Communicating Parallel Processes

Stephen Brookes

Abstract

Tony Hoare's 1978 paper introducing the programming language *Communicating Sequential Processes* is now a classic. CSP treated input and output as fundamental programming primitives, and included a simple form of parallel composition based on synchronized communication. This paper provides an excellent example of Tony's clarity of vision and intuition. The notion of processes is easy to grasp intuitively, provides a natural abstraction of the way many parallel systems behave, and has an abundance of applications. Ideas from CSP have influenced the design of more recent programming languages such as *occam* and Ada. Investigations of the semantic foundations of CSP and its successors and derivatives have flourished, bringing forth a variety of mathematical models, each tailored to focus on a particular kind of program behavior. In this paper we re-examine the rationale for some of the original language design decisions, equipped with the benefit of hindsight and the accumulation of two decades of research into programming language semantics. We propose an "idealized" version of CSP whose syntax generalizes the original language along familiar lines but whose semantics is based on *asynchronous* communication and *fair* parallel execution, in direct contrast with the original language and its main successors. This language permits nested and recursive uses of parallelism, so it is more appropriate for us to refer to *Communicating Parallel Processes*. We outline a simple semantics for this language and compare its structure with the most prominent models of the synchronous language. Our semantic framework is equally well suited to modelling asynchronous processes, shared-variable programs, and Kahn-style dataflow networks, so that we achieve a unification of paradigms.

1 Introduction

In a now classic article, published in the *CACM* in 1978, Tony Hoare proposed a parallel programming language known as CSP, short for *Communicating Sequential Processes* [17]. This paper introduced a notion of *process*, intended as a mathematical abstraction of the interactions between a computing system and its environment. Building on the sequential control primitives of Dijkstra's language of guarded commands [12], CSP treated input and output as fundamental programming primitives, and introduced a form of synchronized parallel composition: a process reaching an input (respectively, output) command waits until the corresponding process reaches an output (respectively, input) command, whereupon they can communicate by handshake, one process sending a data value and the other receiving it. The paper provides an excellent example of Tony's clarity of vision and intuition, and included a wide-ranging collection of parallel programming problems and their solution in CSP. Ideas from CSP have also influenced the design of more recent parallel programming languages such as *occam* [21], Ada, and Concurrent ML (CML) [33].

The original paper contains an informal sketch of a semantics for CSP, couched in abstract terms but with plenty of operational intuition. The paper discusses a number of important pragmatic issues and provides a rationale for certain key language design decisions and a critique of several alternatives. Subsequently Hoare and his students and colleagues, including Bill Roscoe and myself, worked on formalizing the intuitive semantics of CSP. Many variants of the original language have been introduced, and much of the work has been concentrated on an abstract version of CSP (sometimes known as TCSP, or *Theoretical CSP* [2]) which suppresses the imperative aspects of the language, and various generalizations such as *Timed CSP* [32]. Somewhat confusingly the name CSP tends to be retained across many of these successor or derivative languages, even though the bare syntax of each differs greatly from that of the original language and (more importantly) the semantic models differ radically.

In this paper we revisit the rationale for some of the original language design decisions, equipped with the benefit of hindsight and the accumulation of two decades of research into programming language semantics[1]. We

[1]Perhaps it is worth remarking that it is now 21 years since the publication of the CACM paper on *Communicating Sequential Processes*, so that CSP may be said to have "come of age".

propose an idealized alternative version of CSP based on *asynchronous* communication and *fair* parallel execution, in direct contrast with the original language and its main successors [8]. Like many of CSP's existing variants we also permit nested parallel composition and we allow the use of parallel composition inside the body of a recursive procedure, so that our language deserves the title *Communicating Parallel Processes*.

We describe a simple semantic model for this language and we compare its structure and features with those of the most prominent models of the synchronous language. A virtue of our semantics is that it brings out the underlying essential similarities between three erstwhile separate paradigms of parallel programming: (asynchronous) communicating processes, shared-variable parallelism, and Kahn-style dataflow networks. This kind of unification of paradigms is an achievement that fits well within the spirit of Tony Hoare's research principles.

2 The design of CSP

As originally specified CSP augmented Dijkstra's language of guarded commands with primitives for input and output, and a form of parallel composition of named sequential processes.

In a parallel construct of the form $[\pi_1::P_1 \| \cdots \| \pi_n::P_n]$ the processes P_i $(i = 1, \ldots, n)$ were built from sequential programming constructs together with input and output; the process names π_i are assumed to be distinct, and the processes must be *disjoint*, in that no variable subject to change by one process is used by any other process. This means that synchronized input-output is the only way for processes to influence each other. An input command $\pi_j?x$ in process P_i represents a request for P_j to send an update value for the variable x; an output command $\pi_i!e$ in P_j represents a request for P_i to receive the value of expression e. If P_i and P_j reach such a "matching pair" of communications they may synchronize, with the effect of assigning the value of e to x, so that this kind of handshake can be viewed as a "distributed assignment".

Input commands were also permitted in the guards g_i of a guarded conditional command

$$\textbf{if } (g_1 \to P_1) \,\square\, \cdots \,\square\, (g_n \to P_n) \textbf{ fi}$$

or a guarded loop

$$\textbf{do } (g_1 \rightarrow P_1) \Box \cdots \Box (g_n \rightarrow P_n) \textbf{ od},$$

and in general a guard may be built from a purely boolean component (a boolean expression b) and an input command[2]. A guard of form $b \wedge \pi_j?x$ in the body of process π_i is true in any state satisfying b when the process named π_j is at a "matching" output of form $\pi_i!e$. Such a guard fails if no matching output is available. An input-guarded conditional must wait if necessary until matching output appears. An input-guarded do-loop terminates when all of the processes named as sources in its input guards have terminated. The latter requirement, known as the *distributed termination convention*, has proven rather controversial, awkward to model, difficult to implement, and hard to reason about. Hoare's misgivings about this feature were clearly stated in the original paper. This seems to be one of the early language design decisions that has not survived the test of time, and is not usually regarded as part of the "essence" of CSP.

The form of parallelism provided in CSP is in marked contrast to the kind of *shared memory* parallelism characteristic of Dijkstra's language of *cooperating sequential processes* [11], although the choice of a similar name (together with explicit acknowledgement by Hoare) emphasizes the common intellectual roots of these two paradigms. Hoare was also aware of the then recent paper of Kahn, outlining an extremely simple denotational model of deterministic asynchronous communicating processes [22]. In Kahn's semantics a deterministic process computes a continuous function from input streams to output streams, so that the Kahn paradigm seems inherently "functional" in contrast to the imperative CSP style of process. Given the very different intuitive process models inherent in these three paradigms it is not surprising that over the years the semantic foundations of these paradigms have grown apart, obscuring their common features and underlying similarities. We will return to this point later.

As already mentioned, Hoare's paper included an extensive discussion section which outlined some of the decisions made in his language design and suggested the possible advantages or disadvantages of various alternative choices. We will now revisit some of the main issues, interspersing points from Hoare's critique with our own commentary. The reader should of course remember that our comments are being made with the benefit of hindsight.

[2]Some versions of CSP also permit output guards in such contexts, but this is a matter of some contention and is not important for our purposes.

2.1 Communication vs. assignment

Hoare's use of a special syntax for communication emphasizes his apparent view that communication and assignment are independent and orthogonal concepts: assignment was "familiar and well understood", input and output "not nearly so well understood" [17]. A CSP-style notation for input and output is now widespread, with similar syntax occurring for instance in *occam* and Ada as well as a host of other programming languages.

The separation of communication from assignment has also shaped the development of semantic foundations for CSP. Since the original language was imperative, state change is an important aspect of program behavior, and we normally use the term state to refer to the current values of a process's local variables; but a process's behavior will also depend on what communication possibilities it offers, and on what matching communications are presented by its environment. Hoare-style semantics typically contains a separate component representing information about *environmental* interactions, such as communication traces and refusal sets, as we will see in more detail shortly. Our point here is that these semantic models treat (internal) "state" and (external) "environment" as separate components, rather than regarding local variables and communication potential together as comprising the "overall state" of a process. We will see later that it can be advantageous, from the semantic point of view, to blend communication channels into the "state" and regard input and output as (special kinds of) imperative operations that cause a state change, thus *blurring* the distinction between assignment and communication.

2.2 Nested parallelism

Hoare realized that the *sequential* restriction on the processes of a parallel composition was rather severe. Limiting the P_i to the sequential subset of the language means that one cannot "nest" parallel constructs, so that one needs a whole family of n-ary forms of parallel composition and cannot make do with an associative binary parallel composition operator. This is especially annoying to semanticists, who generally prefer dealing with a single binary construct to coping with an entire family of closely related constructs which cannot be derived from the binary case.

As another consequence of the sequentiality restriction the number of processes in a program is statically determined by the program's syntactic

structure. This may prove advantageous when trying to schedule execution of a particular program, but prevents the language's use to describe dynamic or recursively evolving networks.

Hoare also realized that explicit naming of the processes in a parallel composition creates pragmatic difficulties, for instance with library programs. Any program intended to be used as part of a library must name explicitly all processes with which it might ever need to communicate, but obviously these process names are not typically available.

Even if one were to permit nested uses of parallel composition the process-naming convention would cause difficulties with *scoping* and *associativity*. Consider for example the program

$$[\pi_1::[\sigma_1::P_{11} \parallel \sigma_2::P_{12}] \parallel \pi_2::P_2].$$

How might P_2 communicate with the process named π_1? Should P_2 be permitted to refer to process names σ_1 and σ_2 directly, or perhaps to employ compound names $\pi_1.\sigma_1$ and $\pi_1.\sigma_2$? The latter seems more logical, since we can interpret a compound name like $\pi.\sigma$ to mean "the sub-process named σ of the process named π". But even if we adopt the use of compound names it is difficult to formulate a general associativity law: the process names get in the way. The kind of name calculus inherent in such an approach seems clumsy and unappealing, and the awkwardness becomes more painful still when we generalize to n processes.

Considerations like these suggest a de-coupling of the naming mechanism from the parallel construct, as proposed in Plotkin's operational semantics for CSP [31]. A more flexible mechanism, based on named channels, was adopted subsequently, notably in *occam* and TCSP. These and other derivatives of CSP have typically included binary parallel operators and a separate scoping construct; the *hiding* operator in TCSP serves to localize certain communications between processes, and plays a role analogous to a local declaration in limiting the visibility of actions and names construed to be local.

2.3 Channel names vs. process names

Aware of the "library problem" outlined above, Hoare cited *port naming* (or named channels) as an attractive and more general alternative to process naming. The rigid use of process names in the manner of the original paper

is tantamount to assuming a single port or channel connecting each pair of processes. Most subsequent work in the CSP framework assumes channel-based communication: an output command $h!e$ represents an attempt to output the value of e on channel h; and input command $h?x$ represents an attempt to receive a new value for variable x off channel h; when two processes reach a matching pair of communications (on a particular channel) they can synchronize and perform a handshake.

Although the move to named channels rather than named processes is a very natural generalization it raises further pragmatic and semantic issues. The most fundamental questions concern the nature and usage of a channel. Intuitively, we regard a channel as an abstraction of a communication path between processes. Consequently a channel is naturally to be regarded as *shared* between the processes which use it. CSP insists that processes do not share *variables*, but it would obviously not be sensible to insist that processes do not share channels. (There is also the question of whether to allow more than two processes to share a single channel.) In any case we will see that there are good semantic reasons to blur the distinction between variables and channels, and it will turn out to be straightforward to give an account of a language permitting both variables and channels to be shared among processes.

2.4 Synchrony vs. asynchrony

CSP assumed a synchronous implementation of communication, so that an input or output command in one process is delayed until the other process is ready with corresponding output or input, such delay being "invisible" to the delayed process. All traditional models of CSP reflect this assumption, and abstract away from any delay implied by this mechanism, so that one typically deals with "events" whose occurrence is assumed to be instantaneous.

As Hoare commented, it would have been equally reasonable to adopt and assume an *asynchronous* notion of communication in which a process attempting output should always proceed and a process wanting input should only wait if a matching output is not yet available. Indeed this is the form of communication assumed in Kahn's dataflow semantics, and Kahn even used a CSP-like notation for input and output.

An obvious way to implement asynchronous communicating processes is to employ buffers to hold data waiting to be consumed. Hoare argued

against adopting asynchrony as the underlying mechanism, partly because he regarded it as "less realistic to implement", and partly because buffering can "readily be specified" using the synchronous primitives. With characteristic honesty he also admitted that the second of these reasons is uncompelling, since one could equally well argue that a synchronization can be specified readily with a pair of asynchronous operations. In retrospect the argument against implementability of asynchronous communication is also weak, and an argument at least as forceful could have been made that instantaneous synchronized handshakes are hard to implement. Perhaps it is best to agree that both synchrony and asynchrony are implementable, with manageable cost and overhead. Modern communication-based parallel languages such as Concurrent ML (CML) include asynchronous primitives as basic constructs and implement their synchronous cousins on top of them [33].

Given these philosophical and pragmatic considerations one might be led to believe that the decision to assume synchrony or asynchrony has little direct pay-off. Programmers wishing to work in the "other" setting can always do so with a little extra effort, either inserting buffer processes to simulate asynchrony in the synchronous language, or using request-acknowledge protocols to enforce synchronization on top of an asynchronous implementation. Yet the decision turns out to have deep ramifications, especially for semantic foundations. A major case in point concerns *fairness*.

2.5 Fairness

It has long been recognized that fairness is important when reasoning about parallel programs, since the assumption that parallel processes are executed without unreasonable delay is vital in establishing many liveness properties of networks [29]. A fairness assumption allows us to abstract away from imponderable details of process scheduling, to prove program properties that can safely be asserted to hold in any "reasonable" implementation.

Around the time of Hoare's paper fairness seemed to be semantically awkward, because of the apparent difficulty of reconciling fairness with powerdomains, which were currently being used by Hennessy and Plotkin for modelling non-determinism and concurrency [16]. Park's 1979 paper on the semantics of fair parallelism (albeit for shared-variable programs) was not yet in print. In the prevailing climate of 1978 fairness may have seemed to be more trouble to model than it was worth. For example, the *history tree* semantics of CSP [15], couched in terms of powerdomains, did not attempt

to model fair execution.

Moreover the concept of fairness is not so simple to formalize. The informal description of fairness given above, which seems to be consistent with Hoare's intended usage, is not precise enough. Many alternative formalizations of fairness notions have since been proposed[3]. For our purposes it suffices to focus on *strong* and *weak* fairness. A scheduler is said to be *strongly fair* if it guarantees that every process that becomes "enabled" infinitely often will eventually be scheduled; a scheduler is *weakly fair* if every process that becomes persistently enabled will eventually be scheduled. In other words, weak fairness means that *continuously* enabled processes get scheduled, and strong fairness means that *continually* enabled processes get scheduled.

The following example is adapted from Hoare's original paper, presented here in a slightly more modern notation. We also include the key points made in Hoare's analysis of the example, although we will draw rather different conclusions. Hoare made no explicit distinction between weak and strong forms of fairness.

Assume that a represents an integer-carrying channel, and that go and n are, respectively, a boolean variable and an integer variable. Consider the program

$$
\begin{array}{lll}
a!0 & \| & n{:=}0;\ go{:=}\mathbf{true}; \\
 & & \mathbf{do} \\
 & & \quad (go \wedge a?x \rightarrow go{:=}\mathbf{false}) \\
 & & \Box\ (go \qquad\ \rightarrow n{:=}n+1) \\
 & & \mathbf{od}
\end{array}
$$

The first process wants to output on channel a. The second process initializes its variables and enters a guarded loop, using go as a flag to cause termination, which can only occur if the first guard gets selected.

Hoare says that it would be *unfair* to keep executing the second alternative of the do-loop, since this would keep ignoring the potential for synchronized communication between the two processes, which could have been performed on an infinite number of occasions. Since synchronization is only enabled when both processes are ready, such communication is only intermittently possible during such an execution, *continually* enabled but not *continuously* enabled, infinitely often but not forever. An execution which keeps choosing the second guard and incrementing n is thus *not strongly*

[3] A panoramic survey of a plethora of fairness definitions is provided in Nissim Francez's book [14]. It is doubtful if most of these truly constitute reasonable abstractions.

fair, which is presumably what Hoare meant by "unfair". (After all, such an execution *is* weakly fair.) If we assume that the scheduler is strongly fair the above program is guaranteed to terminate, but the final value of n may be any non-negative integer; if we assume a weakly fair scheduler the additional possibility of non-termination arises. In both cases the program exhibits unbounded non-determinism, since the value of n is unbounded. At the time of Hoare's paper the treatment of unbounded non-determinism in the powerdomain setting seemed technically challenging.

Hoare raised the question: Should a programming language definition specify that an implementation must be (strongly) fair? In answer, citing unbounded non-determinism as the main reason, he was "fairly[4] sure that the answer is NO". It is certainly arguable that strong fairness is not a realistic abstraction: a scheduler can only achieve strong fairness by maintaining book-keeping information, perhaps using a priority queue, concerning the set of currently enabled processes; it is not reasonable to assume that this extensive and expensive work is going on in the background whenever we run a parallel program. I would therefore agree with Hoare's dismissal of strong fairness, given that Hoare seems to have used "fairness' to refer to the strong notion. However these criticisms and defects are not also relevant for weak fairness.

Weak fairness *is* a much more reasonable abstraction from realistic schedulers: any scheduler based on a simple strategy such as round-robin will be weakly fair. If we assume only that processes are scheduled in a weakly fair manner we will be able to prove program correctness properties that hold in *any* reasonable implementation. The problems caused by modelling unbounded non-determinism can be handled appropriately with proper choice of semantic model, as shown initially by Park in 1979 and in later work of other researchers. There is, therefore, a strong case to be made for building a semantic framework for CSP based on weakly fair parallel composition.

It is also worth noting that Hoare went on to suggest that "an efficient implementation should try to be reasonably fair" and should ensure that "an output command is not delayed unreasonably often after it first becomes executable". Perhaps, looking back, this can be read as an implicit nod in the direction of weak fairness, if we assume that Hoare intended "reasonably fair" to mean "weakly fair".

[4]The pun was presumably intended.

3 The models of CSP

Following the initial CSP paper, with its appealingly simple notion of process but only informal discussion of semantic issues, a veritable industry grew up, largely based at the Programming Research Group under Tony Hoare's supervision, with the aim of developing semantic foundations for communicating processes. We will now summarize some of the semantic design issues that arose in those investigations, and which led to the formulation of TCSP, a simple *traces* model [19], the *failures* semantics [2], and later to the more refined *failures+divergences* semantics [3, 20]. Many of the foundational details were developed in the D.Phil. theses of Bill Roscoe and myself [35, 1]. Alternative models have also been developed, notably [27]. We supply only a truncated and grossly over-simplified picture; the reader is referred to Roscoe's book for a more systematic account [36].

In providing a formal semantics it is desirable to work with a streamlined or stripped-down programming language with a minimum of constructs, so that fewer semantic clauses need be specified. Correspondingly these semantic investigations began with an abstract channel-based version of CSP, so that parallel composition could be treated as a binary associative operator rather than requiring n-ary parallel operators for all $n > 0$. (Actually TCSP included special notation for a "synchronous" parallel operator, an "asynchronous" or interleaving parallel operator, as well as a general "mixed" alphabetized form.) Input, output, and local actions such as assignment were regarded as *events*. Instead of local variable (and local channel) declarations *à la* Algol a similar purpose was served by the *hiding* operator. For example in the TCSP process

$$[(a?x; b!x) \parallel a!0]\backslash a$$

channel a is hidden, causing communication on this channel to occur "autonomously"; this process is equivalent to $x{:=}0; b!0$.

Hoare's paper stressed the conceptual distinction between the well known assignment operation, which changes the "internal state" of a machine, and the ("less well understood") communication primitives, which affect the "external environment" of a machine. Correspondingly, traditional semantic models of CSP have been set up to reflect this distinction, and keep "state" (the values of local variables) separate from "environment" (communication on external channels). Our discussion is hampered by the fact that many of these models ignore imperative features and that TCSP is usually presented

as a *process algebra* over an "alphabet" of abstract events. In our presentation we will put the state back in, so as to facilitate comparison with the original language.

3.1 Communication traces

An early semantic model for CSP [19] was based on communication traces, which describe the sequences of communications a process can perform in a finite amount of time if placed in an environment that offers a sequence of matching actions. Such traces represent *partial* histories of interaction up to some finite stage of execution, so that the trace set of a process is naturally non-empty and prefix-closed. This model has the virtue of extreme mathematical simplicity, but is too abstract for many purposes, because it abstracts away from *deadlock*. For example the processes

$$P = \textbf{if } (\textbf{true} \to a?x) \,\Box\, (\textbf{true} \to b?x) \textbf{ fi}$$

and

$$Q = \textbf{if } (a?x \to \textbf{skip}) \,\Box\, (b?x \to \textbf{skip}) \textbf{ fi}$$

have identical trace sets, but $(P \| a!0) \backslash a$ can deadlock and $(Q \| a!0) \backslash a$ cannot.

3.2 Failures

The desire to interpret deadlock properly led to the *failures* semantics of TCSP, in which a process denotes a set of failures, closed under certain natural conditions [2]. A failure has the form (s, α, X, s'), where s and s' are states describing the values of the variables used by the process, α is a communication trace as above and X is a *refusal*, a set of events that the process can fail to accept. A process P exhibits such a failure if it is possible for P to reach deadlock in state s', when executed from state s in an environment that permits the sequence α and then wants to perform any event from X. Such a deadlock is caused by the process "refusing" the set X, which prevents further communication because the environment's next step must correspond to a member of X.

The failures model distinguishes between the above processes P and Q appropriately: P can (initially) refuse communication on a, but Q cannot. Technically this difference manifests itself in the existence of failures such as $(s, \epsilon, \{a?, a!\}, s)$ for P but not for Q. Failure semantics can be defined

denotationally, yielding a compositional model of TCSP tailored to reasoning about communication traces and deadlock. However, there is a further behavioral phenomenon not properly handled using failures: *divergence*.

3.3 Failures + divergences

A process is said to diverge if it can perform an infinite sequence of "internal" actions. Such a potential may be regarded as bad, because it could prevent the process from responding (either by accepting or refusing a communication offered by the environment) in a finite amount of time. In any case it is natural to ask what failures should be taken to represent the behavior of a diverging program such as

$$[\textbf{do true} \rightarrow a?x \textbf{ od} \parallel \textbf{do true} \rightarrow a!0 \textbf{ od}]\backslash a$$

Obviously no visible communication ever occurs here, but equally well the process's environment will never discover in a finite time whether a matching communication might become available, so no refusals will occur either. Simply put, there is no way to represent this kind of behavior inside the failures framework.

A slightly more complicated example raises a further issue with divergence. What if a program may either diverge or do something visible by communicating? Consider for instance the following program:

$$[\textbf{do } (a?x \rightarrow \textbf{skip}) \ \Box \ (in?y \rightarrow out!y) \textbf{ od} \parallel \textbf{do } (\textbf{true} \rightarrow a!0) \textbf{ od}]\backslash a$$

This program has executions in which it diverges, forever assigning 0 to x; it also has the potential to keep behaving like a 1-place buffer between channels *in* and *out*. The question is: to what extent should the potential for divergence influence our view of this program?

An analogous issue arises when modelling non-deterministic sequential programs, for instance in Dijkstra's language of guarded commands. There are three obvious alternatives in the sequential setting:

- ignore non-termination
- insist on termination
- model non-termination and termination separately

In fact three distinct *powerdomains* have become associated with these three alternatives: respectively, the *Hoare powerdomain* – so called because of its connection with *partial correctness* and hence with *Hoare logic*[5]; the *Smyth powerdomain* [39], reflecting *total correctness*; and the *Plotkin powerdomain* [30], providing a more general account that deals properly with non-termination as a legitimate form of behavior.

For CSP semantics concerns other than termination *per se* are vital; we want to be able to reason about communication sequences and deadlock. It is obviously inappropriate to ignore non-termination completely; indeed if we did this we would run into technical difficulties with the failures approach, since a divergent process would have an *empty* set of failures. The choice made in the early development of CSP models was to argue that even potential divergence is "catastrophic", since processes really ought to be designed so as to respond in a finite amount of time to their surrounding environment. We will see later that the third alternative is equally tenable.

The first model to provide a proper account of divergence, modulo the catastrophic assumption, is now known as *failures-divergences* semantics [3]. A process is modelled by a set of failures as above, together with a set of *divergence traces*; a divergence trace (s, α) for process P means that, when P is run from start state s in an environment that allows the sequence of communications α, it is possible for P to begin to diverge. In line with the desire to treat divergence as a disaster, the model imposed certain closure conditions on the failures and divergences of a process, so that all potentially divergent processes are "equally bad" and become indistinguishable. Note, for instance, that the two programs discussed above, one simply diverging and the other only potentially diverging, are ascribed identical meanings in this model.

Despite the historical adoption of the Smyth-style approach to divergence and the persistence of this philosophy in the CSP school of research it is natural to ask what might have been done differently if we had instead taken a Plotkin-style view of divergence. Indeed Roscoe's book does discuss a more refined treatment of divergence, based on joint work with Albert Meyer and Lalita Jategaonkar, and closely related to work of Valmari [41]. We will see shortly that a simpler approach also works, provided we make a few alterations in the surrounding semantic fabric.

[5] As far as I know the attribution of this powerdomain to Hoare is by acclamation; the "Hoare powerdomain" did not appear first in a paper of Hoare.

3.4 Assessment

All of the early models of CSP shared a common philosophy: aiming for mathematical simplicity, focussing on a specific combination of program properties, and dealing in terms of "partial" computations. The most sophisticated of these, the failures-divergences model, viewed divergence as catastrophic. The success of this semantic framework is quite striking. CSP has been applied in a huge variety of settings, ranging from standard chestnuts (such as Dining Philosophers) to systolic arrays [20] and security protocols [23]. Yet it is worth examining what developments might have occurred if we had begun by adopting a different set of philosophical principles. We should also note that as a consequence of the way the traditional framework evolved it is a rather difficult to incorporate some of the features missing in the original CSP.

In particular none of the now traditional semantic models of CSP has embraced fairness (either weak or strong) as a foundational assumption. As Roscoe describes, in order to cope with fairness in the failures-divergences framework one must incorporate *infinite* communication sequences as well as *finite*. This seemingly natural step is actually far from straightforward [36]. The problems that arise, and the book-keeping intricacies with which it is necessary to dress up the semantic details, are set out in Sue Older's Ph.D. thesis [26]. The conclusion is that the synchronous nature of communication in CSP makes it extremely difficult to deal tractably with fairness, since "enabledness" of a process depends on the ability of other processes to agree to a matching communication and this forces us to push around (in the semantic clauses) information about the sets of potential communications at all stages of an execution.

4 Idealized CSP

We will now introduce an "idealized" version of CSP based on *asynchronous* communication and (weakly) *fair* parallel composition [8]. The language can also be viewed as generalizing Reynolds' Idealized Algol by adding input and output primitives and the ability to spawn parallel processes. Our generalization of Hoare's language allows nested and recursive uses of parallelism, and we use named channels, as in *occam*, rather than process names, since this yields a more flexible communication mechanism. The inclusion of nested

parallelism makes the language more uniform and causes no extra difficulties from a semantic point of view.

The combination of procedures and parallelism was already suggested in the original paper on CSP: Hoare commented on the similarity between his notation for an array of processes and Algol-like procedures. We permit recursive procedures, and even the use of parallel composition inside a procedure body, so that it becomes straightforward to specify dynamic process creation. Procedures can also be used to encapsulate common communication protocols, such as the alternating-bit protocol. Local variable declarations and local channel declarations provide a way to delimit the scope of interference between parallel agents.

A raw syntax for our language is pretty standard, and is described as follows. (We omit the details concerning proper usage of types, which can easily be handled in a conventional manner.) For simplicity we will let $x, y, \ldots$ range over (integer-valued) identifiers and h range over (integer-carrying) channel names; d ranges over declarations, which for simplicity we write as a sequence of identifiers and channel names. We also let e range over integer-valued expressions and b range over boolean-valued expressions, whose syntax is not further specified here. An abstract grammar for processes P, guarded commands gc, and guards g is given by:

$$\begin{array}{rcl} P & ::= & \textbf{skip} \mid x{:=}e \mid P_1; P_2 \mid h?x \mid h!e \mid \\ & & \textbf{if } gc \textbf{ fi} \mid \textbf{do } gc \textbf{ od} \\ & & P_1 \| P_2 \mid \textbf{local } d \textbf{ in } P \\ gc & ::= & (g \to P) \mid gc_1 \,\Box\, gc_2 \\ g & ::= & b \mid b \wedge h?x \end{array}$$

We use an Algol-like notation for procedures. For example, the following procedures encapsulate a common way to build one-place and unbounded buffers in CSP:

$$\begin{array}{l} \textbf{procedure } \mathit{buff1}(\mathit{in},\ \mathit{out}) = \\ \quad \textbf{local } x \textbf{ in do } (\mathit{in}?x \to \mathit{out}!x) \textbf{ od}; \end{array}$$

$$\begin{array}{l} \textbf{procedure } \mathit{buff}(\mathit{in},\ \mathit{out}) = \\ \quad \textbf{local } \mathit{mid} \textbf{ in } \mathit{buff1}(\mathit{in},\ \mathit{mid}) \parallel \mathit{buff1}(\mathit{mid},\ \mathit{out}); \end{array}$$

In any call to *buff*, locality of the channel *mid* guarantees that the actual parameters of the call are distinct from *mid*. The correct behavior of this

procedure depends crucially on the inability of the two calls to *buff1* to interact except via the local channel.

For another example here is one way (*cf.* [22]) to program the Sieve of Eratosthenes in our language:

$$\begin{array}{l}\textbf{procedure } \mathit{filter}(p,\ \mathit{in},\ \mathit{out}) = \\ \quad \textbf{local } x \textbf{ in do } (\mathit{in}?x \rightarrow \textbf{if } x \textbf{ mod } p \neq 0 \textbf{ then } \mathit{out}!x) \textbf{ od} \\ \textbf{procedure } \mathit{sieve}(p, c) = \\ \quad (c!p;\ \textbf{local } h \textbf{ in } \mathit{filter}(p, h, c) \parallel \mathit{sieve}(p+1, h));\end{array}$$

If c is an integer-carrying channel the call $\mathit{sieve}(2, c)$ results in the outputting of the prime numbers in ascending order on this channel. Note that each recursive call to *sieve* introduces new parallel processes sharing a local channel, and each call to *filter* makes use of a local variable to hold the integer currently being tested for divisibility.

Combining procedures and communicating processes raises significant semantic problems. Indeed, the early semantic models for CSP did not incorporate procedures, and most existing semantic models for procedures seem unsuitable for a process language like CSP. Nevertheless, despite the fundamental differences in the underlying model of computation, the ideas behind our earlier work on shared-variable parallelism [4, 7] can be adapted to the setting of communicating processes. In [4] we used "transition traces" to build a simple fully abstract model for a shared-variable parallel language. In [7] we showed how to incorporate a procedure mechanism based on the simply typed call-by-name λ-calculus, obtaining an idealized language called Parallel Algol. Our semantics for Parallel Algol combined transition traces with "possible worlds"[28, 34] in a "modular" style, bringing out the orthogonality of procedures and shared-variable concurrency. With suitable generalization and adjustment, we can obtain a semantics for Idealized CSP by similar means.

The advantage of this approach is that transition traces, which were originally tailored for the shared-variable paradigm, and possible worlds, which appear best suited for modelling imperative programming, can be adapted to deal with communication-based programs. As shown in [9] transition trace semantics also provides a model for non-deterministic Kahn-style dataflow networks. Thus transition traces can serve as a unifying common semantic basis for three parallel paradigms.

To facilitate comparison between our semantics and traditional models of CSP we now summarize briefly some of the key ideas.

4.1 Transition traces

A "transition trace" is a finite or infinite sequence of pairs of states,

$$\langle s_0, s_0' \rangle \langle s_1, s_1' \rangle \ldots \langle s_n, s_n' \rangle \ldots$$

representing a generalized computation of a command during which the state is changed as indicated: steps from s_i to s_i' being caused by the command, changes from s_i' to s_{i+1} being made by the command's environment. This kind of structure is very natural for modelling shared-variable parallelism, since interference is captured precisely by state changes "across step boundaries". Transition traces have been used to give denotational semantics to a simple shared-variable language, originally by Park [29], and by the author in [4] to achieve full abstraction, by imposing certain closure conditions on trace sets. In particular, a trace set T is said to be closed under *stuttering* if every trace obtained from a trace in T by inserting steps of the form $\langle s, s \rangle$ also belongs to T; and T is closed under *mumbling* if every trace obtained from a trace in T by replacing adjacent steps of the form $\langle s, s' \rangle \langle s', s'' \rangle$ by $\langle s, s'' \rangle$ is also in T. In Park's traces each step represents a single atomic action, while in [4] a step represents a finite sequence of atomic actions.

4.2 Blending communication with state

Channel names (or process names, or some similar kind of communication label) play a prominent role in traditional accounts of the semantics of communicating processes. Yet from an abstract point of view the reliance on channel names seems awkward. By analogy, the traditional reliance on a location-based model of machine state causes semantic problems that motivated the search for more abstract models in which location names become implicit [24, 28, 25]. The decision to treat local state and channels as separate aspects of a process's behavior, state being affected by assignment and environment being affected by communication, is the main reason for the prominence of channel names. Nevertheless it is possible to treat channels as just another kind of "variable".

A channel potentially carries a sequence of data values. Over the course of an entire computation an individual channel may participate in an infinite sequence of communications, but at each stage only finitely many actions have occurred so far. It follows that we can regard a channel as a variable holding a finite sequence (actually, a queue) of data, representing those items

that have been output to the channel but not yet consumed by an input operation. We can then treat input and output as operations which modify the queue associated to a channel name; of course an input operation must wait if the channel is currently empty. We can thus blend channels into the state, so that a state describes the current contents of both variables and channels. This paves the way for an adaptation of the transition traces approach to the setting of communicating processes. A trace of the form indicated above now represents a possible computation of a process assuming certain patterns of communication with its environment (modelled as "state changes between steps").

When a process wants to perform input but the intended channel is empty, it seems reasonable to model this situation as a form of *busy waiting*, since such a process will keep waiting for an output to the channel by another process; while waiting, the process never changes the state, and the waiting continues provided the channel stays empty. In trace-theoretic terms this amounts to a form of infinite stuttering.

As usual, sequential composition is modelled by concatenation of traces. Assignment, conditional and while-loops may be handled in the standard way too, as in our earlier treatment of shared-variable parallelism. Recursion and while-loops are interpreted via *greatest fixed-points* [40] in order to deal appropriately with both finite and infinite traces. Assuming a fair scheduler, the behavior of a parallel system of processes can be built by fairly interleaving traces of the individual processes. The *fairmerge* relation on traces can be defined in a straightforward manner, again by means of greatest fixed points [29, 4].

Local channel declarations can be handled rather simply using an extension of the idea used in our shared-variable semantics. The traces of **local** h **in** P are obtained by projecting away the h-component from (the states in) traces of P in which the initial contents of h is the empty sequence and the contents of h are never changed across step boundaries. This "interference-freedom" constraint on local variables reflects the scoping rules: only P has access to the local channel.

Transition traces provide a semantic framework for compositional reasoning about safety and liveness properties of parallel processes, assuming (weakly) fair execution. This semantics validates a collection of useful laws of program equivalence, including several which rely on and reflect the fairness assumption; these are especially helpful in proving liveness properties [9, 10].

Traditional CSP also enjoys a large battery of laws of equivalence, which

have been used to great effect in the development of *model checking* tools [13]. Naturally the move to asynchrony means that we need to work with laws tailored to asynchronous communication. For instance, we obtain the equivalence

$$\textbf{local } h \textbf{ in } (h!0; P) \;=\; P$$

if h does not occur free in P. Note also that

$$\textbf{local } h \textbf{ in } (h?x; P) \;=\; \textbf{while true do skip},$$

having only infinite stuttering traces, because of the unrequited request for input. Notice how these laws reflect our assumptions that an attempt to input from an empty channel is blocked, but output is asynchronous. By analogy the TCSP law

$$(e \rightarrow P)\backslash e = P\backslash e$$

models the assumption that a hidden event can occur "autonomously"; in the asynchronous world this would be reasonable for an *output* event (as in the first law above) but not for an input.

In our semantics divergence is *not* interpreted as catastrophic. We see no good reason to *insist* that a process which may diverge is "as bad" as any other process at all. The rationale typically given for assuming that any possibility of divergence is a disaster is tantamount to insisting that processes should be *designed to terminate*, and this seems excessive in the concurrent setting. Consider for example the program

$$\textbf{do } (\textbf{true} \rightarrow \textbf{skip}) \textbf{ od} \parallel \mathit{buff1}(\mathit{in}, \mathit{out}).$$

Assuming fair parallel composition there is no reason to distinguish this from $\mathit{buff1}(\mathit{in}, \mathit{out})$. The transition traces of both of these processes are identical, so that our model equates them. It would be unreasonable to interpret the first process differently simply because one of its component processes can diverge[6].

5 Conclusions

CSP has proven to be a highly influential contribution to the literature. Its effects have been felt in language design, as witnessed by the rendezvous

[6]This example is also closely related to the "potentially divergent" example discussed earlier, which was presented in TCSP-style notation.

mechanism of Ada, the *occam* language, and the input-output and synchronizing primitives of Concurrent ML. An enormous body of research has grown up dealing with variants and derivatives of CSP, concerning logics for reasoning about programs, the construction of special-purpose semantic models, and specification and verification of program properties by automated techniques [13, 38].

In this paper we have re-examined some of the language design alternatives that were not adopted in the original paper. We have shown that it is possible to build a simple and flexible semantic framework based on *asynchronous* communication and *fair* parallelism, simultaneously suitable for interpreting programs from the shared-variable paradigm and the communicating process paradigm. Despite being based "only" on a form of traces, our semantics provides a proper account of deadlock and divergence. In certain respects our framework has advantages: it provides a much more natural account of fairness than seems possible in the synchronous setting, and we achieve a unification of parallel paradigms belied by the very disparate collection of semantic models that have evolved historically. This kind of unification and its reaffirmation of the common roots of these paradigms, along with the simplicity and generality of our framework, are surely resonant of the research principles and philosophy that characterize Tony Hoare's work.

As shown in [7] our semantics may be recast into a relationally parametric setting [25]. This permits an elegant generalization of the principle of representation independence, familiar from the use of abstract datatypes and modules in sequential programming. This provides another link to an important early paper of Hoare, on proving the correctness of data representations [18].

The traditional CSP models were developed within the established bounds of Scott-Strachey denotational semantics: all program constructs were taken to denote continuous functions on a semantic domain, and the meaning of a recursive definition was interpreted as a least fixed-point. Relying implicitly on the finitistic nature of the failures model (and on "constructivity" properties of program constructs) Hoare's book [20] showed how to reason in a straightforward manner about the correctness of recursive process designs, taking advantage of the fact that (in the failures semantics) any guarded recursive definition has a unique solution. We cannot adopt such an approach in our idealized setting, since our decision to build in fairness means that our model is no longer finitistic, and guarded equations may have more than one solution. Indeed our approach uses *greatest* fixed-points to interpret recur-

sion, in order to give a proper account of infinite behaviors. Despite these complications one can develop a straightforward style of reasoning about fair recursive processes, as outlined in [10].

In summary, Idealized CSP and its semantic framework provides a satisfying alternative to synchronous CSP. It remains to be seen to what extent it is possible to emulate the successes of the original CSP school, for instance by developing automated tools for model checking asynchronous processes, or by incorporating time. Perhaps this will be a suitable topic for discussion when CSP turns 40.

References

[1] Brookes, S., *A model for communicating sequential processes*, D. Phil. thesis, Oxford University (1983).

[2] Brookes, S.D. and Hoare, C.A.R., and Roscoe, A.W., *A theory of communicating sequential processes*, JACM 31(3):560–599 (1984).

[3] Brookes, S. and Roscoe, A.W., *An improved failures model for CSP*, Proc. Seminar on Concurrency, Springer-Verlag LNCS 197, 1985.

[4] Brookes, S., *Full abstraction for a shared-variable parallel language*, Proc. 8th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press (1993), 98–109.

[5] Brookes, S., *Fair communicating processes*, in A. W. Roscoe (ed.), **A Classical Mind: Essays in Honour of C. A. R. Hoare**, Prentice-Hall International (1994), 59–74.

[6] Brookes, S., and Older, S., *Full abstraction for strongly fair communicating processes*, Proc. 11th Conference on Mathematical Foundations of Programming Semantics (MFPS'95), ENTCS vol. 1, Elsevier Science B. V. (1995).

[7] Brookes, S., *The essence of Parallel Algol*, Proc. 11th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press (1996), 164–173.

[8] Brookes, S., *Idealized CSP: Combining Procedures with Communicating Processes*, Proc. 13th Conference on Mathematical Foundations of Programming Semantics (MFPS'97), ENTCS vol. 6, Elsevier Science B.V. (1997).

[9] Brookes, S., *On the Kahn Principle and Fair Networks*, 14th Conference on Mathematical Foundations of Programming Semantics (MFPS'98), submitted to TCS (1998).

[10] Brookes, S., *Reasoning about Recursive Processes: Expansion is not always fair*, ENTCS, Elsevier Science B.V., to appear (1999).

[11] Dijsktra, E. W., *Cooperating sequential processes*, in: **Programming Languages**, in F. Genuys (ed.), Academic Press (1968), 43–112.

[12] Dijkstra, E. W., *Guarded Commands, Nondeterminacy, and Formal Derivation of Programs*, Comm. ACM 18(8):453–457 (1975).

[13] Formal Systems (Europe) Ltd, *Failures-Divergence Refinement: FDR2 Manual*, 1997.

[14] Francez, N., **Fairness**, Springer-Verlag (1986).

[15] Francez, N., Hoare, C.A.R., Lehmann, D., and de Roever, W. P., *Semantics of Nondeterminism, Concurrency, and Communication*, JCSS 19, 290–308 (1979).

[16] Hennessy, M. and Plotkin, G.D., *Full abstraction for a simple parallel programming language*, Proc. 8th MFCS, Springer-Verlag LNCS vol. 74, pages 108-120 (1979).

[17] Hoare, C. A. R., *Communicating Sequential Processes*, Comm. ACM, 21(8):666–677 (1978).

[18] Hoare, C.A.R., *Proof of correctness of data representations*, Acta Informatica 1:271–281 (1972).

[19] Hoare, C.A.R., *A model for communicating sequential processes*, in: **On the construction of programs**, McKeag and McNaughton (eds.), Cambridge University Press (1980).

[20] Hoare, C.A.R., **Communicating Sequential Processes**, Prentice-Hall (1985).

[21] Inmos Ltd., *occam2* reference manual, Prentice-Hall (1988).

[22] Kahn, G., *The semantics of a simple language for parallel programming*, Proc. IFIP'74, North-Holland, pages 471-475 (1974).

[23] Lowe, G., *Breaking and fixing the Needham-Schroeder public-key protocol using FDR*, Proc. TACAS'97, Springer-Verlag LNCS 1055 (1996).

[24] Halpern, J. Y., Meyer, A. R., and Trakhtenbrot, B. A., *The semantics of local storage, or What makes the free list free?*, ACM Symposium on Principles of Programming Languages, 1983, pages 245-257.

[25] O'Hearn, P. and Tennent, R., *Parametricity and local variables*, J. ACM 42(3), 658-709, May 1995.

[26] Older, S., *A Denotational Framework for Fair Communicating Processes*, Ph.D. thesis, Carnegie Mellon University, (1997).

[27] Olderog, E-R., and Hoare, C.A.R., *Specification-oriented semantics for communicating processes*, Acta Informatica 23, 9-66, 1986.

[28] Oles, F.J., *A Category-Theoretic Approach to the Semantics of Programming Languages*, Ph.D. thesis, Syracuse University, 1982.

[29] Park, D., *On the semantics of fair parallelism.* In D. Bjørner, editor, **Abstract Software Specifications**, Springer-Verlag LNCS vol. 86 (1979), 504–526.

[30] Plotkin, G.D., *A power domain construction*, SIAM J. Comput. 5 (3), Sept. 1976.

[31] Plotkin, G. D., *An operational semantics for CSP*, In D. Bjørner, editor, **Formal Description of Programming Concepts II**, Proc. IFIP Working Conference, North-Holland (1983), 199-225.

[32] Reed, G.M. and Roscoe, A.W., *A timed model for communicating sequential processes*, Theoretical Computer Science 58: 249–261 (1988).

[33] Reppy, J., *Concurrent ML: Design, Application and Semantics*, in: **Functional Programming, Concurrency, Simulation and Automated Reasoning**, P. Lauer (ed.), Springer-Verlag LNCS 693, 165–198 (1993).

[34] Reynolds, J. C., *The essence of Algol*. In van Vliet and de Bakker, editors, **Algorithmic Languages**, North-Holland, Amsterdam (1981), 345-372.

[35] Roscoe, A.W., *A mathematical theory of communicating processes*, D. Phil. thesis, Oxford University (1982).

[36] Roscoe, A.W., **The Theory and Practice of Concurrency**, Prentice-Hall (1998).

[37] Roscoe, A.W. and Hoare, C.A.R., *The laws of occam programming*, Theoretical Computer Science, 60:177–229 (1988).

[38] Roscoe, A.W., *Model checking CSP*, in **A classical mind: essays in honour of C.A.R. Hoare**, Prentice-Hall (1994).

[39] Smyth, M.B., *Power domains*, JCSS 16(1):23–36, Feb. 1978.

[40] Tarski, A., *A lattice-theoretical fixpoint theorem and its applications*, Pacific Journal of Mathematics, **5** (1955).

[41] Valmari, A., *The weakest deadlock-preserving congruence*, Information Processing Letters **53**, 341–346, 1995.