Chapter 8

Communication

The fundamental property of a sequential process is that its behaviour can be adequately described by observations made on just two occasions, at the moment of its initiation and at the moment of its termination. The basic distinguishing property of a *reactive* process is that it also admits intermediate observations on suitable occasions between initiation and termination. Additional variables and naming conventions are introduced to denote the results of these observations.

We take a view similar to that of modern quantum physics, that an observation is always an interaction between a process and one or more observers in its neighbouring environment. The role of observer will often be played by other processes, and the subsequent behaviour of all the processes involved will usually be affected by the interaction. In any case, the occurrence of interaction is always regarded as an atomic event, without duration in time. If P is a process, we let $\mathcal{A}P$ denote the set of all actions which are physically and logically possible for the process (even in the case of one that never actually performs them). This is taken to be an unchangeable part of the process alphabet.

A common kind of interaction is a communication between processes; this can be analysed in greater detail as the input or output of a message m along some channel with name c; the event is then denoted by the pair $c.m$, and the set of all such pairs is included in $\mathcal{A}$. The names of the channels which connect a process to its environment are therefore part of its alphabet: they are classified as input channels or as output channels to indicate the direction of transmission. But for many purposes it is simpler to ignore this analysis, and regard a communication as no different from every other atomic event in $\mathcal{A}$. That is the view taken in the early sections of this chapter.

A process may engage many successive interactions, and we assume that they can be recorded sequentially in the order in which they occur. Simultaneous actions are allowed, but the recording conventions require that they are written down in some particular order, maybe arbitrary. The sequence of interactions recorded up to some given moment in time is called a *trace*; this is abbreviated to tr and is included in the alphabet of every reactive process. The variable tr represents the sequence of actions which takes place before the process is started, and the variable tr' stands for the sequence of all actions which have been recorded so far, up to the moment of observation. The values of tr and tr' range over all finite sequences of events, that is the Kleene closure $\mathcal{A}^*$ of the alphabet $\mathcal{A}$. Since the execution of a process can never *undo* any action performed previously, the trace can only get longer. The current value of tr must therefore always be an extension of its initial value. The predicate P which describes a reactive process behaviour must therefore imply this fact. So it satisfies the healthiness condition

R1 $P \;=\; P \wedge (tr \le tr')$

Note that the sequence $tr' - tr$ represents the trace of events in which the process itself has engaged from the moment that it starts to the moment of observation.

The purpose of the undashed variable tr is to permit reuse in the theory of reactive processes of the same definition of sequential composition as in a sequential language. In fact this variable plays no other significant role. In particular it has no influence on the behaviour of the process. So a reactive process also satisfies a second healthiness condition

R2 $P(tr, tr') \;=\; \bigsqcap_s P(s, s^\frown(tr' - tr))$

This means that the initial value of tr may be replaced by an arbitrary s, and the events in which the process itself engages remain the same.

We assume that the occurrence of an event and its observation by the environment are always exactly simultaneous. In fact they are recorded in the trace as the same event. If there is some delay in communication along a channel, we model this by including in $\mathcal{A}$ the separate events of sending the message to the channel and receiving the message from it. When the environment is actually another process, simultaneity requires an implementation to ensure that each event occurs only when both the process and its environment are ready, willing and waiting for it. So occurrence of an event is often preceded by a wait on the part of one (or more) of the processes involved.

Because of the requirement for synchronisation, an active process will usually engage in alternate periods of internal activity (computation) and periods of quiescence or stability, while it is waiting for reaction or acknowledgement from its environment. We therefore introduce into the alphabet of a reactive process a variable *wait'*, which is true just during these quiescent periods. Its main purpose is to distinguish intermediate observations from the observations made on termination. If *wait'* is true, then all the other dashed variables also stand for intermediate observations rather than final ones.

The introduction of intermediate waiting states has implications for sequential composition: all the intermediate observations of P are of course also intermediate observations of $P; Q$. Control can pass from P to Q only when P is in a final state, distinguished by the fact that $wait'$ is false. Rather than change the definition of sequential composition, we enforce these rules by means of a healthiness condition. If the process Q is asked to start in a waiting state of its predecessor, it leaves the state unchanged

R3 $Q \;=\; I\!I \lhd wait \rhd Q$

where the predicate $I\!I$ adopts the following new meaning in this chapter

$$I\!I \;=_{df}\; \neg ok \wedge (tr \leq tr') \;\;\vee\;\; ok' \wedge (tr' = tr) \wedge \ldots \wedge (wait' = wait)$$

A process reaches a waiting state when it can make no further internal progress until after the occurrence of some event that requires participation from its environment. In fact, we allow a process to wait simultaneously for more than one kind of event, in a way that permits the environment to select which of these events will actually occur. For example, an inputting process may wait for any of the possible messages that can be transmitted along some subset of its input channels, and the outputting environment selects which one. An efficient implementation will ensure that the first event which actually becomes possible will occur immediately. The occurrence of the event is observed by the environment and recorded in the trace. The subsequent behaviour of the process is therefore as predictable as before the event. Waiting for the first of a selection of events does not itself introduce non-determinism. Non-determinism only becomes a risk if the event is concealed from the observer.

Another grave risk incurred by reactive systems is that of deadlock. This occurs when the environment continuously refuses to participate in any of the events which the process is waiting for; the usual cause is that the process itself is also refusing to participate in any of the events in which the environment is ready to engage. To avoid such risk, we need to model it by keeping track of the set of events which are refused by a process in each of its waiting states. The variable ref' is introduced to denote this set.

The following definition summarises the discussion so far.

Definition 8.0.1 (Reactive process)

A reactive process is one which satisfies the healthiness conditions **R1**, **R2** and **R3**, and has an alphabet consisting of the following:

- $\mathcal{A}$, the set of events in which it can potentially engage.
- $tr : \mathcal{A}^*$, the sequence of events which have happened up to the time of observation.

- *wait* : *Boolean*, which distinguishes its waiting states from its terminated states.
- *ref* : $\mathcal{PA}$, the set of events refused by the process during its wait. □

As a subtheory, reactive processes can be defined by a monotonic idempotent. Here and elsewhere, we will use the same name for the healthiness condition and its associated embedding.

Theorem 8.0.2

P is a reactive process **iff** it is a fixed point of $\mathbf{R} =_{df} \mathbf{R3} \circ \mathbf{R2} \circ \mathbf{R1}$ where

$$\begin{aligned} \mathbf{R1}(X) &=_{df} X \wedge (tr \leq tr') \\ \mathbf{R2}(X(tr, tr')) &=_{df} \bigsqcap_{s} X(s,\ s^\frown(tr' - tr)) \\ \mathbf{R3}(X) &=_{df} I\!I \lhd wait \rhd X \end{aligned}$$

Proof It is evident that a process P satisfies the healthiness condition $\mathbf{R}i$ iff it is a fixed point of the monotonic mapping $\mathbf{R}i$ defined above. The conclusion follows from the commutativity of the idempotents $\mathbf{R}i$ and Theorem 4.1.7. □

Corollary

Reactive processes form a complete lattice. □

Theorem 8.0.3 (Closure of reactive processes)

The set of reactive processes is a $\{\wedge,\ \vee,\ \lhd tr' = tr \rhd,\ ;\}$-closure. □

The theory of reactive processes with synchronisation has been a prolific topic of research, and many systems and notations have been proposed. In this chapter, we select just three of them for unified treatment on the basis of the observable properties listed above. Ignoring notational trivialities, the theories are differentiated mainly by a different selection of healthiness conditions. The three theories are the Algebra of Communicating Processes (ACP) [23], Communicating Sequential Processes (CSP) [88], and Data Flow (Kahn–McQueen networks) [106, 107].

The simplest model is that of ACP. Its simplicity derives from three related design decisions.

1. It does not include a unit for sequential composition. In fact, every terminating process satisfies a healthiness condition that it must engage in at least one event before termination.
2. It does not include any divergent process. Partly because of 1, it is possible to avoid divergence by restricting recursion to cases in which there is known to be an unique fixed point.

3. There is no explicit refinement ordering between processes. Non-determinism is present, but there is no separate symbol for it.

The well-known Calculus of Communicating Systems (CCS) [125, 126] is very similar to ACP, and the main minor differences will be pointed out in passing. The absence from CCS of a primitive combinator for sequential composition can be easily remedied by a construction. One valuable objective of CCS was to define the common properties of a whole family of process algebras. It therefore preserves a number of more subtle distinctions between processes, which cannot be represented in our limited alphabet of observations. For example, significance is attached to the order in which non-deterministic choices are made. A full explicit model for CCS is therefore beyond the scope of this chapter [46, 81, 83].

In ACP and CCS, internal communications between components of a system are identified by a special event name τ, which is recorded in the trace. It is interpreted as a silent or invisible transition, and its occurrence never needs to be synchronised with the environment. One good motive for its retention in the trace is to preserve the uniqueness of fixed points. Nevertheless, τ presents a noticeable inconvenience in specification and design. The main distinguishing feature of CSP is to define a hiding operator that succeeds in total concealment of internal events. But it is necessary to recognise the danger that a system will engage in an *infinite* sequence of internal actions, and never again reach a stable state in which it is ready to interact with its environment – a phenomenon known as *livelock*. CSP models this danger by divergence, so that its avoidance can be proved.

Our last example paradigm is data flow. This was the first paradigm to be modelled mathematically, and achieved early implementation as the pipe of UNIX. The characteristic property of data flow is that its communication channels are all buffered, so that they can accept an output message at any time, storing it until the process at the other end of the channel is ready to input it. Consequently, a process never has to wait to perform output. The reduction in wait times is paid for in the overhead of buffering and the risk of buffer overflow.

The first theories of data flow were deterministic, in that the output messages had to be functions of earlier input, or otherwise paradoxical results would emerge. By recording the sequence of all events in a single trace, our treatment of non-determinism avoids the classical paradoxes. Data flow is still one of the most elegant of process paradigms, and it can be modelled simply by traces, without any need for explicit waiting states or refusals [33, 34]. The purpose of the data flow paradigm is to replace control flow by data flow. So sequential composition has no role; it is replaced by a chaining operator, which hides the data that are output by one process and transmitted for input by the other.

Both synchronous and asynchronous models of communication have found application in industry.

Example 8.0.4 (LOTOS)

The international standard formal language LOTOS [98] (ISO 8807) is defined for mathematical analysis of networked, distributed and concurrent systems. It uses synchronised communication, and has been beneficially influenced by the process algebras described in this chapter. □

Example 8.0.5 (SDL)

The specification and description language SDL has been standardised internationally (CCITT Z.100, 1993). It is based on an asynchronous model of communication, as in data flow. □

8.1 Algebra of Communicating Processes (ACP)

ACP [23] is an algebra that includes both sequential and concurrent composition. For purposes of unification, we want to use exactly the same definition for sequential composition as in all the theories described so far. This means that we must include in the alphabet both dashed and undashed versions of all variables, for example *tr* as well as *tr'*. Because an ACP process never diverges, there is no need for the variables *ok* and *ok'*. Whenever it is not waiting it has terminated.

In ACP there is a slightly unexpected healthiness condition that no ACP process can terminate before performing at least one atomic action.

ACP1 $[(P \wedge (tr' = tr)) \;\Rightarrow\; wait']$

Definition 8.1.1 (ACP processes)

A reactive process is an ACP process if it satisfies **ACP1**. □

Example 8.1.2 (Deadlock)

The process δ never terminates and never performs a proper action. It is the representation of deadlock, which waits forever

$$\delta \;=_{df}\; \mathbf{R3}(tr' = tr \;\wedge\; wait') \qquad \square$$

Theorem 8.1.3 (Closure of ACP processes)

ACP processes form a complete lattice, which is closed under $\lhd\delta\rhd$ and sequential composition.

Proof From Theorem 4.1.7 and Definition 8.1.1 it follows that P is an ACP process **iff** it is a fixed point of the monotonic idempotent

$$\Phi \;=_{df}\; \mathbf{R} \circ and_B \;=\; and_B \circ \mathbf{R}$$

where

$$B =_{df} ((tr' = tr) \wedge wait' \vee (tr < tr'))$$

and **R** was defined in Theorem 8.0.2. That is why ACP processes form a complete lattice. We are going to show that **ACP1** is preserved by composition.

$$\begin{array}{lll} & (X;Y) \wedge (tr' = tr) & \{\mathbf{R1}\} \\ \Rightarrow & \exists s \bullet (X[s/tr'] \wedge tr = s) ; (Y[s/tr] \wedge tr' = s) & \{\mathbf{ACP1}\} \\ \Rightarrow & \exists s \bullet (wait'; wait') & \{\text{def of ;}\} \\ = & wait' & \Box \end{array}$$

Example 8.1.4 (Communication)

Let $a \in \mathcal{A}$. The process $do_{\mathcal{A}}(a)$ cannot refuse to perform an a-action at the very beginning. After performing that action, it terminates successfully. It therefore has exactly two stable states, denoted by

$$do_{\mathcal{A}}(a) =_{df} \Phi(a \notin ref' \lhd wait' \rhd tr' = tr^\frown < a >)$$

Because Φ (see Theorem 8.1.3) is idempotent and $do_{\mathcal{A}}(a)$ lies in its range, so $do_{\mathcal{A}}(a)$ is an ACP process. In future we will write just a instead of $do_{\mathcal{A}}(a)$. □

The deadlock process δ is a left zero of sequential composition.

L1 $\delta ; P = \delta$

Proof

$$\begin{array}{lll} & \delta; P & \{\text{Theorem 8.0.3 and } \mathbf{R3}\} \\ = & I\!I \lhd wait \rhd (\delta; P) & \{\text{def of } \delta\} \\ = & I\!I \lhd wait \rhd ((\delta \wedge wait'); P) & \{P \text{ meets } \mathbf{R3}\} \\ = & I\!I \lhd wait \rhd ((\delta \wedge wait'); I\!I) & \{\text{def of } \delta \text{ and } \mathbf{R3}\} \\ = & \delta & \Box \end{array}$$

Let P and Q be ACP processes. The notation $P+Q$ describes a process that may behave like P or like Q. It is distinguished from the disjunction $P \vee Q$ by the fact that the selection is made after the first event has happened rather than before. If the first event is one in which P cannot engage, then it must be one in which Q has engaged, and the rest of the behaviour of $P + Q$ is described by Q. Similarly, P can be selected by occurrence of an event possible for P but not for Q. Finally, if the first event is possible for both P and Q, then selection between them is fully non-deterministic. Since the choice of first event can be made by some other process running in parallel with both P and Q, $+$ is often called *external* choice, whereas $\sqcap$ denotes a choice made *internally*, as it were by the process itself.

The distinctive feature of $+$ is that initially, when tr' is still equal to tr, the process $P + Q$ cannot refuse a set of actions unless it is refused by *both* P and by Q. That explains its very simple truth functional definition, which permits the properties of $+$ to be proved in propositional calculus.

Definition 8.1.5 (Alternative composition)

$$P + Q \;=_{df}\; (P \wedge Q) \lhd \delta \rhd (P \vee Q) \qquad \square$$

Example 8.1.6

$$\begin{array}{lll} & a + b & \{\text{Def. 8.1.5}\} \\ = & (a \wedge b) \lhd \delta \rhd (a \vee b) & \{\text{Theorem 8.0.3}\} \\ = & \mathbf{R3}(a \wedge b) \lhd \delta \rhd \mathbf{R3}(a \vee b) & \{2.1\mathbf{L3}\} \\ = & \mathbf{R3}((a \wedge b) \lhd \delta \rhd (a \vee b)) & \{\text{Example 8.1.4}\} \\ = & \mathbf{R3}((ref' \cap \{a, b\}) = \{\} \lhd wait' \rhd (tr' - tr) \in \{< a >, < b >\}) & \square \end{array}$$

Alternative composition is idempotent, commutative and associative.

L2 $P + P \;=\; P$

L3 $P + Q \;=\; Q + P$

L4 $(P + Q) + R \;=\; P + (Q + R)$

The deadlock process δ is unable to perform an action; thus it can never be chosen to be executed by alternative composition.

L5 $\delta + Q \;=\; Q$

An alternative composition makes its choice in accordance with the first events which its alternatives are offering to perform. Since no ACP process can terminate before executing an action, the successor of alternative composition cannot affect that choice. Hence the distributive law

L6 $(P + Q); R \;=\; (P; R) + (Q; R)$

This law is not valid if P is the unit II, and this is perhaps one reason why II is omitted from ACP.

The CCS $+$ operator is essentially the same as that of ACP, and the CCS NIL process plays the same role as δ. They obey all the algebraic laws quoted above. However, in CCS sequential composition is restricted to the case when the first component is a single action. Furthermore, this is the only context in which a single action may be written. Finally, dot is used instead of semicolon

$$a.P \qquad b.NIL \qquad a.(b.(c.NIL))$$

The laws **L2** to **L5** justify introduction of the notation

$$\Sigma_{i \in I} P_i$$

to represent the alternative composition of a family of i-indexed processes

$$\{P_i \mid i \in I\}$$

We also adopt the convention that

$$\delta \;=\; \Sigma_{i \in \{\}} P_i$$

This notation will be used to define a normal form for ACP processes.

Definition 8.1.7 (Normal form)

An ACP process

$$\Sigma_{i \in I} (a_i; P_i) \;+\; \Sigma_{j \in J} b_j$$

is said to be in *head normal form.* If all the P_i and their descendants are similarly expressed, the process is in *normal form* (which may therefore be infinite). □

The interleaving of ACP processes is defined in a manner similar to the interleaving of Example 7.2.2. Each action of the system $P|||Q$ is an action of exactly one of its components. If one of the processes cannot engage in the action, then it must be the other one, but if both processes could have engaged in the same action, the choice between them is non-deterministic. Conversely, $P|||Q$ can refuse an action only when it is refused by both the operands, and it terminates when both its operands have terminated.

Definition 8.1.8 (Interleaving)

$$P|||Q \;=_{df}\; P\|_{M_{ACP}}Q$$

where the merge relation M_{ACP} is defined

$$\begin{aligned} M_{ACP} \;=_{df}\; & (wait' \;=\; 0.wait \,\vee\, 1.wait) \,\wedge \\ & (ref' \;=\; 0.ref \,\cap\, 1.ref) \,\wedge \\ & (tr' - tr) \,\in\, (0.tr \;-\; tr)|||(1.tr \;-\; tr) \end{aligned}$$ □

Since the merge relation M_{ACP} is valid, from Theorem 7.2.11 it follows that the ACP interleaving operator $|||$ is commutative and associative (it cannot have a unit, because there isn't one in ACP). The following expansion law shows how to transform an interleaving composition into normal form.

L7 Let $P \;=\; (\Sigma_i (a_i; P_i) \;+\; \Sigma_j b_j)$ and $Q \;=\; (\Sigma_l (c_l; Q_l) \;+\; \Sigma_m d_m)$.

Then $P|||Q \;=\; \Sigma_i (a_i; (P_i|||Q)) + \Sigma_j (b_j; Q) + \; \Sigma_l (c_l; (P|||Q_l)) \;+\; \Sigma_m (d_m; P)$

The interleaving composition $P|||Q$ does not allow communication between processes P and Q, because their behaviour is completely independent. A different operator $||$ is therefore needed to allow processes to interact by synchronisation. Communication is accomplished when one of the two processes selects an alternative offered by a $+$ operator in the other.

In ACP, synchronisation is defined in considerable generality. For example, suppose a and b are two actions that can occur simultaneously, but when they do so, they actually result in some different action c. ACP postulates a partial function $(a|b)$ which maps pairs of synchronisable actions to the action that results from successful synchronisation. As in Example 7.2.5, it is postulated to be commutative and associative. When $(a, b) \notin dom(|)$, this indicates that no synchronisation between actions a and b is expected or intended. Let S and T be subsets of $\mathcal{A}$; we will adopt the following convention

$$S|T \ =_{df} \ \{(a|b) \mid (a, b) \in dom(|) \wedge a \in S \wedge b \in T\}$$

Definition 8.1.9 (Parallel composition)

The parallel composition $P||_{ACP}Q$ may at any time perform an independent action of P or an independent action of Q. Alternatively, it may perform an action synchronised between P and Q, whenever such a synchronisation is defined. It then behaves as the parallel composition of the remainders of P and Q

$$P||_{ACP}Q \ =_{df} \ P||_N Q$$

where

$$\begin{aligned} N \ =_{df} \ & (wait' = 0.wait \vee 1.wait) \wedge \\ & (tr' - tr) \in (0.tr - tr) \ || \ (1.tr - tr) \wedge \\ & ref' \subseteq 0.ref \cap 1.ref - ((\mathcal{A} \setminus 0.ref)|(\mathcal{A} \setminus 1.ref)) \end{aligned}$$

and the trace merge operator $||$ is defined by

$$\begin{aligned} u|| <> \ &=_{df} \{u\} \\ <> ||u \ &=_{df} \{u\} \\ (< a >^\frown u)||(< b >^\frown v) \ &=_{df} \ < a >^\frown(u||(< b >^\frown v)) \cup < b >^\frown((< a >^\frown u)||v) \cup \\ & \quad < a|b >^\frown((u||v) \lhd (a, b) \in dom(|) \rhd \{\}) \end{aligned} \qquad \Box$$

The three disjuncts in the last definition represent the three possibilities that a or b may occur independently without mutual synchronisation, or that they may occur simultaneously (provided that they are synchronisable).

This parallel composition of ACP is a generalisation of the parallel composition $|$ of CCS, where events are classified into

(1) passive events a, b, c, ...

(2) active events, which have the same name as passive events, but are written with an overbar $\bar{a}$, $\bar{b}$, $\bar{c}$, ...

(3) a special event τ, which is supposed to occur silently and invisibly, and without synchronisation with any other event.

Synchronisation occurs only between a passive event and an active event with the same name, and the resulting event is always silent. So the CCS definition of parallelism may be obtained from the ACP definition by a special choice of synchronising function

$$(x|\bar{x}) = \tau, \qquad\qquad \text{for all } x$$

Like the interleaving operator, the ACP parallel operator is commutative and associative. The following complicated expansion law enables us to convert a parallel composition into a head normal form.

L8 Let $P = (\Sigma_i\,(a_i; P_i) + \Sigma_j\, b_j)$ and $Q = \Sigma_l\,(c_l; Q_l) + \Sigma_m\, d_m$,

and let P_i and Q_l be head normal forms.

$$\begin{aligned}\text{Then } P||_{ACP}Q \;=\; & \Sigma_i\,(a_i; (P_i||_{ACP}Q)) + \Sigma_j\,(b_j; Q) + \\ & \Sigma_l\,(c_l; (P||_{ACP}Q_l)) + \Sigma_m\,(d_m; P) + \\ & \Sigma_{i,l}\,((a_i|c_l); (P_i||_{ACP}Q_l)) + \Sigma_{i,m}\,((a_i|d_m); P_i) + \\ & \Sigma_{j,l}\,((b_j|c_l); Q_l) + \Sigma_{j,m}\,(b_j|d_m)\end{aligned}$$

The expansion law shows that whenever a synchronisation $(a|b)$ is possible, the separate actions a and b are also possible. These often represent an unsuccessful attempt to synchronise, which may be successful when additional parallel processes are added to the system. But after all the processes have been included, these unsuccessful attempts cannot now occur. Their removal is achieved by an encapsulation operator, which prevents them from happening: they are replaced by deadlock in those cases when all the actions have been prevented.

Definition 8.1.10 (Encapsulation)

Let $E \subseteq \mathcal{A}$. The process $\varrho_E(P)$ behaves like P, except that it always refuses to perform any action of E, and all traces containing them are simply omitted.

$$\varrho_E(P(ref')) =_{df} \mathbf{R3}((tr' - tr) \downarrow E = <> \wedge \exists X \bullet P(X) \wedge (ref' \subseteq X \cup E))$$

where $s \downarrow E$ stands for the subsequence of s consisting only of elements of E. □

The following law permits symbolic execution of the encapsulation operator by distributing it through a head normal form

L9 $\varrho_E(\Sigma_i\,(a_i; P_i) + \Sigma_j\, b_j) \;=\; \Sigma_{a_i \notin E}\,(a_i; \varrho_E(P_i)) + \Sigma_{b_j \notin E}\, b_j$

Theorem 8.1.11 (Closure of ACP processes)

ACP processes are closed under $+$, ϱ_E, $|||$ and $||$.

Proof From Theorem 8.1.3 it follows that the set of ACP processes is closed under $\vee$, $\wedge$ and $\lhd\delta\rhd$. The conclusion that ACP processes are closed under $+$ follows directly from Definition 8.1.5. For the encapsulation operator one has

$$\begin{array}{lll}
 & \Phi(\varrho_E(P)) & \{\text{def of } \Phi \text{ and } B \text{ in Theorem 8.1.3}\} \\
= & (\mathbf{R3} \circ \mathbf{R2} \circ and_B)(\varrho_E(P)) & \{\wedge - \vee \text{ distributivity}\} \\
= & (\mathbf{R3} \circ \mathbf{R2})(\varrho_E(and_B(P))) & \{\text{Theorem 8.1.3 and idemp of } and_B\} \\
= & (\mathbf{R3} \circ \mathbf{R2})(\varrho_E(P)) & \{\varrho_E \text{ is disjunctive}\} \\
= & \mathbf{R3}(\varrho_E(\mathbf{R2}(P))) & \{\text{Theorem 8.1.3 and idemp of } \mathbf{R2}\} \\
= & \mathbf{R3}(\varrho_E(P)) & \{\text{idemp of } \mathbf{R3}\} \\
= & \varrho_E(P) &
\end{array}$$

The interleaving operator $|||$ preserves the conditions **R2** and **R3** because

$$\begin{array}{lll}
 & \mathbf{R3}(\mathbf{R2}(P|||Q)) & \{||| \text{ is disjunctive}\} \\
= & \mathbf{R3}(\mathbf{R2}(P)|||\mathbf{R2}(Q)) & \{P \text{ and } Q \text{ satisfy } \mathbf{R2}\} \\
= & \mathbf{R3}(P|||Q) & \{\text{def of } \mathbf{R3}\} \\
= & \mathit{II} \lhd wait \rhd (P|||Q) & \{\mathit{II}|||\mathit{II} = \mathit{II}\} \\
= & (\mathit{II}|||\mathit{II}) \lhd wait \rhd (P|||Q) & \{P \text{ and } Q \text{ satisfy } \mathbf{R3}\} \\
= & (P|||Q) \lhd wait \rhd (P|||Q) & \{2.1\mathbf{L1}\} \\
= & P|||Q &
\end{array}$$

From the fact that $(tr \le tr')|||(tr \le tr') = (tr \le tr')$ we conclude that $|||$ also preserves **R1**.

In a similar way we can prove $||$ preserves the healthiness conditions. □

ACP does not model non-determinism directly, and does not include disjunction as an operator. Furthermore, there is no concept of an ordering defined between processes. Thus it is not possible to define recursion by means of a strongest or a weakest fixed point. Instead, ACP restricts the use of recursion to cases when it is known that there exists exactly one fixed point of the defining equation. The technique is to ensure by a textual check that a process always engages in a visible action before invoking a recursion. Thus no empty sequence of actions is possible. Every other trace has a finite length n, so it is always possible to tell whether that trace is possible for a recursively defined process by unfolding the recursion just n times.

Definition 8.1.12 (Guarded recursion)

Let X be a process variable, and let $P(X)$ be a process expression. $P(X)$ is *guarded* if it is a head normal form. Then the notation $\mu X_{\mathbf{Rea}} \bullet P(X)$ stands for the weakest reactive process satisfying the recursive equation

$$X \;=\; P(X) \qquad \Box$$

Theorem 8.1.13 (Unique fixed point of guarded recursion)

If $P(X)$ is guarded then $\mu X_{\mathbf{Rea}} \bullet P(X) \;=\; \nu X_{\mathbf{Rea}} \bullet P(X)$.

Proof Define for $n \geq 0$

$$E_n =_{df} \;(tr \leq tr') \wedge (\#tr' < \#tr + n)$$

Clearly $E = \{E_n \mid n \geq 0\}$ is an approximation chain (Definition 2.7.3) for

$$C = (tr \leq tr')$$

Because P is guarded we have

$$P(X) \wedge E_{n+1} = P(X \wedge E_n) \wedge E_{n+1}$$

for all X and n. From Theorem 2.7.6 it follows that

$$C \wedge \mu X_{\mathbf{Rea}} \bullet P(X) = C \wedge \nu X_{\mathbf{Rea}} \bullet P(X)$$

The conclusion follows directly from the healthiness condition (**R1**). □

Corollary

If $P(X)$ is guarded then $\mu X_{\mathbf{Rea}} \bullet P(X) \;= \Phi(\mu X_{\mathbf{Rea}} \bullet P(X))$.

Proof From the fact that **ACP** $\subseteq$ **Rea** it follows that

$$\nu X_{\mathbf{Rea}} \bullet P \sqsupseteq \nu_{\mathbf{ACP}}\, X \bullet P \sqsupseteq \mu_{\mathbf{ACP}}\, X \bullet P \sqsupseteq \mu X_{\mathbf{Rea}} \bullet P \;=\; \nu X_{\mathbf{Rea}} \bullet P \qquad \Box$$

Example 8.1.14 (Bag)

The guarded recursive equation

$$BAG \;=\; in.0; (out.0 ||_{ACP} BAG) \;+\; in.1; (out.1 ||_{ACP} BAG)$$

describes the process behaviour of a bag that may contain finitely many instances of data 0 and 1. The actions *in*.0 and *out*.0 are putting a 0 into the bag and getting a 0 from the bag respectively, and likewise for *in*.1 and *out*.1. □

Example 8.1.15 (Queue)

Let T be a finite set of data. The alphabet consists of read actions $r.d$ and write actions $w.d$ for $d \in T$. Consider the following family of guarded recursive equations

$$\begin{aligned} Q_{<>} &= \Sigma_{d\in T}(r.d\,;\, Q_{<d>}) \\ Q_{<d>^\frown u} &= w.d\,;\, Q_u + \Sigma_{e\in T}(r.e\,;\, Q_{<d>^\frown u^\frown <e>}) \end{aligned}$$

They specify the process behaviour of an unbounded buffer which reads and writes the data of the set T. What it writes is always an initial segment of what it reads; the balance is stored in a queue for later output on demand. □

Finally, we show how to convert a finite ACP process into normal form.

Theorem 8.1.16 (Normal form reduction)

Every finite ACP process can be transformed into head normal form algebraically.

Proof Let P and Q be normal forms. From Definition 8.1.8 it is clear that $P+Q$ is also in normal form. $P; Q$ can be transformed into normal form by using **L6**. Applying **L7** and **L8** allows us to convert $(P|||Q)$ and $(P||_{ACP}Q)$ into normal form respectively. **L9** indicates that $\varrho_E(P)$ can be rewritten as a normal form. □

8.2 Communicating Sequential Processes (CSP)

The theory of CSP [32, 78, 88] was developed to reason about the behaviour of a network of microprocessors, communicating along the wires which connect them. In addition, each microprocessor is capable of executing one or more normal sequential programs, with internal parallelism simulated by timesharing, and internal communication simulated by copying between storage locations. It was very important that the theory should maintain a level of abstraction at which the same program could be executed interchangeably on single processors or on multiple processors, without any discernible change to its logical behaviour. Only this makes it safe to improve performance of a correct program by adjusting the architecture, the number of processors, and the network of interconnections.

In order to accommodate sequential programs as part of a system of communicating processes, it was essential that CSP should allow a process that engages only on internal computations, even if it never engages in communication. The simplest example is $SKIP$, the unit of sequential composition. Such a process violates the healthiness condition **ACP1** of ACP. In addition, unification with the sequential language requires the addition of *ok* and *ok′*, to the alphabet, as well as the *wait* and *wait′* of ACP. But *ok′* and *wait′* can be both false, indicating that the process never reaches a stable state. This is what happens on divergence.

Since sequential programs can diverge, there is no reason to exclude this possibility for communicating processes. Indeed, it is no longer possible to rely on guarded recursions to ensure uniqueness of fixed points, and the general theory of weakest fixed points must be used. As a result, CSP is able to model the possibility of livelock, which occurs when a process expends an unlimited amount of resource on internal concealed communication, without ever making externally perceived progress, or ever reaching a stable or terminated state.

These are the main differences between CSP and ACP. But their similarities are much more significant. A process in CSP certainly conforms to the healthiness conditions **R1** to **R3** for reactive processes. Furthermore, it also conforms to the basic healthiness conditions **H1** and **H2** of sequential processes given in Section 3.2, that it makes no prediction about a process that has not started, and it is also monotonic in the variable ok'.

CSP1 $P = or_Q(P)$, where

$$Q =_{df} \neg ok \wedge (tr \leq tr')$$

CSP2 $P = post_J(P)$, where

$$J =_{df} (ok \Rightarrow ok') \wedge (v' = v) \wedge (tr' = tr) \wedge (wait' = wait) \wedge (ref' = ref)$$

Definition 8.2.1 (CSP process)

A reactive process P is a CSP process if it satisfies **CSP1** and **CSP2**. □

Theorem 8.2.2 (Closure of CSP processes)

CSP processes form a complete lattice, which is closed under sequential composition. It also contains $\mathbf{R}(x := e)$ where x is any list of stored program variables, and **R** has been defined in Theorem 8.0.2.

Proof The first conclusion follows from the fact that P is a CSP process **iff**

$$P = \mathbf{R}(\neg P[false/wait,\, false/ok'] \vdash P[false/wait,\, true/ok'])$$

The second conclusion can be proved in a similar way to Theorem 8.1.3. □

Examples 8.2.3

(1) The deadlock process $STOP$ is incapable of communicating with its environment, and always stays in a waiting state

$$STOP =_{df} \mathbf{CSP1}(ok' \wedge \delta) = \mathbf{R}(wait := true)$$

(2) The process $a \to SKIP$ waits to perform a communication event a at the very beginning. After engaging in that event it terminates successfully.

$$a \to SKIP \;=_{df}\; \mathbf{CSP1}(ok' \wedge do_{\mathcal{A}}(a))$$

where $do_{\mathcal{A}}(a)$ was defined in Example 8.1.4.

(3) The process $SKIP$ refuses to engage in any communication event, but terminates immediately.

$$SKIP \;=_{df}\; \mathbf{R}(\exists ref \bullet I\!I)$$

Existential quantification indicates that the initial value of ref is entirely irrelevant, as discussed again in Definition 8.2.16.

(4) Defined as the weakest fixed point of the monotonic mapping $\mathbf{R}$, $CHAOS$ is the worst CSP process.

$$CHAOS \;=_{df}\; \mathbf{R}(\mathbf{true})$$

Note that this is strictly stronger than **true**. Not even $CHAOS$ is allowed to undo events that have already happened. For this reason $CHAOS$ is not a right zero of sequential composition. □

Exercise 8.2.4

Prove that $\lhd STOP \rhd$ preserves **CSP1** but not **CSP2**. □

Both $CHAOS$ and $STOP$ are left zeros of sequential composition of CSP processes.

L1 $CHAOS; P \;=\; CHAOS$

L2 $STOP; P \;=\; STOP$

The CSP prefix operator is defined in the same way as in ACP. The process $a \to P$ is willing to engage in the event a at the very beginning. After performing that event it behaves like P.

Definition 8.2.5 (Prefix)

Let $a \in \mathcal{A}P$.

$$a \to P \;=_{df}\; (a \to SKIP); P$$
$$\mathcal{A}(a \to P) \;=_{df}\; \mathcal{A}P$$
□

CSP is closed under the prefix operator $a \to$ because it is closed under sequential composition. Clearly, the prefix operator is disjunctive.

Example 8.2.6 (Clock)

Consider a perpetual clock which never does anything but tick

$$\mathcal{A}(CLOCK) \;=_{df}\; \{tick\}$$

Consider next a process that behaves exactly like the clock, except that it first emits a single tick

$$tick \rightarrow CLOCK$$

The behaviour of this process is indistinguishable from that of the original clock. This reasoning leads to the following definition of $CLOCK$

$$CLOCK \;=\; \mu X_{\mathbf{CSP}} \bullet (tick \rightarrow X) \qquad \square$$

The external choice operator of CSP is denoted by $[\!]$. In order to deal with $SKIP$ and $CHAOS$, its definition is slightly more complicated than that of ACP. In particular, if either operand is $CHAOS$, so is the result, and if either of them is $SKIP$, the result may non-deterministically be $SKIP$. These effects are achieved simply by applying the relevant healthiness condition.

Definition 8.2.7 (External choice)

Let $\mathcal{A}P \;=\; \mathcal{A}Q$. Then

$$P[\!]Q \;=_{df}\; \mathbf{CSP2}((P \wedge Q) \lhd STOP \rhd (P \vee Q)) \qquad \square$$

Theorem 8.2.8

The set of CSP processes is closed under the external choice operator $[\!]$.

Proof From the idempotency of the embedding **CSP2** and Exercise 8.2.4. □

The external choice operator $[\!]$ enjoys almost the same algebraic laws as the ACP $+$ operator: it is idempotent, commutative and associative, and has the deadlock process $STOP$ as its unit. Furthermore, it distributes over $\sqcap$ and the other way round.

L3 $(P \sqcap Q)[\!]R \;=\; (P[\!]R) \sqcap (Q[\!]R)$

L4 $P \sqcap (Q[\!]R) \;=\; (P \sqcap Q)[\!](P \sqcap R)$

If one component can terminate at the very beginning, then the alternative composition may terminate immediately.

L5 $P[\!]SKIP \;\sqsubseteq\; SKIP$

Let $E = \{a_1, \ldots, a_n\}$ be a set of events. We use the notation

$$x : E \rightarrow P(x)$$

to denote *guarded choice*

$$(a_1 \rightarrow P(a_1)) \| (a_2 \rightarrow P(a_2)) \| \ldots \| (a_n \rightarrow P(a_n))$$

Sequential composition distributes over guarded choice.

L6 $(x : E \rightarrow P(x)); Q \quad = \quad x : E \rightarrow (P(x); Q)$

However, sequential composition does not in general distribute backward through the external choice operator in CSP, because one of its components may be $SKIP$, which by **L5** may occur non-deterministically. The relevant law is

$$((a \rightarrow SKIP) \| SKIP); (b \rightarrow Q) \quad = \quad ((a \rightarrow b \rightarrow Q) \| (b \rightarrow Q)) \ \sqcap \ (b \rightarrow Q)$$

If the environment refuses a, the only possibility is $(b \rightarrow Q)$. Otherwise, the event a may happen, but it does not have to; $(b \rightarrow Q)$ may be chosen instead, which will result in deadlock if the environment refuses b.

Example 8.2.9

A copying process engages in the following events

in.0	input of zero on its input channel
in.1	input of one on its input channel
out.0	output of zero on its output channel
out.1	output of one on its output channel

Its behaviour consists of a repetition of pairs of events. On each cycle, it inputs a bit and outputs the same bit

$$COPYBIT \ =_{df} \ \mu X_{\mathbf{CSP}} \bullet (in.0 \rightarrow out.0 \rightarrow X \,\|\, in.1 \rightarrow out.1 \rightarrow X)$$

Note how this process allows its environment to choose which value should be input, but no choice is offered in the case of output. This distinction between input and output will be developed in Section 8.3. □

The CSP interleaving operator is defined in the same way as its ACP counterpart: it joins processes with the same alphabet to be executed concurrently without interacting or synchronisation. In this case, each action of the system is an action of exactly one process. An interleaving composition enters a divergent state when one of its components becomes chaotic. In that case, the only sure fact is that initial value of the trace cannot be shortened.

Definition 8.2.10 (Interleaving)

Let $\mathcal{A}P = \mathcal{A}Q$. Define

$$\begin{aligned} \mathcal{A}(P|||Q) &=_{df} \mathcal{A}P = \mathcal{A}Q \\ P|||Q &=_{df} (P\|_M Q) \end{aligned}$$

where the merge operation M is defined by

$$M =_{df} (M_{ACP} \wedge ok' = (0.ok \wedge 1.ok)); SKIP$$

where M_{ACP} was defined in Definition 8.1.8. □

Theorem 8.2.11

The set of CSP processes is closed under the interleaving operator $|||$.

Proof

$$\begin{aligned} & P|||Q && \{\text{Def. 8.2.10}\} \\ = \; & P \|_M Q && \{P \text{ and } Q \text{ satisfy } \mathbf{CSP1}\} \\ = \; & \mathbf{R}(ok \Rightarrow P) \|_M \mathbf{R}(ok \Rightarrow Q) && \{7.1\mathbf{L5}\text{–}\mathbf{L6}\} \\ = \; & \mathbf{R}(\mathbf{R1}(ok \Rightarrow P)\|_M \mathbf{R1}(ok \Rightarrow Q)) && \{\|_M \text{ is disjunctive}\} \\ = \; & \mathbf{R}((\mathbf{R1}(\neg ok)|||\mathbf{R1}(\neg ok)) \vee (P|||Q)) && \{\text{Def. 8.2.10}\} \\ = \; & \mathbf{R}(\mathbf{R1}(\neg ok) \vee (P|||Q)) && \{\mathbf{R1} \text{ is idemp}, \mathbf{R} \text{ is disj}\} \\ = \; & \mathbf{R}(ok \Rightarrow (P|||Q)) && \square \end{aligned}$$

Interleaving composition is commutative and disjunctive, and has the following expansion law

L7 Let $P = (x : A \to P(x))$ and $Q = (y : B \to Q(y))$.

Then $P|||Q = (x : A \to (P(x)|||Q)) \,[\!]\, (y : B \to (P|||Q(y)))$

CSP parallel composition is used to model interaction and synchronisation between concurrent processes. The parallel system $P\|_{CSP}Q$ executes P and Q in such a way that actions in the alphabet of both components require simultaneous participation of them both, whereas the remaining events of the system occur in an arbitrary interleaving. The system terminates after both P and Q have terminated successfully, and it becomes divergent after either one of its components does so.

Definition 8.2.12 (Parallel composition)

$$\begin{aligned} \mathcal{A}(P\|_{CSP}Q) &=_{df} \mathcal{A}P \cup \mathcal{A}Q \\ P\|_{CSP}Q &=_{df} P\|_N Q \end{aligned}$$

where

$$\begin{aligned} N =_{df}\ & (ok' = (0.ok \wedge 1.ok) \wedge \\ & wait' = (0.wait \vee 1.wait) \wedge \\ & ref' = (0.ref \cup 1.ref) \wedge \\ & \exists u \bullet (u \downarrow \mathcal{A}P = (0.tr - tr) \ \wedge \ u \downarrow \mathcal{A}Q = (1.tr - tr) \wedge \\ & \qquad u \downarrow \mathcal{A}(P \| Q) = u \ \wedge \ tr' = tr^\frown u)) \ ; \\ & SKIP \end{aligned}$$ □

Like the interleaving operator, the CSP parallel operator is commutative and disjunctive.

Theorem 8.2.13

The set of CSP processes is closed under the parallel operator $\|_{CSP}$.

Proof Similar to Theorem 8.2.11. □

Let E be a set of events which are regarded as internal events of a process P; for example, they may be interactions between component processes from which P is composed. We want the events of E to occur silently and automatically whenever they can, without the participation or even the knowledge of the environment of P. We use

$$P \backslash E$$

to denote the resultant process. $P \backslash E$ can reach a stable state only when P is stable and unable to perform any event of E; if any internal event can happen, P is obviously unstable until after it does. $P \backslash E$ becomes divergent when P does, and it terminates when P has terminated successfully.

Definition 8.2.14 (Hiding)

$$\begin{aligned} \mathcal{A}(P \backslash E) &=_{df} \mathcal{A}P - E \\ P(tr', ref') \backslash E &=_{df} \mathbf{R}(\exists s \bullet P(s, E \cup ref') \wedge L) \, ; SKIP \end{aligned}$$

where $\quad L =_{df} (tr' - tr) = (s - tr) \downarrow (\mathcal{A}P - E)$ □

Like the abstraction operator in ACP, the CSP hiding operator is associative and disjunctive. Furthermore, it does not affect the behaviour of $SKIP$ and $CHAOS$, and it distributes over prefixing.

L8 $(a \rightarrow Q) \backslash E \ = \ a \rightarrow (Q \backslash E), \quad$ if $a \notin E$

Examples 8.2.15

(1) $(\mu X_{\mathbf{CSP}}(a \rightarrow X)) \backslash \{a\} \ = \ CHAOS$

(2) $(\bigsqcap_n P_n) \backslash \{a\} \ = \ STOP$, where $P_0 = STOP$ and $P_{n+1} = a \rightarrow P_n$. □

In the proof of Example 8.2.15(1) we need to allow tr to range over infinite traces as well as finite. The infinite trace consisting of all the a is not ruled out by the weakest fixed point $\mu X \bullet (a \to X)$, but the value $ok' = false$ is also not ruled out. This possibility of divergence is turned into the actual $CHAOS$ by the $SKIP$ which is included in Definition 8.2.14.

Theorem 8.2.16

The set of CSP processes is closed under the hiding operator $\backslash E$.

Proof From Definition 8.2.14 it follows that the hiding operator preserves the healthiness condition **CSP2**.

$$\begin{array}{ll}
P(tr', ref')\backslash E & \{\text{Def. } 8.2.14\} \\
= \mathbf{R}(\exists s \bullet P(s, E \cup ref') \land L); SKIP & \{P \text{ satisfies } \mathbf{CSP1}\} \\
= \mathbf{R}(\exists s \bullet \mathbf{R}(ok \Rightarrow P)(s, E \cup ref') \land L); SKIP & \{\mathbf{R1},\ \mathbf{R3} \text{ are idemp and disj}\} \\
= \mathbf{R}(\exists s \bullet \mathbf{R2}(ok \Rightarrow P)(s, E \cup ref') \land L); SKIP & \{\neg ok \text{ and } P \text{ satisfy } \mathbf{R2}\} \\
= \mathbf{R}(\exists s \bullet (ok \Rightarrow P)(s, E \cup ref') \land L); SKIP & \{\text{Theorem 8.0.3 and def of } L\} \\
= \mathbf{R}((\neg ok); SKIP \lor P(tr', ref')\backslash E) & \{(\neg ok); SKIP \ = \ \neg ok\} \\
= \mathbf{CSP1}(P(tr', ref')\backslash E) & \square
\end{array}$$

In spite of all the healthiness conditions introduced above for CSP, there are still many laws of the standard theory of CSP that cannot be proved when the operands are general predicates rather than programmable processes. In the remainder of this section we will present many of the additional healthiness conditions that are required. Adopting the technique of Chapter 3 used in describing healthiness conditions of the sequential programming language, we will introduce a set of monotonic weakening idempotent mappings, each of which embeds CSP processes into a complete lattice of processes with a given healthiness condition. We will also show how each individual healthiness condition can be characterised by a specific algebraic law.

Definition 8.2.17

A CSP process P is **CSP3** if its behaviour does not depend on the initial value of ref, that is it satisfies

$$\mathbf{CSP3} \quad \neg wait \ \Rightarrow \ (P \ = \ \exists ref \bullet P)$$

The antecedent $\neg wait$ is necessary for the same reason as the healthiness condition (**R3**). $\square$

Examples 8.2.18

$SKIP$, $STOP$, $CHAOS$ and $a \to P$ are **CSP3**. □

Theorem 8.2.19

P is **CSP3** **iff** it satisfies the left unit law

L9 $SKIP; P \;=\; P$ □

CSP3 is not only a desirable condition in itself; it is also needed to prove the important expansion law for $||$, and other obvious distribution laws for hiding.

Theorem 8.2.20 (Algebraic properties of **CSP3** processes)

Let $P(x)$ and $Q(y)$ and U be **CSP3**, for all x and y.
Let $P \;=\; (x : A \to P(x))$ and $Q \;=\; (y : B \to Q(y))$. Then

L10 $$P||_{CSP}Q \;=\; (x : (A - B) \to (P(x)||_{CSP}Q)) \,[\!]\, (y : (B - A) \to (P||_{CSP}Q(y))) \,[\!]\, (z : (A \cap B) \to (P(z)||_{CSP}Q(z)))$$

L11 $(a \to U)\backslash E \;=\; U\backslash E, \quad \text{if } a \in E$

L12 Let $A \cap E \;=\; \{\}$ and $B \;\subseteq\; E$ and $S \;=\; \sqcap_{y \in B}(Q(y)\backslash E)$.

Then $(P[\!]Q)\backslash E \;=\; ((x : A \to (P(x)\backslash E))[\!]S) \sqcap S$

As described before (under **L6**), the environment cannot force the occurrence of an event a in A, unless it is acceptable to S as well. In that case, choice between S and $P(x)$ is non-deterministic. □

Definition 8.2.21 (Right unit)

P is **CSP4** if it satisfies the right unit law

CSP4 $P; SKIP = P$ □

Example 8.2.22

$STOP$, $SKIP$, $CHAOS$ and $a \to SKIP$ are **CSP4**. □

Theorem 8.2.23 (Algebraic properties of **CSP4** processes)

Let P be **CSP4**, and let S and T be **CSP3** and **CSP4**. Then

L13 $P\backslash\{\} \;=\; P$

L14 $(S; T)\backslash E \;=\; (S\backslash E); (T\backslash E)$ □

The next healthiness condition says that if a process is deadlocked, and refusing some set of events offered by its environment, then it would clearly still be deadlocked in an environment which offers even fewer events.

Definition 8.2.24 (Refusal subset-closed)

P is **CSP5** if it satisfies the unit law of interleaving

CSP5 $P|||SKIP \;=\; P$ □

Exercise 8.2.25

Show that P satisfies **CSP5 iff** it is a fixed point of the mapping $\mathbf{R3} \circ post_K$, where

$$K \;=_{df}\; \mathbf{R1}((\mathbf{true} \vdash (ref' \subseteq ref))_{+\{v,\, tr,\, wait\}})$$ □

Theorem 8.2.26 (Closure of healthy processes)

Processes of **CSPi** (for **i** = 3, 4, 5) form a complete lattice, and all CSP operators preserve these healthiness conditions.

Proof The first conclusion comes from the fact that the mappings

$$\begin{aligned}
\mathbf{CSP3} &=_{df} pre_{SKIP}\\
\mathbf{CSP4} &=_{df} post_{SKIP}\\
\mathbf{CSP5} &=_{df} \mathbf{R3} \circ post_K
\end{aligned}$$

(which are used to characterise the healthiness conditions) are monotonic.

It is clear from the definitions of **CSP3**, **CSP4** and **CSP5** that sequential composition respects the healthiness conditions **CSP3** to **CSP5**.

The healthiness condition **CSP3** is preserved by $[\!]$, $|||$, $||_{CSP}$ and $\backslash E$ because they are disjunctive.

Notice that

$$((ok \Rightarrow ok') \wedge (v' = v)); SKIP \;=\; SKIP; ((ok \Rightarrow ok') \wedge (v' = v)) \quad (\dagger)$$

$$\begin{aligned}
& (P[\!]Q); SKIP && \{(\dagger)\}\\
=\; & \mathbf{CSP2}((P \wedge Q \lhd STOP \rhd P \vee Q)\,;\, SKIP) && \{STOP; SKIP = STOP\}\\
=\; & \mathbf{CSP2}((P \wedge Q) \lhd STOP \rhd ((P \vee Q); SKIP)) && \{P \text{ and } Q \text{ satisfy } \mathbf{CSP4}\}\\
=\; & \mathbf{CSP2}((P \wedge Q) \lhd STOP \rhd (P \vee Q)) && \{\text{Def. 8.2.6}\}\\
=\; & P[\!]Q
\end{aligned}$$

From the fact $SKIP; SKIP = SKIP$ it follows that $|||$, $||$ and $\backslash E$ also preserve **CSP4**.

Assume that P and Q satisfy **CSP5**. From the definition of N we can show

$$N; K = (0.K \| 1.K); N \qquad (\ddagger)$$

$$\begin{array}{ll} & \mathbf{R3}((P\|_{CSP}Q); K) & \{\text{Def. } 8.2.12\} \\ = & \mathbf{R3}((P\|_{N}Q); K) & \{(\ddagger)\} \\ = & \mathbf{R3}(((P; U0; 0.K)\|(Q; U1; 1.K)); N) & \{U0; 0.K = K; U0\} \\ = & \mathbf{R3}((P; K)\|_{N}(Q; K)) & \{P,\ Q \text{ satisfy } \mathbf{CSP5}\} \\ = & \mathbf{R3}(P\|_{N}Q) & \{\text{Theorem } 8.2.13\} \\ = & (P\|_{CSP}Q) & \end{array}$$

In a similar way we can also show the other finite CSP operators respect **CSP5**.

From the above proof we conclude that **CSP3**, **CSP4** and **CSP5** are all $\Sigma_{\sqsupseteq}$ links. From Theorem 4.1.15 it follows that the recursion also preserves the healthiness conditions **CSP3** to **CSP5**. □

We have omitted a number of important healthiness conditions, including the most familiar of all: that the set of traces of a process is *non-empty* and *prefix-closed*. For a full treatment see [160].

8.3* Data flow

A data flow network is one in which the channels along which the processes communicate are capable of buffering an arbitrary number of messages between the time that they are output and the time that they are needed for input. As a result, a network as a whole, when in a stable state, can never refuse to accept input (because all its components are waiting for input), nor will it ever wait to perform output (because its output channels are all buffered). Since we want to compose larger networks from smaller ones, we require all subnetworks, right down to the individual processes, to share the same properties. Among other benefits, this convention allows a distributed implementation to store the buffered messages either in the outputting processor or in the inputting processor, or partly in both.

The simplest possible example of a data flow process is a buffer $BUF_{\{a:c\}}$ with an input channel a and an output channel c. The buffer is at all times ready to accept a message from its input channel a, and (whenever possible) is ready to deliver to its output channel c the earliest message which has been input but not yet output. When there is no message left for output, the process $BUF_{\{a:c\}}$ enters a stable state in which it remains inactive unless it receives more input. In fact, each channel of a data flow network behaves just like this, so the only point of a buffer is to connect processes which use different names a and c for the channel along which they communicate. A more interesting process is one that applies some useful function to each message before sending it on (see Example 8.3.1(3)).

In previous sections we have introduced a general concept of an event as an action without duration, whose occurrences may require simultaneous participation by more than one independently described process. In this section we will concentrate on a special class of event known as a communication. A communication is an event that is described by a pair

$$c.m$$

where c is the name of channel on which the communication takes place and m is the value of the message which passes. The set of all messages which P can communicate on channel c is defined

$$\mathcal{A}_c(P) \;=_{df}\; \{m \mid c.m \in \mathcal{A}P\}$$

We also define functions which extract channel and message components of a communication

$$channel(c.m) = c \qquad\qquad message(c.m) = m$$

We will use the notation $inchan(P)$ to denote the set of channels used for input by P, and the notation $outchan(P)$ for the set of channels used for output by P respectively. Any channel can only be used in a single process either for input or for output, but not for both

$$inchan(P) \cap outchan(P) \;=\; \{\}$$

Example 8.3.1 (Output and input)

(1) Let $m \in \mathcal{A}_c(P)$. A process which first outputs m on the channel c and then behaves like P is defined

$$c!m \to P \;=_{df}\; c.m \to P$$

(2) A process which is initially prepared to input any value x communicable on the channel c, and then behaves like $P(x)$, is defined

$$c?x \to P(x) \;=_{df}\; (y : \{y | channel(y) = c\} \to P(message(y)))$$

(3) *Inc* applies the function $\lambda x \bullet (x+1)$ to each input message from channel d before outputting it to channel a. It behaves chaotically if the environment sends it two consecutive messages before it has completed the treatment of the first one.

$$Inc \;=\; \mu X_{\mathbf{CSP}} \bullet (d?x \to ((a!(x+1) \to X) [] (d?x \to CHAOS)))$$

This example models a communication channel (like a simple wire in hardware) which is incapable of buffering more than one message. The designer has to take the responsibility for avoiding buffer overflow. This general model has been developed as a theory for asynchronous circuit design [105, 181]. □

Let P and Q be CSP processes, and let c be a channel used for output by P and for input by Q. When these processes are composed concurrently in the system $P\|_{CSP}Q$, a communication $c.m$ can occur only when both processes engage simultaneously in that event, that is whenever P outputs a value m on the channel c, and Q simultaneously inputs the same value. An inputting process is prepared to accept any communicable value, so it is the outputting process that determines which actual message value is transmitted on each occasion, as in Example 8.2.9. Thus output may be regarded as a specialised case of the prefix operator, and input is a special case of choice; this leads to the law

L1 $(c!m \rightarrow Q)\|_{CSP}(c?x \rightarrow P(x)) \ = \ c!m \rightarrow (Q\|_{CSP}P(v))$

Note that $c!m$ remains on the right hand side of this equation as an observable action in the behaviour of the system. It is available for input by another process added to the system later. Multiple synchronised input of a single output is an occasionally useful feature of CSP. But when no more inputting processes are needed, such a communication event can be concealed by applying the hiding operator described in Section 8.2. Obviously, this can be done only outside the parallel composition of the two processes which communicate on the same channel. The effect is shown by the law

L2 $((c!m \rightarrow Q)\|_{CSP}(c?x \rightarrow P(x)))\backslash C \ = \ (Q\|_{CSP}P(m))\backslash C$
where $C \ =_{df} \ \{c.m \mid m \in \mathcal{A}_c\}$

Example 8.3.2 (Buffer)

$BUF_{\{a:c\}}$ is at all times ready to input a message on the channel a, and to output on the channel c the first message which it has input but not yet output

$$\begin{aligned} inchan(BUF_{\{a:c\}}) &=_{df} \{a\} \\ outchan(BUF_{\{a:c\}}) &=_{df} \{c\} \\ BUF_{\{a:c\}} &=_{df} BUF_{\{a:c\}}(<>) \end{aligned}$$

where

$$\begin{aligned} BUF_{\{a:c\}}(<>) &= a?x \rightarrow BUF_{\{a:c\}}(< x >) \\ BUF_{\{a:c\}}(< x > {}^\frown s) &= a?y \rightarrow BUF_{\{a:c\}}(< x > {}^\frown s {}^\frown < y >) \\ & \quad [\!] \ c!x \rightarrow BUF_{\{a:c\}}(s) \end{aligned}$$

The subscript of the buffer will be dropped if it is clear from the context. □

We now define a restricted form of parallel composition, in which data can flow only from the left operand to the right. Let P and Q be processes satisfying the following three conditions

(1) none of the output channels of Q is used for input by P, that is

$$outchan(Q) \cap inchan(P) = \{\}$$

(2) there is no input channel shared by P and Q

$$inchan(P) \cap inchan(Q) = \{\}$$

(3) no channel is used for output by both P and Q

$$outchan(P) \cap outchan(Q) = \{\}$$

The processes P and Q can be joined together so that the output channels of P are connected to the like-named input channels of Q, and the messages output by P and input by Q on these internal channels are concealed from the environment. The result of the connection is denoted

$$P \gg Q$$

and may be pictured as the series shown in Figure 8.3.3.

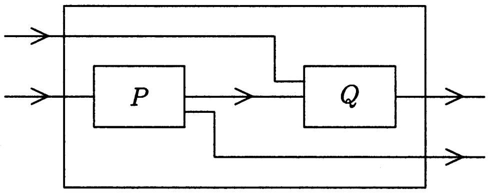

Figure 8.3.3 $P \gg Q$

Definition 8.3.4 (Chain)

$$\begin{aligned} inchan(P \gg Q) &=_{df} inchan(P) \cup (inchan(Q) \setminus outchan(P)) \\ outchan(P \gg Q) &=_{df} outchan(Q) \cup (outchan(P) \setminus inchan(Q)) \\ P \gg Q &=_{df} (P \|_{CSP} Q) \backslash C \end{aligned}$$

where $C =_{df} \bigcup \{c.m \mid m \in \mathcal{A}_c \wedge c \in outchan(P) \cap inchan(Q)\}$ □

Like sequential composition, the chaining operator is associative and disjunctive.

L3 $(P \gg Q) \gg R = P \gg (Q \gg R)$

L4a $(P_1 \sqcap P_2) \gg Q = (P_1 \gg Q) \sqcap (P_2 \gg Q)$

L4b $P \gg (Q_1 \sqcap Q_2) = (P \gg Q_1) \sqcap (P \gg Q_2)$

The following laws are useful in deriving further properties of BUF and the chaining operator.

L5 If any two of P, Q, $P \gg Q$ are buffers, then so is the third.

L6 If $P_s \gg Q_s$ is a buffer with an input channel a and an output channel c for all $s \in S$, then for any function $g : \mathcal{A}_a \to S$ the process

$$a?x \to (P_{g(x)} \gg (c!x \to Q_{g(x)}))$$

is a buffer.

If a buffer holds a message, then either an input from the input channel or the output of the stored message may happen first.

L7 $BUF(<x>) \;=\; BUF \gg (c!x \to BUF)$

Proof

$$\begin{aligned} & a?x \to BUF(<x>) && \{\text{def of } BUF\} \\ = \;& BUF && \{\textbf{L5} \text{ and } \textbf{L6}\} \\ = \;& a?x \to (BUF \gg (c!x \to BUF)) && \square \end{aligned}$$

If P is a CSP process, a buffer can be attached to each of its input and output channels, giving a result which is a data flow process. The buffer ensures that it is willing to receive input at any time, and its output channels can buffer an arbitrary number of outgoing messages. Using such a buffer interface we are able to model data flow processes as a simple closed class of CSP processes.

Definition 8.3.5 (Data flow)

A CSP process P is a data flow process if it satisfies the defining equation $P = [P]$, where

$$[P] =_{df} IN \gg P' \gg OUT$$

where IN and OUT are families of buffers attached to channels of P

$$\begin{aligned} IN &=_{df} \|_{CSP}\{BUF_{\{a:a'\}} \mid a \in inchan(P)\} \\ OUT &=_{df} \|_{CSP}\{BUF_{\{c':c\}} \mid c \in outchan(P)\} \end{aligned}$$

and the process P' behaves the same as P except that all channel names are decorated with a dash. We adopt the convention $(c')' = c$, so $(P')' = P$.

The defining equation is known as the Foam Rubber Wrapper postulate when used to characterise delay-insensitive circuits [105, 181]. For convenience the decoration $'$ in the defining equation will be dropped in the later discussion. $\square$

Because all input channels are buffered, a data flow process can never refuse an input. However, a stable process with an empty output buffer will refuse an output on that channel. When all its output buffers are empty, we say that a stable process is *quiescent.* A process reaches a quiescent state when it has finished all the tasks that are currently required of it; it will do no more until it has input some new task. For example, a buffer is quiescent just when it is empty, because then it has output everything that it has input. A process can be conveniently specified by just describing its quiescent states. We regard a process that never reaches a quiescent state to be like one that is livelocked or divergent.

Example 8.3.6

(1) The process $BUF(< m >)$ (which behaves like $BUF_{\{a:c\}}$ after receipt of a message m from input channel a) is a data flow process because

$$\begin{array}{lll} & BUF(< m >) & \{\mathbf{L5}\text{ and }\mathbf{L7}\} \\ = & BUF \gg BUF(< m >) & \{\mathbf{L2}\text{ and }\mathbf{L5}\} \\ = & BUF \gg (!m \rightarrow BUF) \gg BUF & \{\mathbf{L5}\text{ and }\mathbf{L7}\} \\ = & BUF \gg BUF(< m >) \gg BUF & \end{array}$$

(2) The process $SINK$ never performs any output (but of course must always accept input). It is always quiescent, and represents the deadlocked data flow process

$$SINK =_{df} IN \gg STOP \qquad \square$$

Theorem 8.3.7 (Closure of data flow processes)

Data flow processes form a complete lattice, which is closed under sequential composition, $\gg$ and $\backslash$.

Proof Here we are going to show that data flow machines are closed under $\gg$.
Let $X =_{df} outchan(P) - inchan(Q)$ and $Y =_{df} inchan(Q) - outchan(P)$.

$$\begin{array}{lll} & IN \gg (P \gg Q) \gg OUT & \{\text{Def. 8.3.5}\} \\ = & (\|_{CSP}\{BUF_{\{a:a'\}} \mid a \in inchan(P)\}) \gg P & \\ & \gg (\|_{CSP}\{BUF_{\{c':c\}} \mid c \in X\}) \gg (\|_{CSP}\{BUF_{\{a:a'\}} \mid a \in Y\}) & \\ & \gg Q \gg (\|_{CSP}\{BUF_{\{c':c\}} \mid c \in outchan(Q)\}) & \{\mathbf{L5}\} \\ = & P \gg Q & \square \end{array}$$

Note that $SKIP$ is not a data flow process. The problem cannot be solved by closure, because

$$[SKIP] = SINK$$

and so every terminated program would behave like $SINK$. Furthermore, sequential composition becomes void since for all data flow processes P and Q

$$P;Q = P$$

Data flow processes are not even closed under input or output (Example 8.3.1). This is because in CSP all communications are synchronised, whereas in data flow outputs can occur instantaneously. This effect is modelled in the following definition by reclosure after prefixing. From now on, we will use the word process simply to mean a data flow process.

Definition 8.3.8 (Output prefix)

Let P be a process and let $c \in outchan(P)$. The process $c!m \rightsquigarrow P$ behaves like P, but with one more message m inserted in front of the output buffer on channel c. It is not quiescent, and will not become so until that message has been accepted by the environment.

$$c!m \rightsquigarrow P \;=_{df}\; (c!m \rightarrow P) \gg OUT$$

Here we use a curly arrow notation $\rightsquigarrow$ to distinguish buffered communication from the fully synchronised type. □

The order in which a process transmits messages on distinct channels is not significant, because it does not determine the order in which the messages are received by the environment.

L8 $c!m \rightsquigarrow (d!n \rightsquigarrow P) \;=\; d!n \rightsquigarrow (c!m \rightsquigarrow P), \qquad \text{if } c \neq d$

As in CSP, output prefix is distributive.

L9 $c!m \rightsquigarrow (P \sqcap Q) \;=\; (c!m \rightsquigarrow P) \sqcap (c!m \rightsquigarrow Q)$

We have defined output prefix in a way that allows a process that is capable of diverging to corrupt any value buffered by its output channel, before the environment has received it. As a result, output prefix is strict

L10 $c!m \rightsquigarrow CHAOS \;=\; CHAOS$

Examples 8.3.9

(1) The process $c!0 \rightsquigarrow SINK$ outputs 0 once on channel c, and then deadlocks.

(2) The process $\mu X \bullet (c!0 \rightsquigarrow X)$ is equivalent to $CHAOS$. This gives a clear warning of the danger that the process can overflow any finite amount of buffering that it may be given. □

Example 8.3.9(2) illustrates an obligation placed upon the programmer to fix a bound on the amount of data that needs to be stored in the buffer before quiescence. So a process that inputs only a finite number of times must not output infinitely often. Alternative models of data flow incorporate a concept of fairness, which places the obligation to avoid infinite buffer filling upon the implementor [?]. But it is not easy to combine this kind of infinite fairness with non-determinism, and we shall not do it.

The definition of input is similar to that for output. However, it starts in a quiescent state, and will remain so until it accepts a message on channel c. Messages on other channels will be accepted, but they will be simply buffered for later consumption. That is why closure is needed in the following definition.

Definition 8.3.10 (Input prefix)

Let $a \in \mathit{inchan}(Q(m))$ for all m. The process $a?x \leadsto Q(x)$ will wait until its environment sends a message (say m) on channel a, and then it will behave like the process $Q(m)$

$$a?x \leadsto Q(x) \quad =_{df} \quad IN \gg (a?x \to Q(x)) \qquad \square$$

If a process must wait for input on two different channels before it can continue, it does not matter which of the two channels it waits on first. This means that internal choice is not so often explicitly needed in data flow.

L11 $a?x \leadsto (b?y \leadsto P(x, y)) \;=\; b?y \leadsto (a?x \leadsto P(x, y)), \qquad \text{if } a \neq b$

The following distributive law suggests that a non-deterministic choice can be made before or after receipt of an input.

L12 $(a?x \leadsto P(x)) \sqcap (a?x \leadsto Q(x)) \;=\; a?x \leadsto (P(x) \sqcap Q(x))$

A process that waits for input on channel a and then deadlocks cannot be distinguished from the process $SINK$, since the latter is always waiting for input on any channel. Thus input prefix has $SINK$ as its zero

L13 $(a?x \leadsto SINK) \;=\; SINK$

Examples 8.3.11

(1) The process *Copy* copies data from input channel c to output channel d

$$Copy \;=_{df}\; \mu X \bullet (c?x \leadsto (d!x \leadsto X))$$

As a data flow process, it behaves the same as $BUF_{\{c:d\}}$.

(2) A *Delay* element in hardware is one that outputs on each step the same message on channel d that it has input on the previous step from channel c

$$\begin{aligned} Delay \;&=_{df}\; c?x \leadsto Delay(x) \\ Delay(x) \;&=_{df}\; c?y \leadsto (d!x \leadsto Delay(y)) \end{aligned}$$

(3) The *Fork* process receives each input item on channel a and outputs it to both channels c and d

$$Fork_{\{a:c,d\}} \;=_{df}\; \mu X \bullet (a?x \leadsto (c!x \leadsto d!x \leadsto X)) \qquad \square$$

Examples 8.3.12

(1) Processes $Copy_1$ and $Copy_2$ have input channel c and output channel d. Both processes produce exactly two items x and y as output along c provided they have received x and y as inputs. But $Copy_1$ will produce its first output as soon as it receives its first input, while $Copy_2$ will not produce any output until it has received two input values.

$$Copy_1 \;=_{df}\; c?x \rightsquigarrow d!x \rightsquigarrow c?y \rightsquigarrow d!y \rightsquigarrow SINK$$

$$Copy_2 \;=_{df}\; c?x \rightsquigarrow c?y \rightsquigarrow d!x \rightsquigarrow d!y \rightsquigarrow SINK$$

$Copy_2$ (but not $Copy_1$) has a quiescent state after $< c.x >$, whereas $Copy_1$ is quiescent after $< c.x, d.x >$, which is not even a trace of $Copy_2$. The distinction between $Copy_1$ and $Copy_2$ cannot be made except by observing the relative order of communications on channel c and channel d.

(2) The process $Merge$ combines sequences of items received on two input channels a and b into a single output sequence on channel c. The output sequence produced in any quiescent state is always an interleaving of the two sequences that have been input so far. Note that this merge is fair, in the sense that it offers an external choice of which channel supplies its input. Furthermore, it never reaches a quiescent state until it has fully output all the messages from both its input channels. For most practical purposes, this is an adequate definition of fairness, with the possible advantage that it does not rely on reasoning about infinite behaviours.

$$Merge \;=_{df}\; [a?x \rightsquigarrow (c!x \rightsquigarrow Merge) \;\|\; b?y \rightsquigarrow (c!y \rightsquigarrow Merge)] \qquad \square$$

Let P and Q be CSP processes. For notational simplicity in later discussions we define their composite choice

$$P \oslash Q$$

as a mixture of external choice and internal choice. It describes a process that may either engage in an event acceptable to P and then behave like P, or make internal progress and then behave like Q. As shown in 8.2**L5** and 8.2**L12**, such a behaviour can result from concealment of some but not all of the actions in which a process is ready to engage.

Definition 8.3.13 (Composite choice)

$$P \oslash Q \;=_{df}\; (P \,\|\, Q) \sqcap Q \qquad \square$$

The following theorem states that the function [ ] distributes through prefixing and composite choice.

Theorem 8.3.14 (Distributivity of [ ])

(1) $[c!m \rightarrow Q] = [c!m \rightarrow [Q]] \;=\; c!m \leadsto [Q]$

(2) $[a?x \rightarrow P(x)] = [a?x \rightarrow [P(x)]] \;=\; a?x \leadsto [P(x)]$

(3) $[(\| a_i?x \rightarrow P(x)) \oslash Q] \;=\; [(\| a_i?x \rightarrow [P(x)]) \oslash [Q]]$

Proof of (1) Define

$$OUT(< c.m >) \;=_{df}\; BUF_{\{c':c\}}(< m >) \| (\|_{d \in outchan - \{c\}} BUF_{\{d':d\}})$$

$$\begin{array}{rlr}
 & [c!m \rightarrow Q] & \{\text{Def. } 8.3.5\} \\
= & IN \gg (c!m \rightarrow Q) \gg OUT & \{\mathbf{L2}\} \\
= & IN \gg Q \gg OUT(< c.m >) & \{\mathbf{L5} \text{ and } \mathbf{L7}\} \\
= & IN \gg [Q] \gg OUT(< c.m >) & \{\mathbf{L2}\} \\
= & IN \gg (c!m \rightarrow [Q]) \gg OUT & \{\text{Def. } 8.3.5\} \\
= & [c!m \rightarrow [Q]] & \{\text{Def. } 8.3.8 \text{ and } \mathbf{L5}\} \\
= & c!m \leadsto [Q] & \square
\end{array}$$

It is often convenient to define the cyclic behaviour of a process before deciding at what point in its cycle it should start. The following definition gives a means of getting a process into its desired initial state: it provides an operator which is the inverse of input prefixing; it thereby also assists in reasoning about processes.

Definition 8.3.15 (After)

The process $P/a.m$ behaves like P behaves after it has accepted m on input channel a. The value m remains buffered by the channel until P is ready to use it.

$$P/a.m \;=_{df}\; BUF_{a,\hat{a}}(< m >) \gg P[\hat{a}/a]$$

where $P[\hat{a}/a]$ behaves the same as P except that channel a is renamed as $\hat{a}$. For notational simplicity we adopt the convention that

$$P/a.m \;=\; P$$

whenever $a \notin chan(P)$. □

Theorem 8.3.16 (Well-definedness of the after operator)

$P/a.m$ is a data flow process.

Proof From Theorem 8.3.7 and the fact that $BUF_{\{a:\hat{a}\}}(< m >)$ is a data flow process. □

The after operator is distributive and has both $CHAOS$ and $SINK$ as zeros.

L14 $CHAOS/a.m \;=\; CHAOS$

L15 $SINK/a.m \;=\; SINK$

Because distinct channels buffer their data independently of each other

L16 $(P/a.m)/b.n \;=\; (P/b.n)/a.m, \qquad \text{if } a \neq b$

The value made available to a process which starts with input on channel a is just the one buffered by the after operation

L17 $(a?x \leadsto P(x))/a.m \;=\; P(m)$

The after operator on a distributes through prefixing on channels other than a

L18 $(b?x \leadsto P(x))/a.m \;=\; b?x \leadsto (P(x)/a.m), \qquad \text{if } a \neq b$

L19 $(c!n \leadsto Q)/a.m \;=\; c!n \leadsto (Q/a.m), \qquad \text{if } a \neq c$

After also distributes over $\oslash$.

L20 $P/b.m \;=\; [(\|_{a \in A \setminus \{b\}}\, a?x \to (P_a(x)/b.m)) \;\oslash\; S]$

$$\begin{array}{lll} \text{where} \quad P & =_{df} & [(\|_{a\in A}\, a?x \to P_a(x)) \oslash Q] \\ S & =_{df} & (Q/b.m) \sqcap P_b(m), \qquad \text{if } b \in A \\ & =_{df} & Q/b.m, \qquad \text{if } b \notin A \end{array}$$

The chaining operator acts for data flow in the same way that sequential composition acts for control flow. They are both restricted to flow in a single direction. But more interesting programs can be written with the aid of a facility for constructing loops. In the case of data flow, this is done by a parallel combinator similar to that for CSP. The main difference is that multiple inputs of the same output require an explicit forking for desynchronisation.

Let P and Q be processes with disjoint output channels. The notation

$$P \|_{DF} Q$$

represents a network with nodes P and Q, where the like-named channels of P and Q are connected. If P and Q share an input channel a, then both P and Q will input all messages sent to the network on the channel a. Input by P and Q on a does not have to be synchronised; an implementation should copy all messages sent on a into two buffers, so that they can be consumed at different rates by P and Q (or it can achieve the same effect by two pointers, one for each inputting process). Furthermore, the components P and Q may communicate with each other along any channel c which is an output channel of one and an input channel of the other. In $P\|_{DF}Q$, the output on c is retained so that c may be connected to yet further input channels. Again, extra buffers are needed for desynchronisation. Thus the output alphabet of the network $P\|_{DF}Q$ is the union of its components' output channels

$$outchan(P\|_{DF}Q) \;=_{df}\; outchan(P) \cup outchan(Q)$$

The input alphabet of the network is

$$inchan(P\|_{DF}Q) \;=_{df}\; (inchan(P) \cup inchan(Q)) \setminus outchan(P\|_{DF}Q)$$

Clearly the network $P\|_{DF}Q$ may refuse to output whenever both P and Q so refuse. As a result, $P\|_{DF}Q$ reaches a quiescent state only when both P and Q become inactive and do not produce any more output.

Definition 8.3.17 (Parallel composition)

Let $A \;=\; inchan(P) \cap inchan(Q)$ be the set of shared input channels.

Let $outchan(P) \cap outchan(Q) \;=\; \{\}$.

$$P\|_{DF}Q \;=_{df}\; (\|_{CSP}\{Fork_{\{a:0.a,\,1.a\}} \mid a \in A\}) \gg (0.P\|_{CSP}1.Q) \gg OUT$$

where $0.P$ behaves like P except the input channel names of A are decorated with label 0, and $1.Q$ is defined in a similar way. □

Theorem 8.3.18 (Well-definedness of the parallel operator)

$P\|_{DF}Q$ is a data flow process. □

Parallel composition is commutative, associative and distributive, and has $CHAOS$ as its zero. The expansion laws for data flow parallelism are given in **L21** to **L26**. The first law states that it is immaterial whether two processes in parallel wait independently or together for input to become available on a shared channel

L21 $(a?x \leadsto P(x))\,\|_{DF}\,(a?y \leadsto Q(y)) \;=\; a?z \leadsto (P(z)\,\|_{DF}\,Q(z))$

Let $a \neq b$, and $P \;=\; (a?x \leadsto P(x))$ and $Q \;=\; (b?y \leadsto Q(y))$. If both processes are waiting for input from the external world on different channels a and b, either input may take place. Subsequently, the corresponding process uses the input value, while the other one saves it up for later use.

L22 If $a \notin outchan(Q)$ and $b \notin outchan(P)$

then $P\,\|_{DF}\,Q \;=\; [(a?x \leadsto (P(x)\,\|_{DF}\,(Q/a.x))) \,[\!]\, (b?y \leadsto (P/b.y\,\|_{DF}\,Q(y)))]$

If one process is waiting for input from the environment on channel a and the other one is waiting for input from its partner on channel c, then only input on a may take place.

L23 If $a \notin outchan(Q)$ and $b \in outchan(P)$
then $P \,\|_{DF}\, Q \;=\; a?x \rightsquigarrow (P(x) \,\|_{DF}\, (Q/a.x))$

If each process is waiting for input from the other, we have deadlock.

L24 If $a \in outchan(Q)$ and $b \in outchan(P)$
then $P \,\|_{DF}\, Q \;=\; SINK$

If one of the processes is prepared to output, this may happen straight away. The message is buffered by the other process for future consumption if relevant, and is ignored otherwise.

L25 $(c!m \rightsquigarrow P) \,\|_{DF}\, Q \;=\; c!m \rightsquigarrow (P \|_{DF} (Q/c.m))$

The final complicated expansion law shows how to convert a network of alternative compositions into a single composition.

L26 $P \,\|_{DF}\, Q \;=\; [(S \oslash T)]$

where
$$\begin{aligned} P &=_{df} [(\|_{a \in A} a?x \to P_a(x)) \;\oslash\; U] \\ Q &=_{df} [(\|_{b \in B} b?y \to Q_b(y)) \;\oslash\; V] \\ S &=_{df} (\|_{a \in A \setminus outchan(Q)}\, a?x \to (P_a(x) \|_{DF} (Q/a.x))) \;\| \\ &\phantom{=_{df}} (\|_{b \in B \setminus outchan(P)}\, b?y \to ((P/b.y) \|_{DF} Q_b(y))) \\ T &=_{df} (U \,\|_{DF}\, Q) \sqcap (P \,\|_{DF}\, V) \end{aligned}$$

Concealment of an internal channel of an assembly requires systematic concealment of all possible values passing along the channel. It is convenient just to use the channel name for this, so we introduce the convention that

$$P \backslash \{c\} \quad \text{means} \quad P \backslash \{c.m \mid m \in \mathcal{A}_c(P)\}$$

and similarly for hiding a set C of channels. A hidden channel is used for output by (exactly) one of the component processes, and by definition of $\|_{DF}$ it is an output channel of the whole assembly. In fact we insist that only output channels are hidden, thus avoiding problems of massive non-determinism. Hiding as defined in CSP fortunately preserves the data flow process property. The algebraic properties of hiding in data flow are the ones already familiar from CSP.

L27 $(c!m \rightsquigarrow P) \backslash C \;=\; P \backslash C, \qquad$ if $c \in C$

L28 $(d!m \rightsquigarrow P) \backslash C \;=\; d!m \rightsquigarrow (P \backslash C), \qquad$ if $d \,\tilde{\in}\, C$

L29 $[(\|_{a\in A} a?x \rightarrow P_a(x)) \oslash Q]\backslash C \ = \ [(\|_{a\in A} a?x \rightarrow P_a(x)\backslash C) \oslash (Q\backslash C)]$

The hiding operator in data flow would have a much simpler definition if the single observable trace of our process model were split into a collection of traces, one for each channel of the alphabet. It is attractive to use the channel name itself to denote the sequence of values that have passed along it up to the time of the observation. Such a channel could then be hidden by simple existential quantification

$$P\backslash c \ = \ (\exists c, c' \bullet c = \langle\rangle \ \wedge \ P)$$

Specification could also be simpler: the copying process (Example 8.3.11(1)) is described just by the predicate, stating that the output is a prefix of the input

$$d \leq c$$

Of course, it becomes impossible to describe the difference between $Copy_1$ and $Copy_2$ (Example 8.3.12(1)). Nevertheless, the simpler model works well for deterministic data flow; it fails only on introduction of external choice, as shown by the following example [31].

Example 8.3.19 (Brock–Ackerman anomaly)

Let $P_i = (Merge/b.5/b.5\|_{DF}Copy_i)\backslash\{c\}$, for $i = 1, 2$.

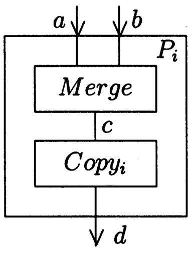

Figure 8.3.20 Data flow processes P_i

Using the algebraic laws we can show

$$\begin{aligned} P_1 \ &= \ [a?x \rightarrow d!x \rightsquigarrow [a?y \rightarrow (d!y \rightsquigarrow SINK) \oslash (d!5 \rightsquigarrow SINK)] \\ &\quad \oslash d!5 \rightsquigarrow [a?y \rightarrow (d!y \rightsquigarrow SINK) \oslash (d!5 \rightsquigarrow SINK)]] \\ P_2 \ &= \ [a?x \rightarrow [(a?y \rightarrow d!x \rightsquigarrow d!y \rightsquigarrow SINK) \\ &\quad \oslash (d!x \rightsquigarrow d!5 \rightsquigarrow SINK \sqcap d!5 \rightsquigarrow d!x \rightsquigarrow SINK)] \\ &\quad \oslash d!5 \rightsquigarrow d!5 \rightsquigarrow SINK] \end{aligned}$$

It is evident that P_2 is strictly more deterministic than P_1, even though the processes have identical input–output histories when recorded in separate variables for each channel a and d. But the two processes *can* be distinguished in the environment which incorporates feedback on channel a as shown below

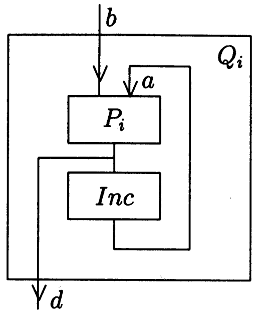

Figure 8.3.21 Data flow processes Q_i

where *Inc* was defined in Example 8.3.1(3). After some calculation we have

$$\begin{aligned} Q_1 &= (d!5 \leadsto d!6 \leadsto SINK) \sqcap (d!5 \leadsto d!5 \leadsto SINK) \\ Q_2 &= d!5 \leadsto d!5 \leadsto SINK \end{aligned}$$

In a practical implementation, the extra non-determinism of Q_1 can hardly be avoided. That is why we need a more complex theory for non-deterministic data flow, one which describes relative ordering of input on different channels. □

In this section, the set of data flow processes has been shown to be a subset of CSP processes. But this embedding is of significant complexity: it is no more and no less than an implementation of data flow within CSP. Unfortunately, the complexity conceals much of the extra value that the theory of data flow offers to the system designer. The value consists in the greater abstraction that the data flow paradigm offers in the specification: only the quiescent states need to be described. Furthermore, no mention need be made of refusals, because no process can ever refuse to input; nor can it refuse to output except in a quiescent state, and then it refuses *all* outputs. The transition between two levels of abstraction is a major step, similar to the step made several times in hardware design, for example between logic design and switching circuit design.