Unifying Theories for Parallel Programming

Tony Hoare and Jifeng He

Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK

Abstract. The progress of science involves a constant interplay between diversification and unification. Diversification extends the boundaries of science to cover new and wider ranges of phenomena; successful unification reveals that a range of experimentally validated theories are no more than particular cases of some more general principle. The cycle continues when the general principle reveals further directions for experimental investigation. This paper suggests that the time has come to attempt a unifying classification of theories of parallel programming. Ideally, this should provide a common basis for reasoning about specifications and the correctness of designs, for optimising programs by algebraic transformation, and for implementing them in a range of technologies on a variety of machine architectures, to satisfy the needs of a wide range of applications.

1 Introduction

There is a lot of inherent diversity in the study of concurrency. There are major dichotomies between shared store and disjoint store approaches, between pairwise synchronisation and global lockstep synchronisation, between channelled and unchannelled communication, between isochronous and buffered input/output, and between hardware and software implementations. These are the variations suggested by the range of architectures available for implementation, and by the needs of applications which range from embedded real time systems through telecommunications to massive scientific simulations. The diversity of the phenomena gives scope for an equal diversity of the theories needed to explain and control them. Further dichotomies can be introduced by theorists, who adopt varying styles of presentation even for very similar theories, for example, by describing their operational, algebraic or denotational semantics.

But we believe that the time is ripe to make a start on a process of unification. A unifying theory is one that reveals the relationships among a family of subtheories, by capturing the essential properties that they share, and clarifying the equally essential features that distinguish them. Unification permits knowledge, skill, experience, languages and tool support to be transferred from one application to another, and helps in the design of systems to be implemented in a mixture of technologies, including even hardware. This ambitious task can be accomplished only by following the example of unifying theories in other branches of mathematics and natural science. It will have to take advantage of the discoveries and skills of many branches of theoretical computer science, especially those of programming language semantics. Denotational, algebraic and operational styles of presentation all have an important role, once their mutual consistency has been proved. We aim at nothing less than a comprehensive theory of programming, providing a common basis for specification, design, coding, implementation, compilation and optimisation of a variety of programming languages and paradigms. Only by unifying theories at a basic level will we consolidate our claim that computing science is an independent and coherent scientific discipline, worthy of study as an aid to the engineering of its applications.

We will illustrate the search for unification by selecting a few examples drawn from theories of parallel programming. We will concentrate on a definition of a family of parallel combinators $\|_M$, where the members of the family are determined by different choices of the single parameter M. We will show how many different theories can be generated by simply varying that parameter. All the variations share the same basic algebraic properties, and the properties that differentiate them are also expressed algebraically. In many cases, a complex theory for a composite programming language can be constructed as the sum of its simple parts; in other cases, account must be taken of unavoidable interaction effects. We hope to show that unification is an interesting study, even for those who do not wish to follow the necessary mathematical details.

2 Algebra

The goal of unification is one that has long been sought by mathematics; and it has been achieved most successfully by application of the techniques of modern algebra. Consider an analogy with the study of the foundations of arithmetic. This has revealed many different meanings for the arithmetic operation of addition, as applied in different kinds of number system — natural, integral, fractional, real, complex, quaternion, matrix, etc. The unifying properties shared by all these variations are algebraic: all forms of addition are associative and symmetric; and furthermore there is always a neutral element which is unchanged when added to itself

$$0 + 0 = 0$$
$$x + y = y + x$$
$$(x + y) + z = x + (y + z)$$

We will insist that any definition of parallelism shares exactly these properties of addition. We will use parallel bars to denote the combination of processes running concurrently: if P and Q are program fragments which describe the behaviour to two processes, $(P \| Q)$ describes the result of executing them in parallel. The neutral program $\mathbb{I}$ (skip) is one that does nothing and changes nothing. The traditional algebraic laws for addition are transcribed in this new application to programs as

$$\mathbb{I} \parallel \mathbb{I} = \mathbb{I}$$
$$P \parallel Q = Q \parallel P$$
$$(P \parallel Q) \parallel R = P \parallel (Q \parallel R)$$

Sequential composition of the components of a program is subject to a very similar set of algebraic laws. The program $(P;\, Q)$ describes the execution of P followed by that of Q, where Q does not start until P has terminated successfully. This operator obeys stronger laws for the neutral element $\mathbb{I}$, but it is obviously not symmetric

$$\mathbb{I}; P = P = P; \mathbb{I}$$
$$(P; Q); R = P; (Q; R)$$

The meaning of sequential composition is the same in every sequential programming language, and it will surely satisfy these laws, independent of the details of notation or the complexity of other features included in the language.

The non-determinism often associated with parallelism is a well-known source of problems, which can also be tackled with the aid of algebra. Let $(P \vee Q)$ describe the possible range of behaviours of a program that may behave like P or it may behave like Q, though in advance we do not know (and cannot control) which of them it will be. The basic algebraic properties expected of this operator are similar to those of any other operator that offers choice between alternatives: idempotence, symmetry and associativity

$$P \vee P = P$$
$$P \vee Q = Q \vee P$$
$$(P \vee Q) \vee R = P \vee (Q \vee R)$$

One great advantage of the modularity of algebra is that it permits the properties of different operators to be captured independently of each other, as we have shown for parallelism, sequentiality and non-determinism. But the interactions between the operators are just as important; these too are expressed algebraically, most often by simple distribution laws like those that hold between multiplication and addition in arithmetic. For example, both parallel and sequential composition distribute through non-determinism

$$(P \vee Q) \parallel R = (P \parallel R) \vee (Q \parallel R)$$
$$(P \vee Q); R = (P; R) \vee (Q; R)$$
$$R; (P \vee Q) = (R; P) \vee (R; Q)$$

These laws help to reason about non-determinism by considering each case separately. There are also useful laws which relate parallel with sequential composition; they are more distinctive to particular theories, and a range of alternatives will be described in later sections.

3 The theory of programming

The practical value of algebraic laws can only derive from their application to particular branches of engineering, science or even mathematics itself. Each application gives its own interpretation to the variables P, Q, R appearing in the equations. The main practical value of a theory of programming is to assist in the design and implementation of programs that are correct, in the sense that they meet their specifications. We therefore allow our variables to range not just over computer programs, but also over system designs and user specifications, which are ideally formulated long before the program is written.

Thus we unify the study of programs, designs, and specifications, regarding them all as different kinds of *predicate* that describe the behaviour of a computer system in relationship to its environment and its users. When a predicate serves as a specification, it describes the full range of the desired and permitted behaviour of the system. In the interests of clarity, a specification may be expressed with the aid of any relevant abstractions and notations of mathematics. When a predicate is used as a program, it describes the full range of possible behaviour of a computer that is executing the program. In order to permit automatic compilation and execution of such a predicate, it has to be expressed wholly in a very restricted notational framework, namely a programming language. The formal semantics of the language defines the meaning of each notation by translating it into the corresponding predicate. Since both specifications and programs are predicates expressed in different notations, it is natural to allow intermediate designs to be expressed as any helpful mixture of programming and mathematical notations.

The strongest reason for interpreting programs, designs and specifications within the uniform mathematical space of predicates is an extremely simple treatment of the elusive concept of program correctness. A program is correct just if it logically implies its specification

$$\text{program} \Rightarrow \text{specification}$$

This means that any behaviour of the actual program when executed is a behaviour described by (and therefore permitted by) the specification. The traditional engineering technique of stepwise design is justified by the transitivity of implication, as expressed by the inference rule, known as *cut*

$$\frac{\text{program} \Rightarrow \text{design} \qquad \text{design} \Rightarrow \text{specification}}{\text{program} \Rightarrow \text{specification}}$$

Repeated application of this rule permits an implementation to be split into many steps. The design formalises the interface between each step. The successive designs are expressed in notations getting closer to the target programming language. The last step is the pure program, whose correctness has been guaranteed by this stepwise method of construction.

Many familiar engineering principles can be described as elementary proof rules in logic, and they can be used directly in reasoning about the correctness of programs. For example non-determinism can be simply equated with the propositional concept of disjunction! A sufficient (and necessary) condition for the correctness of a non-deterministic program $(P \vee Q)$ is that both alternatives should be correct, as shown by the familiar rule for proof by cases

$$\frac{\text{prog1} \Rightarrow \text{design} \qquad \text{prog2} \Rightarrow \text{design}}{(\text{prog1} \vee \text{prog2}) \Rightarrow \text{design}}$$

The predicate calculus introduces free variables, bound variables, and quantification. Free variables are used in programming theory in the same way as in natural science, to describe phenomena that are more or less directly observable in the real world. Each observation of the system under study yields a measurement which is taken as the value of the appropriately named variable occurring (free) in the predicate that describes the system. Theoretical and experimental scientists have to agree on the association between the name of each variable and the timing and method for making the measurement that will determine its value. A scientific theory or prediction expressed as a predicate is valid if it is always true whenever its free variables are replaced by the values observed in the course of a properly conducted experiment. The set of free variables relevant for a given scientific theory may be called its *alphabet*; and different theories have different alphabets describing different kinds of experiment.

In the theory of programming, the alphabet is determined by the names of the global variables, say $x, y, \ldots, z$, which are accessible and updatable by a given program. As in other branches of science, the first and most important occasion for making an observation is when the experiment starts, or when the program is set to run. We indicate such initial observations by placing a left arrow over the name of the global variable: $\overleftarrow{x}, \overleftarrow{y}$ are the initial values of the global variables x and y; conversely, a right arrow on $\overrightarrow{x}, \overrightarrow{z}$ indicates the final values observed on termination of the program. To begin with, we will assume that the start and the finish are the only two occasions when observations can be made. The *alphabet* of a program P is denoted by αP, and it is split into initial and final subalphabets, similarly distinguished by arrows.

$$\alpha P = \overleftarrow{\alpha} P \cup \overrightarrow{\alpha} P$$

In programs, the alphabet is usually determined by the context of declarations which surround a program rather than by the variables that appear in its text. Certain conventions ensure satisfaction of various constraints that we shall place on the definition of programming notations. But this is an issue that we will largely ignore.

4 Sequential and Parallel Programs

The basic constituent of every program in our theory is the assignment statement, for example

$$x := x + y$$

When executed, this ensures that the final value of the variable on the left of the assignment symbol is equal to the value of the expression on the right, where all the variables in the expression take their initial values. A mathematical statement of this fact is taken as the *definition* of the assignment, for example

$$\begin{aligned} x := x + y \;&=_{df}\; \overrightarrow{x} = \overleftarrow{x} + \overleftarrow{y} \\ \alpha(x := x + y) \;&=\; \{\overrightarrow{x}, \overleftarrow{x}, \overleftarrow{y}\} \end{aligned}$$

For simplicity, we will assume that evaluation of expressions always terminates.

Larger programs are built from the basic assignments by combinations expressed in the notations of the programming language. Consider for example, the conditional **if** b **then** P **else** Q. Given that P and Q are predicates, the semantics of the language must define the predicate corresponding to this combination. Let $\overleftarrow{b}$ be the initial value of the boolean expression b, obtained by distributing the left arrow to all its variables. If b is true initially, the behaviour of the program is described by P, and if false, the behaviour is described by Q; and one of these two cases must hold. These facts are expressed simply in the propositional calculus by the definition

$$P \lhd b \rhd Q \;=_{df}\; (\overleftarrow{b} \wedge P) \vee (\neg \overleftarrow{b} \wedge Q) \qquad \text{provided } \alpha P = \alpha Q$$

The algebraic properties of the conditional are easily proved as mere tautologies. The infix notation gives a familiar shape to the laws — idempotence, symmetry, associativity, etc.

$$\begin{aligned} P \lhd b \rhd P &= P \\ P \lhd b \rhd Q &= Q \lhd \neg b \rhd P \\ (P \lhd b \rhd Q) \lhd b \rhd R &= P \lhd b \rhd (Q \lhd b \rhd R) \\ &= P \lhd b \rhd R \end{aligned}$$

The definition of sequential composition of programs is based on that of composition in the relational calculus. In executing $(P;\, Q)$, the final value of each global variable of P is taken as the initial value of the variable of Q that shares the same name. But this transmission of values is never observed; all that is known is that each value exists (or at least that it could have existed) at the time that control passes from P to Q. In the predicate calculus, such an unknown value is concealed by existential quantification over the global variables of the intermediate state. These are then removed from the alphabet of the result.

$$\begin{aligned} P(\ldots \overrightarrow{y}, \overrightarrow{z});\, Q(\overleftarrow{y}, \overleftarrow{z}, \ldots) \;&=\; \exists\, y, z, \ldots P(\ldots, y, z) \wedge Q(y, z, \ldots) \\ \alpha(P;\, Q) \;&=\; \alpha P \cup \alpha Q \;-\; \overrightarrow{\alpha} P \cap \overleftarrow{\alpha} Q \qquad \text{provided } \overrightarrow{\alpha} P = \overleftarrow{\alpha} Q \end{aligned}$$

The proviso on the alphabet is usually met by adding identity clauses to P and Q, as needed to increase their alphabets, for example

$$P \wedge (\overrightarrow{y} = \overleftarrow{y})$$

But again, we will ignore this slight complexity.

The simplest possible definition of parallel composition of P_0 and P_1 is just their conjunction $P_0 \wedge P_1$. But a proviso is needed to avoid programs like

$$(x := 3 \wedge x := 4)$$

Considered as a predicate, this is uniformly false and so cannot truly describe any observation whatsoever of the real world. Such contradictions must be forbidden in any programming language that is to be implementable. So we define the restricted case of what is called *disjoint* parallelism, by requiring that there is no variable updated by both processes

$$\begin{aligned} P_0 \parallel P_1 &= P_0 \wedge P_1 \qquad\qquad \text{if } \overrightarrow{\alpha}P_0 \cap \overrightarrow{\alpha}P_1 = \{\} \\ \alpha(P_0 \parallel P_1) &= \alpha P_0 \cup \alpha P_1 \end{aligned}$$

An example is a simultaneous assignment to two distinct variables x and y

$$(x := x + y) \parallel (y := x - y) \;=\; (\overrightarrow{x} = \overleftarrow{x} + \overleftarrow{y} \;\wedge\; \overrightarrow{y} = \overleftarrow{x} - \overleftarrow{y})$$

The effect is more commonly written as a multiple assignment

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

Note that when the second expression $x - y$ is computed, the original value of x must always be used, and not the value updated by the other part of the assignment.

In the general implementation of $(P_0 \parallel P_1)$, execution of P_1 (say) may be delegated to a separate processor. But first, a separate copy must be made of the initial value of any variable which is updated by one of them and accessed by the other. As noted above, this is needed to ensure that the accessing process always gets the initial value, and cannot detect any interference from the other process updating it. Synchronisation takes place when execution of both P_0 and P_1 has terminated; and then all the changes made by P_1 must be copied back to the single global store. The additional processor that has executed P_1 can then be released, and the original processor continues execution (if necessary).

But this is not the only way of executing disjoint parallel composition. If neither process updates any of the global variables of the other, parallel execution can be simulated on a single timeshared computer (without copying initial values) by arbitrary interleaving of the actions of the individual processes. This fact is expressed in algebraic laws which formalise the interaction between sequential and parallel composition

$$\begin{aligned} P \parallel \mathrm{I\!I} &= P \\ (P; Q) \parallel R &= P; (Q \parallel R) \qquad\qquad \text{provided } \overrightarrow{\alpha}P \cap \overleftarrow{\alpha}R = \{\} \end{aligned}$$

Under a similar proviso we can deduce

$$(P \parallel R) = (P; R) = (R; P)$$

Disjoint parallelism is the easy case to define and reason about. We now deal with the more interesting case when a certain variable, say m, is updated by both the processes

$$\overrightarrow{\alpha}P_0 \cap \overrightarrow{\alpha}P_1 = \{\overrightarrow{m}\}$$

The same strategy we have described for parallel execution still works (with two copies of m) right up to moment of synchronisation, when the final value of m in the global store has to be determined. Each process offers its own rival candidate, which we will call m_0 and m_1, and these are in general different. To resolve the conflict, we need a *merging* operation M to determine the final value of m, usually as a function of the rival values, and also perhaps of the original initial value of m. So its alphabet is defined

$$\alpha M = \{\overleftarrow{m}, \overleftarrow{m}_0, \overleftarrow{m}_1, \overrightarrow{m}\}$$

Such an M is written as a parameter of the parallel operator $\|_M$

$$P_0(\overrightarrow{m}) \|_M P_1(\overrightarrow{m}) =_{df} (P_0(\overrightarrow{m}_0) \| P_1(\overrightarrow{m}_1)); M \qquad \text{provided } \overrightarrow{\alpha}P_0 \cap \overrightarrow{\alpha}P_1 = \{\overrightarrow{m}\}$$

The parallelism on the right side of this definition has been made disjoint by the substitution of $\overrightarrow{m}_0$ for $\overrightarrow{m}$ in P_0 and $\overrightarrow{m}_1$ for $\overrightarrow{m}$ in P_1.

What then is the predicate M? That is a choice that is made individually for each kind of shared variable and for each kind of theory. In addition, each theory may specify a restricted range of atomic actions by which a process is allowed to update the shared variable. Different choices of merge operator and atomic actions are illustrated in the following sections. They all satisfy the basic laws for parallelism displayed in section 2, but they have different sets of expansion laws.

5 Varieties of parallel composition

As the simplest example of a shared variable, consider a *sum* which is used to accumulate the total of some resource (say machine cycles) used during execution of a program. Every use of the resource is accompanied by an increase of the *sum* by the amount x of the resource consumed.

$$use(x) =_{df} sum := sum + x$$

This is an atomic action, and it is the only kind allowed. When it is invoked in parallel by two processes, we need to define a merge operation that will compute the final value as the sum of the resources used by both processes.

$$M =_{df} sum := sum_0 + sum_1 - sum$$

The final subtraction of the sum is needed to counteract the fact that both processes started with the same initial value of *sum*, and one of them must be removed.

As in the case of disjoint parallelism, it is usually better to avoid taking a separate copy of the shared variable; instead the original variable is updated in its original position in global store by an interleaving of the atomic actions invoked by the two concurrent processes. Of course, some exclusion mechanism is needed to ensure that the actions remain atomic: if one action is in progress, the start of the other must be delayed until the first action is complete. This form of sharing by interleaving is justified by the same algebraic laws as for disjoint parallelism. Their proof depends on the fact that all the atomic actions *commute* with each other

$$use(x); use(y) = use(y); use(x)$$

The next example is of a theory that predicts the minimum amount of elapsed clock time taken by a program; the time can be either simulated or real. Any process which takes one unit of clock time is accompanied by the atomic action

$$tick =_{df} clock := clock + 1$$

When a parallel process terminates, it no longer performs these clock ticks. The final value of the clock is therefore that of the process that terminates later

$$M =_{df} clock := max(clock_0, clock_1)$$

The expansion law for this kind of parallelism indicates that the clock keeps a truly global time, and ticks simultaneously for all the processes in the system

$$(tick; P_0)||_M(tick; P_1) = tick; (P_0||_M P_1)$$

Actions that occur simultaneously in two processes are called *synchronising.*

A commonly shared resource in a parallel computing system is an output device or display on which all processes record messages describing important events in their evolution. This *log* is represented by a shared variable, with values ranging over sequences of messages. The atomic action outputs a message by appending its value to the end of the log

$$op(x) =_{df} log := log\ ^\wedge\langle x\rangle$$

On termination of a parallel execution, the log has been extended by an interleaving of the messages appended by the two processes

$$M =_{df} (\overrightarrow{log} - \overleftarrow{log}) \text{ is an interleaving of } (\overrightarrow{log}_0 - \overleftarrow{log}) \text{ and } (\overrightarrow{log}_1 - \overleftarrow{log})$$

The atomic operations on the *log* obey a form of the commuting law in which the equality had been weakened to an implication

$$op(x); (P_0||_M P_1) \Rightarrow (op(x); P_0)||_M Q_0$$

A logical implication between programs always means that the antecedent is a valid way of implementing the consequent (because, by transitivity of implication, every specification satisfied by the consequent is also satisfied by the antecendent). Actions which obey a law of this kind are called *interleaving*. The full expansion law for the shared log is an equivalence, but a rather complicated and expansive one. It states that of two parallel atomic actions, one must occur first, but the choice is not determinate

$$(op(x); P_0)||_M(op(y); P_1) = op(x); (P_0||(op(y); P_1)) \vee op(y); ((op(x); P_0)||P_1)$$

We have now described three different paradigms for sharing a single variable among parallel processes; and we have classified the atomic actions by their expansion laws as commuting, synchronising, or interleaving. A similar treatment can be given to shared variables of more substantial type, such as sets, bags, or arrays. Indeed, a realistic programming language may include many variables of all these types, and allow them all to be shared at any time among any number of processes, updating them in parallel by any of their permitted atomic actions. Fortunately, the theory of programming for such a complex language is no more complicated than the sum of the theories for its individual single shared variables. All that is needed is to define the merge operation for parallel composition as the conjunction of the separate merges for the individual variables. Similarly, the atomic actions are just the union of the atomic actions on the individual variables. And all their algebraic properties are preserved, whether they be commuting, synchronising or interleaving. The proof of this fact, under quite general hypotheses, perhaps deserves the title of a fundamental theorem of parallel programming.

6 Process Algebra

In our suggested implementation of parallelism we have described the variables shared between processes in terms that suggest that their values will be held, maybe even twice, in the memory of a computer. But this is not essential; and exactly the same theory applies when the shared concept is part of the external environment of the computer, like the output log or the time of day, or even a completely abstract trace of the history of the events in which the system engages. Such a history is called a *trace*, and it is updated, like the log, by an atomic action of the form

$$do(a) =_{df} (trace := trace\ ^\wedge\langle a\rangle)$$

where a is the event whose occurrence is recorded by this assignment.

A theory of parallelism which deals exclusively with the occurrence of events rather than the updating of stored values is called a process algebra. Many varieties have been studied; they are generally known by their acronyms, for example

CCS	Calculus of Concurrent Systems
SCCS	Synchronised CCS
ACP	Algebra of Concurrent Processes
CSP	Communicating Sequential Processes

Each of these has a different definition of the parallel combinator. They are unified by showing that these definitions differ only in the choice of a merge function as parameter.

Our first example is SCCS, in which every event is synchronised in lockstep with events in the history of every other concurrently active process. This theory is adapted to the needs of a certain kind of real-time application, where real-world inputs and outputs are constrained to occur at regular intervals, say 24 times per second. The input data (or parts of it) are distributed to every active process; and the output is composed from data supplied at the same time by every active process. We describe the theory as applied to output. First we need a binary operator, denoted by $|$, which describes the way that output data supplied by two processes should be combined before transmission to their common environment. The environment may include other processes, which will combine this data yet again with their own data before eventual output to the world outside the computer. To simplify this kind of co-operation, the operator $|$ is usually assumed to be associative and commutative.

The merge operation for SCCS is simply defined by applying $|$ to the component outputs produced simultaneously by the two processes

$$\begin{array}{lll} \overrightarrow{trace} := & \overleftarrow{trace}{}^\wedge\langle(a_1 \mid b_1),\ (a_2 \mid b_2), \ldots, (a_n \mid b_n)\rangle & \\ \text{where} & \overrightarrow{trace}_0 \ = \ \overleftarrow{trace}{}^\wedge\langle a_1, a_2, \ldots, a_n\rangle & \\ \text{and} & \overrightarrow{trace}_1 \ = \ \overleftarrow{trace}{}^\wedge\langle b_1, b_2, \ldots, b_n\rangle & \end{array}$$

The expansion law for this merge is obviously of the synchronising kind

$$(do(a); P_0)\|(do(b); P_1) = do(a \mid b); (P_0\|P_1)$$

The total synchronisation of SCCS is relaxed in ACP, which allows an event to occur with the participation of only a subset of the concurrently active processes, missing perhaps any that are not ready. As a result, the merge operation on the traces is a mixture of synchronisation and interleaving, in which each event of each process either occurs independently or is combined by $|$ with the corresponding event of the other processes. For example

$$\begin{array}{l} \text{merges } (t, \langle\,\rangle) = \{t\} \\ \text{merges } (\langle a\rangle, \langle b\rangle) = \{\langle a|b\rangle, \langle a, b\rangle, \langle b, a\rangle\} \end{array}$$

We cannot give the expansion law for ACP or the remaining process algebras, because it uses a variant of the choice operator which is not defined until the last section.

The parallel operation of CCS is a special case of that of ACP. It introduces a special action τ, called the silent action, which represents an internal and invisible transition within a process. Non-silent actions are split into two classes: *outputs* which are indicated by overbar, and *inputs* which are undecorated. A synchronisation only takes place between a single input and a single output, and the result is always a silent action

$$(a \mid \bar{a}) = \tau \qquad \text{for all actions } a$$

CSP is a theory which explicitly differentiates the set of atomic actions that are allowed in each of the parallel processes. The parallel combinator is indexed by these sets: in $(P_A \|_B Q)$, P engages only in events from the set A, and Q only in events from the set B. Furthermore, each event in the intersection of A and B requires synchronous participation of both processes, whereas other events only require participation of the relevant single process. As a result, the parallel composition of CSP is defined by a rather simple form of merge

$$\begin{array}{ll} \overrightarrow{trace} = & \overleftarrow{trace}\,^\wedge t \\ \text{where} & t \in (A \cup B)^* \text{ and } t\restriction A = u \ \wedge t\restriction B = v \\ \text{where} & \overrightarrow{trace}_0 = \overleftarrow{trace}\,^\wedge u \text{ and } \overrightarrow{trace}_1 = \overleftarrow{trace}\,^\wedge v \end{array}$$

In this section we have concentrated on the simplest aspects of each of the four process algebras, the ones that can be described just by a trace of the events which actually happen. Unification has been achieved not by reduction of one algebra to another, but by an abstraction that transcends them all. This leaves open for separate consideration the more interesting differences between the theories. Consider, for example, their treatment of the problem of livelock, which can occur when a system spends all its time on internal communication, ignoring the needs of its external environment. CCS places upon the implementation an obligation to avoid livelock wherever possible by some kind of fair scheduling. CSP places this obligation upon the programmer, so that even strict priority scheduling is a valid implementation. ACP insists that every process engage in at least one action, and avoids livelock by ensuring that every recursively defined process is an unique fixed point of its defining equations. CSP chooses the weakest fixed point in those cases when there is a choice. These and other differences can often be attributed to differences in the chosen style of presentation of the semantics — operational for CCS, algebraic for ACP and model-theoretic for CSP. A potent stimulus to unification is to require a mature theory to be presented in a variety of semantic styles, together with a proof of their mutual consistency.

7 Reactive Systems

The simple trace model of process algebra is fully adequate to deal with processes that engage in communications only with their common global environment. From the environment the system gets its input as an initial value, and provides its output as a final value, observable (in principle) only when all the processes have terminated. But most process algebras also allow internal communication between concurrent processes: from time to time a process inputs a message that has been output by another process; and if the other process is not ready, the input operation has to be delayed until the message is available. In many process algebras, outputs will also be delayed whenever the matching input is not ready. These potential delays are needed for reliable communication, but they carry with them the dreadful risk of lasting forever — a phenomenon known as *deadlock*.

The primary purpose of a theory of programming is to assist in the design of systems guaranteed to avoid this kind of risk. To do this, the theory needs to assume that the phenomenon of waiting is observable, perhaps on many occasions, between the start of a process and its termination. A Boolean variable *wait* is introduced to distinguish intermediate waiting states from termination. A deadlock is represented by a process STOP that admits only waiting states

$$\mathtt{STOP} =_{df} (wait := \text{true})$$

The variable *wait* is shared by all processes, and so it needs a merge operation; this merely states the obvious fact that termination occurs only when both processes have terminated; consequently a parallel composition waits when either of its processes is waiting, even if the other has terminated

$$M =_{df} wait := wait_0 \lor wait_1$$

While a process is waiting, its environment may offer various kinds of input and output. If an offer is accepted, occurrence of the communication will be observed as the next event in the trace of the process. But if an offer is refused, the process will continue to wait until an acceptable communication is offered. If no communication is acceptable, the system is deadlocked and will wait forever; indeed that is how deadlock is defined. In order to predict such deadlocks (and thereby avoid them), we need to assume that the refusal of a communication by a process while waiting is just as observable as its acceptance. We therefore introduce a new variable *ref*, which contains the set of communications refused by a process while it is in a waiting state. In other states, whether busy or terminated, the value of *ref* is irrelevant. The *ref* variable is often implemented in hardware as an interrupt inhibit register, which prevents a computer from responding to interrupts which the software is not ready to cope with.

The *ref* variable needs a merge operation

$$M =_{df} ref := ref_0 \cup ref_1$$

This states that an acceptable communication has to be acceptable by both processes. The definition given above applies to the CSP parallel combinator. For a purely interleaving definition of $||$, we would merge the refusals by an intersection instead of union

$$M =_{df} ref := ref_0 \cap ref_1$$

We have now introduced separately the basic observational variables needed to model the synchronisation of internal communications. But they are not as independent as the shared variables treated earlier; they are connected by joint appearance in the same set of atomic actions, which update them all simultaneously. The action is the one that actually engages in an event, say a. It cannot be described as an assignment statement, because it has *three* states in which it can be observed

1. The initial state, as always.
2. A waiting state, in which nothing happens until the other participants in the event are also ready for it. Of course, the offered action cannot be refused.
3. A terminated state, after the event has happened.

These last two occasions are described by the alternatives displayed in the following definition

$$do(a) =_{df} (\overrightarrow{trace} = \overleftarrow{trace} \wedge \overrightarrow{wait} \wedge a \,\tilde{\in}\, \overrightarrow{ref}\,) \vee \ (\overrightarrow{trace} = \overleftarrow{trace}\,{}^\wedge\langle a\rangle \wedge \neg \overrightarrow{wait})$$

But there is a slight complexity we have overlooked. In sequential composition, we must recognise the fact that the waiting states of P are also waiting states of $(P;\, Q)$. Fortunately, Q can distinguish its predecessor's waiting states by testing the truth of *wait*. The obligation on Q to leave them unchanged can therefore be elegantly expressed by the algebraic law

$$Q = \mathrm{II} \lhd \ wait \ \rhd Q$$

Such a condition has been known as a ***healthiness condition***, and a theory of programming must ensure that it is satisfied at least by all programs expressible in the notations of the programming language. These, of course, include the action $do(a)$, which needs to be slightly redefined.

Programs and healthy predicates satisfy many more algebraic laws than arbitrary predicates. For example, the following law expresses the sad fact that deadlock cannot be rescued by any program written to run after it

$$\mathtt{STOP}; P = \mathtt{STOP}$$

Any theory of programming that lays claim to realism must ensure validity of this law. But it is obviously not true of every predicate P, for example

$$(\mathtt{STOP};\, wait := \text{false}) \ = \ (wait = \text{false}) \ \neq \ \mathtt{STOP}$$

Fortunately, the law is valid for all predicates that satisfy the healthiness condition; and that includes all programs and also many designs and specifications. Indeed, one can classify designs and design languages by how many healthiness conditions they satisfy — the more conditions, the closer they are to programs.

The original reason for introducing the complexity of waiting is to allow information to pass reliably from one concurrent process to another. This can now be done quite simply, by means of a new form of choice operator, denoted by plus (+) in CCS and ACP. In CSP it is denoted $[\!]$, and called *external* choice. This is because $P [\!] Q$ behaves either like P or like Q; the choice between them is not arbitrary (as for $(P \vee Q)$), but rather it is made by another process running concurrently, as explained below.

Suppose $\mathbf{0}$ is an event in which P is ready to engage, but Q refuses it. Conversely, let $\mathbf{1}$ be an event refused by P, but initially acceptable to Q. Let R be a concurrent process in the environment. All three processes have both $\mathbf{0}$

and **1** in their alphabet. Now R can begin by performing **0**, with the effect that P will be chosen for execution; or it can select Q by choice of the initial event **1**. The effect is neatly described by the following expansion laws, using CSP $||$

$$((\mathbf{0};P)[\!](\mathbf{1};Q)) \,||\, (\mathbf{0};R) = \mathbf{0};(P \,||\, R)$$
$$((\mathbf{0};P)[\!](\mathbf{1};Q)) \,||\, (\mathbf{1};R) = \mathbf{1};(Q \,||\, R)$$

Effectively, R has communicated one bit of information to its concurrent partner, which uses it to determine the subsequent behaviour, either P or Q. More information can be communicated by choice between a larger set of alternatives.

The definition of external choice is amazingly simple, and uses only propositional connectives

$$(P[\!]Q) = (P \wedge Q) \lhd (\overrightarrow{trace} = \overleftarrow{trace}) \rhd (P \vee Q)$$

Observations of this process fall into two classes. Either nothing has happened ($\overrightarrow{trace} = \overleftarrow{trace}$), in which case $(P[\!]Q)$ can refuse an event just if both P and Q refuse it; or else something has happened, and every subsequent observation is either one of P or one of Q. By looking at the actual event, you will usually be able to tell which of P or Q has been chosen; but if both P and Q allow this same first event a, the choice between them is non-deterministic, as shown by the law

$$(a;P)[\!](a;Q) = a;(P \vee Q)$$

Reactive systems are distinguished from familiar sequential systems by the introduction of observable waiting states. But they are usually implemented in a programming language which mixes sequential and reactive features. We have shown how to develop theories for each kind of feature separately. In putting the theories together, it may be necessary to adjust them to account for possible interactions. This is done simply by ensuring that the programs of each theory satisfy the healthiness conditions of the other. Much of the apparent complexity of programming language semantics arises from this multiplicity of healthiness conditions.

We have not abolished the complexity, any more than chemistry abolishes the complexity of molecules compounded from their simpler elements. But as in chemistry, we have shown how to control the complexity by analysing it into components studied separately at first, and then exploring their interactions. As in chemistry, our work is built on the discoveries of many earlier researchers, only some of whom are mentioned in the references. And for the future, we hope to enlist the help of many other theorists in continuing our exploration of the wide range of possible programming paradigms, their implementation in different technologies, and their application in the design and delivery of useful and reliable systems.

References

1. J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.
2. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
3. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, 1976.
4. C.A.R. Hoare. *Communicating sequential processes*. Prentice-Hall International series in computing science. Prentice-Hall International, Englewood Cliffs, N.J. London, 1985.
5. C.A.R. Hoare. The varieties of programming language. In *Proc TAPSOFT*, volume 351 of *LNCS*, pages 1–18. Springer, 1989.
6. C.B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP '83*, pages 321–332. North-Holland, 1983.
7. L. Lamport. The Hoare logic of concurrent programs. *Acta Informatica*, 14:21–37, 1980.
8. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
9. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.
10. C. Stirling. A generalisation of Owicki-Gries's Hoare logic for a concurrent while language. *Theoretical Computer Science*, 58:347–359, 1988.