and algebra is constantly fruitful; and each of them provides guidance in taking rational decisions between otherwise arbitrary lines of development in the other.

11. Outlook

I have described the ways in which both models and algebras can contribute to solution of practical design problems in computing; and I have illustrated my points by examples which may have given the impression it is easy. This is not so. The construction of a single mathematical model obeying an elegant set of algebraic laws is a significant intellectual achievement; so is the formulation of a set of algebraic laws characterising an interesting and useful set of models.

But neither of these individual achievements is enough. We need to build up a large collection of models and algebras, covering a wide range of computational paradigms, appropriate for implementation either in hardware or in software, either of the present day or of some possible future. But even this is not enough. What is needed is a deep understanding of the relationships between all the different models and theories, and a sound judgement of the most appropriate area of application of each of them. Of particular importance are the methods by which one abstract theory may be embedded by translation or interpretation in another theory at a lower level of abstraction. In traditional mathematics, the relations between the various branches of the subject are well understood, and the division of the subject into its branches is based on the depth of this understanding. When the mathematics of computation is equally well understood, it is very unlikely that its branches will have the same labels that they have today. The investigations by various schools, now labelled as CSP, CCS, ACP, Petri Nets, etc., will have contributed to the understanding which leads to their own demise.

The establishment of a proper structure of branches and sub-branches is essential to the progress of science. Firstly, it is essential to the efficient education of a new generation of scientists, who will push forward the frontiers in new directions with new methods unimagined by those who taught them. Secondly, it enables individual scientists to select a narrow specialisation for intensive study in a manner which assists the work of other scientists in related branches, rather than just competing with them. It is only the small but complementary contributions made by many thousands of scientists that has led to the achievements of the established branches of modern science. But until the framework of complementarity is well understood, it is impossible to avoid gaps and duplication, and achieve rational collaboration in place of unscientific competition and strife.

The advantages to practical engineering are equally important. In most branches of engineering, product design involves mixture of a number of differing materials and technologies. Each separate technology must be well understood; but most of the difficulties and misunderstandings and unpleasant surprises occur at the interfaces between the technologies. And the same is true in computing, when attempting to put together a system from programs written, perhaps for a good reason, in different languages, with equipment of differing architectures, and perhaps increasingly in the future, with highly parallel application-specific integrated circuits. An appropriate theory can help in each individual aspect of the design; but only an understanding of the relationships between the theories, as branches of some more abstract theory, can help to solve the really pressing problems of overall system integration.

12. Conclusion.

It is possible and quite common to conduct valid and useful mathematical research in theoretical computing science, avoiding all consideration of mathematical models. It is possible to confine attention almost wholly to operational models like Petri nets. Or it is possible to start with a structured operational semantics, and investigate a range of equivalence relations based on various choices of bisimulation. Differing bisimulations give different collections of algebraic laws. These laws are then applied directly in case studies both to specification and design of useful algorithms and protocols. All these are valid specialisations, each with its own goals and methods of research. Their conceptual framework is simple and operationally intuitive, and it is attractive as well as advantageous to explore their range of usefulness to its limits and even well beyond.

But specialisation also has its dangers; and here I am worried that too much concentration on operational origins may inhibit, discourage, or delay the introduction, investigation and use of theories at higher levels of abstraction, closer to the user's problem. Surely it is only by taking advantage at all times of reasoning at the highest possible level of abstraction that we can master and control the incredible complexity of software, computers and communications devices of the present day. That is why my own preference is to start my investigations not with a particular algebra or computational paradigm but by exploring a class of related problems and the language in which they are most naturally expressed. The next task is to relate this to a conceptual framework and language in which a solution can be designed. Only after passing through several levels of abstraction is it necessary to consider the intricate detail of actual implementation.

If this general top-down method of constructing models is accepted as a useful complement to the bottom-up approach, then I can make a number of more detailed recommendations arising from my experience as a maker of models.

1. A model intended specification should describe only variables directly observable or controllable by the user.

2. A model intended for design should include enough indirect observations to permit the definition and accurate prediction of the behaviour of composite processes in terms of the behaviour of their components.

3. The indirect observations should include as far as possible the errors or failures of an implemented process, covering both safety and liveness conditions, and even fairness if desired.

4. Implementable processes should be defined by closure conditions, sufficient to ensure realism, avoidance of irrelevant distinctions, efficiency of implementation, and satisfaction of algebraic laws.

5. Combinators are defined to construct implementable processes from implementable components.

6. At all stages, the elegance of the model should be checked by proof of nice algebraic laws.

7. Combinators applied to specifications provide a calculus of design, with weak inverses to help in top-down development.

8. Non-determinism should be accepted, either as an inherent property of computations, or as a convenient (or at least harmless) mathematical fiction.

Each of these recommendations can be vigorously disputed, and considerable research can and should be conducted on the consequences of violating them. However, if the recommendations are accepted as a whole, they provide a coherent methodology for achieving one of the major goals of engineering research, namely the establishment of a link between theory and its practical application.

So let's make models. It's challenging, instructive and enjoyable; and it may even one day be useful.