Chapter 1

Communicating Sequential Processes and Deadlock

Introduction

This chapter is concerned with laying the mathematical foundations for the thesis. In order to construct rigorous design rules for program design, we must first define a programming environment. This chapter introduces the CSP language of C. A. R. Hoare, which stands for *Communicating Sequential Processes* [Hoare 1985]. It is a notation for describing patterns of communication by algebraic expressions. These may be manipulated and transformed according to various laws in order to establish important properties of the system being described.

Behind CSP lies a mathematical theory of *failures* and *divergences*. Here a process is defined in terms of abstract sets representing circumstances under which it might be observed to go wrong. The model supplies a precise mathematical meaning to CSP processes, and is consistent with the algebraic laws which govern them.

The standard operational model of CSP is also described. Here processes are represented by transition systems which illustrate their inner machinery. There is a close relationship between the operational model of CSP and the Failures-Divergences model which means that the former may be used to prove properties of a system phrased in terms of the latter.

Following this, the concept of deadlock is formalised and we introduce techniques for deadlock analysis, developed by S.D.Brookes, A.W.Roscoe and N.Dathi. The problem of livelock is also considered.

CSP is not a programming language strictly speaking; it is a mathematical notation. However there are a number of concurrent programming languages based on CSP, such as occam and Ada, so theoretical results derived using this model are applicable to real programming.

1.1 The CSP Language

The basic syntax of CSP is described by the following grammar

$$
\begin{array}{rcl}
\mathit{Process} & ::== & \mathit{STOP} \quad | \\
 & & \mathit{SKIP} \quad | \\
 & & \mathit{event} \rightarrow \mathit{Process} \quad | \\
 & & \mathit{Process} \,;\, \mathit{Process} \quad | \\
 & & \mathit{Process}\ [\![\, \mathit{alph} \mid \mathit{alph} \,]\!]\ \mathit{Process} \quad | \\
 & & \mathit{Process} \mid\mid\mid \mathit{Process} \quad | \\
 & & \mathit{Process} \sqcap \mathit{Process} \quad | \\
 & & \mathit{Process} \,\Box\, \mathit{Process} \quad | \\
 & & \mathit{Process} \setminus \mathit{event} \quad | \\
 & & f(\mathit{Process}) \quad | \\
 & & \mathit{name} \quad | \\
 & & \mu\, \mathit{name} \bullet \mathit{Process}
\end{array}
$$

Here *event* ranges over a universal set of events, Σ, *alph* ranges over subsets of Σ, f ranges over a set of function names, and *name* ranges over a set of process names.

A process describes the behaviour of an object in terms of the events in which it may engage. The simplest process of all is *STOP*. This is the process which represents a deadlocked object. It never engages in any event. Another primitive process is *SKIP* which does nothing but terminate successfully; it only performs the special event $\surd$, which represents successful termination.

An event may be combined with a process using the prefix operator, written $\rightarrow$. The process $\mathit{bang} \rightarrow \mathit{UNIVERSE}$ describes an object which first engages in event *bang* then behaves according to process *UNIVERSE*. If we want to give this new process the name *CREATION* we write this as an equation

$$\mathit{CREATION} = \mathit{bang} \rightarrow \mathit{UNIVERSE}$$

Processes may be defined in terms of themselves using the principle of recursion. Consider a process to describe the ticking of an everlasting clock.

$$\mathit{CLOCK} = \mathit{tick} \rightarrow \mathit{CLOCK}$$

CLOCK is a process which performs event *tick* and then starts again. (This is a somewhat abstract definition. No information is given as to the duration or frequency of ticks. We are simply told that the clock will keep on ticking.)

In an algebraic sense *CLOCK* has been defined as the solution to an equation of the form

$$X = F(X)$$

It is not always the case in mathematics that such equations have solutions (*e.g.* there is no real solution to $x = x^2 + x + 1$). Fortunately the underlying mathematical theory of CSP guarantees that solutions exist to all such equations. The reason for this will be explained later. The solution to $X = F(X)$ is written

$$\mu\, Y \bullet F(Y)$$

where Y is a dummy process variable. Using this notation we could write *CLOCK* as

$$\mu\, Y \bullet \mathit{tick} \rightarrow Y$$

The recursive notation is commonly extended to a set of simultaneous equations where a number of processes are defined in terms of each other. This is known as mutual recursion, several examples of which will be found in later chapters.

There are a number of CSP operations which combine two processes to produce a new one. The first of these that we shall consider is sequential composition.

$$\mathit{UNIVERSE} = \mathit{EXPAND} \, ; \mathit{CONTRACT}$$

is the process which first behaves like *EXPAND*, but when *EXPAND* is ready to terminate it continues by behaving like *CONTRACT*. However it may also be possible that *EXPAND* will never terminate.

It is rather more complicated to compose two processes in parallel than in sequence. It is necessary to specify a set of events for each process, known as its *alphabet*. The process denoted

$$\mathit{PANTOHORSE} = \quad \begin{array}{c} \mathit{FRONT} \\ |[\, \{\mathit{forward,backward,nod}\} \mid \{\mathit{forward,backward,wag}\} \,]| \\ \mathit{BACK} \end{array}$$

represents the parallel composition of two processes: *FRONT* with alphabet $\{$*forward, backward,nod*$\}$ and *BACK* with alphabet $\{$*forward,backward,wag*$\}$. Here each process behaves according to its own definition, but with the constraint that events which are in the alphabet of both *FRONT* and *BACK*, *i.e. forward* and *backward*, require their simultaneous participation. However they may progress independently on those events belonging solely to their own alphabet. If a situation were to arise where *FRONT* could only perform event *forward* and *BACK* could only perform event *backward* then deadlock would have occurred.

Parallel composition may be extended to three or more processes; given a sequence of processes $V = \langle P_1, .., P_N \rangle$ with corresponding alphabets $\langle A_1, ..A_N \rangle$ we write their parallel composition as

$$\mathit{PAR}(V) \quad = \quad \|_{i=1}^{n} \; (P_i, A_i)$$

Note that it is implicitly assumed that the termination event $\surd$ requires the joint participation of each process P_i, whether or not it is included in their process alphabets.

An alternative form of parallel composition is *interleaving*, where there is no communication between the component processes. In the parallel combination

$$\mathit{BRAIN} \mid\mid\mid \mathit{MOUTH}$$

the two processes, *BRAIN* and *MOUTH*, progress independently of each other and no cooperation is required on any event, except for $\surd$, the termination event. Any other actions which are possible for both processes will only be performed by one process at a time. Interleaving is a commutative and associative operation and so we may extend the notation to various indexed forms, such as

$$\mid\mid\mid_{i=1}^{n} P_n, \quad \mid\mid\mid_{x:X} P_x$$

A useful feature of CSP is the ability to describe *nondeterministic* behaviour, which is where a process may operate in an unpredictable manner. The process

$$\mathit{BUFFER} = \mathit{TWOPLACE} \sqcap \mathit{THREEPLACE}$$

may behave either like process *TWOPLACE* or like process *THREEPLACE*, but there is no way of telling which in advance. The purpose of the $\sqcap$ operator is to specify concurrent systems in an abstract manner. At the design stage, there is no reason to provide any more detail than is necessary and, where possible, implementation decisions should be deferred until later.

This operation is known as *internal choice*. CSP also contains an *external choice* operator $\Box$ which enables the future behaviour of a process to be controlled by other processes running along side it in parallel, which, collectively, we call its *environment*.

The process

$$\mathit{MW} = \mathit{DEFROST} \Box \mathit{COOK}$$

may behave like *DEFROST* or like *COOK*. Its behaviour may be controlled by its environment provided that this control is exercised on the very first event. If an initial event *button1* is offered by *DEFROST* that is not an initial event of *COOK*, then the environment may coerce *MW* into behaving like *DEFROST*, by performing *button1* as its initial event. If, however, the environment were to offer an initial event that is allowed by both *DEFROST* and *COOK* then the choice between them would be nondeterministic.

Both the choice operators may be extended to indexed forms. We write

$$\Box_{x:A}\ x \rightarrow P_x$$

to represent the behaviour of an object which offers any event of a set A to its environment. Once some initial event x has been performed the future behaviour of the object is described by the process P_x. However, the process

$$\sqcap_{x:A}\ x \rightarrow P_x$$

(where, for technical reasons, A must be finite) offers exactly one event x from A to its environment, the choice being non-deterministic.

Sometimes it is useful to be able to restrict the definition of a process to a subset of relevant events that it performs. This is done using the hiding operator ($\backslash$). The process

$$\mathit{CREATION} \setminus \mathit{bang}$$

behaves like *CREATION*, except that each occurrence of event *bang* is concealed. Note that it is not permitted to hide event $\surd$.

Concealment may introduce nondeterminism into deterministic processes. It may also introduce the phenomenon of *divergence*. This is a drastic situation where a process performs an endless series of hidden actions. Consider, for instance, the process

$$\mathit{CLOCK} \setminus \mathit{tick}$$

which is clearly a divergent process.

It is conventional to extend the notation to $P \setminus A$, where A is a finite set of events.

Finally let us briefly consider process relabelling. Let f be an *alphabet transformation function* $f : \Sigma \to \Sigma$, which satisfies the property that only finitely many events may be mapped onto a single event. Then the process $f(P)$ can perform the event $f(e)$ whenever P can perform event e. As an example consider a function *new* which maps *tick* to *tock*. Then we have

$$\mathit{new}(\mathit{CLOCK}) = \mathit{tock} \to \mathit{new}(\mathit{CLOCK})$$

Some important algebraic laws which govern CSP processes are given in figures 1.1 and 1.2, which vary in complexity. They are taken from [Hoare 1985], [Brookes 1983], and [Brookes and Roscoe 1985a]. (In some cases the syntax has been modified to conform to the version of CSP described above.) Note that this is not a complete list. The following example illustrates the use of these laws.

Consider a process to describe a vending machine which sells tea for a price of one coin and coffee for two coins.

$$\mathit{VM} = \mathit{coin} \to ((\mathit{tea} \to \mathit{VM}) \,\Box\, (\mathit{coin} \to \mathit{coffee} \to \mathit{VM}))$$

After inserting a coin, a customer can control the future behaviour of the machine by either inserting another coin, or taking a cup of tea.

We now define a process which describes a particular customer who loves tea and is prepared to pay for it. Coffee he will tolerate, but only if it is provided free of charge.

$$\mathit{TD} = (\mathit{coin} \to \mathit{tea} \to \mathit{TD}) \,\Box\, (\mathit{coffee} \to \mathit{TD})$$

To illustrate the use of algebraic laws to simplify CSP process definitions, consider what happens when the tea drinker tries to use the vending machine. Both processes have alphabet $\{coin, coffee, tea\}$.

$$\begin{array}{rcl}
SYSTEM & = & VM \; [| \{coin,coffee,tea\} \mid \{coin,coffee,tea\} |] \; TD \\
& = & \left(\begin{array}{c} (coin \rightarrow ((tea \rightarrow VM) \Box (coin \rightarrow coffee \rightarrow VM))) \\ [| \{coin,coffee,tea\} \mid \{coin,coffee,tea\} |] \\ ((coin \rightarrow tea \rightarrow TD) \Box (coffee \rightarrow TD)) \end{array} \right) \\
& = & coin \rightarrow \left(\begin{array}{c} ((tea \rightarrow VM) \Box (coin \rightarrow coffee \rightarrow VM)) \\ [| \{coin,coffee,tea\} \mid \{coin,coffee,tea\} |] \\ tea \rightarrow TD \end{array} \right) \\
& & \text{using law 1.22 with } X = \{coin\},\ Y = \{coin,coffee\},\ Z = \{coin\} \\
& = & coin \rightarrow tea \rightarrow (VM \; [| \{coin,coffee,tea\} \mid \{coin,coffee,tea\} |] \; TD) \\
& & \text{using law 1.22 with } X = \{tea,coin\},\ Y = \{tea\},\ Z = \{tea\} \\
& = & coin \rightarrow tea \rightarrow SYSTEM
\end{array}$$

The system has been reduced to a very simple sequential definition. We see that although no coffee will be consumed in this situation, the system will never deadlock.

The account of the CSP language given here is incomplete. Only the core language has been considered with certain ‘advanced’ operators omitted. The language described corresponds to the modern version of CSP, as given in [Formal Systems 1993], which differs slightly from the language presented in Hoare’s book [Hoare 1985].

Figure 1.1: Laws of CSP I

$$
\begin{aligned}
\mathit{SKIP} \mathbin{;} P &= P \mathbin{;} \mathit{SKIP} \quad = \quad P && (1.1)\\
\mathit{STOP} \mathbin{;} P &= \mathit{STOP} && (1.2)\\
(P \mathbin{;} Q) \mathbin{;} R &= P \mathbin{;} (Q \mathbin{;} R) && (1.3)\\
(a \to P) \mathbin{;} Q &= a \to (P \mathbin{;} Q) && (1.4)\\
P \mathbin{|[\, A \mid B \,]|} Q &= Q \mathbin{|[\, B \mid A \,]|} P && (1.5)\\
P \mathbin{|[\, A \mid B \cup C \,]|} (Q \mathbin{|[\, B \mid C \,]|} R) &= (P \mathbin{|[\, A \mid B \,]|} Q) \mathbin{|[\, A \cup B \mid C \,]|} R && (1.6)\\
P \mathbin{|||} Q &= Q \mathbin{|||} P && (1.7)\\
P \mathbin{|||} \mathit{SKIP} &= P && (1.8)\\
P \mathbin{|||} (Q \mathbin{|||} R) &= (P \mathbin{|||} Q) \mathbin{|||} R && (1.9)\\
P \sqcap P &= P && (1.10)\\
P \sqcap Q &= Q \sqcap P && (1.11)\\
P \sqcap (Q \sqcap R) &= (P \sqcap Q) \sqcap R && (1.12)\\
P \mathbin{\Box} P &= P && (1.13)\\
P \mathbin{\Box} Q &= Q \mathbin{\Box} P && (1.14)\\
P \mathbin{\Box} (Q \mathbin{\Box} R) &= (P \mathbin{\Box} Q) \mathbin{\Box} R && (1.15)\\
P \mathbin{|[\, A \mid B \,]|} (Q \sqcap R) &= (P \mathbin{|[\, A \mid B \,]|} Q) \sqcap (P \mathbin{|[\, A \mid B \,]|} R) \\
& && (1.16)\\
P \mathbin{\Box} (Q \sqcap R) &= (P \mathbin{\Box} Q) \sqcap (P \mathbin{\Box} R) && (1.17)\\
P \sqcap (Q \mathbin{\Box} R) &= (P \sqcap Q) \mathbin{\Box} (P \sqcap R) && (1.18)\\
(x \to P) \mathbin{\Box} (x \to Q) &= (x \to P) \sqcap (x \to Q) \\
&= x \to (P \sqcap Q) && (1.19)\\
P \mathbin{\Box} \mathit{STOP} &= P && (1.20)\\
\Box_{x:\{\}}\, x \to P_x &= \mathit{STOP} && (1.21)
\end{aligned}
$$

Figure 1.2: Laws of CSP II

$$\begin{array}{rcll}
\text{Let} \quad P & = & \Box_{x:X}\ x \to P_x & \\
Q & = & \Box_{y:Y}\ y \to Q_y & \\
\text{Then} \quad P\ [\![A \mid B]\!]\ Q & = & \Box_{z:Z}\ z \to (P_z{}'\ [\![A \mid B]\!]\ Q_z{}') & \\
\text{where} \quad P_z{}' & = & \left\{ \begin{array}{ll} P_z & \textit{if} \quad z \in X \\ P & \text{otherwise} \end{array} \right. & \\
\text{and} \quad Q_z{}' & = & \left\{ \begin{array}{ll} Q_z & \textit{if} \quad z \in Y \\ Q & \text{otherwise} \end{array} \right. & \\
\text{and} \quad Z & = & (X \cap Y) \cup (X - B) \cup (Y - A) & \\
\text{assuming} \quad X \subseteq A & & \text{and} \quad Y \subseteq B & (1.22) \\
\Box_{b:B}\ (b \to P_b)\ |||\Box_{c:C}\ (c \to Q_c) & = & (\Box_{b:B}\ (b \to (P_b\ |||\Box_{c:C}\ (c \to Q_c))))\ \Box & \\
& & (\Box_{c:C}\ (c \to (Q_c\ |||\Box_{b:B}\ (b \to P_c)))) & (1.23) \\
\mathit{SKIP} \setminus x & = & \mathit{SKIP} & (1.24) \\
\mathit{STOP} \setminus x & = & \mathit{STOP} & (1.25) \\
(P \setminus x) \setminus y & = & (P \setminus y) \setminus x & (1.26) \\
(x \to P) \setminus x & = & P \setminus x & (1.27) \\
(x \to P) \setminus y & = & x \to (P \setminus y) \quad \text{if} \quad x \neq y & (1.28) \\
(P\ ;\ Q) \setminus x & = & (P \setminus x)\ ;\ (Q \setminus x) & (1.29) \\
(P\ [\![A \mid B]\!]\ Q) \setminus x & = & P\ [\![A \mid B - \{x\}]\!]\ (Q \setminus x) & \\
& & \text{if} \quad x \notin A & (1.30) \\
(P \sqcap Q) \setminus x & = & (P \setminus x) \sqcap (Q \setminus x) & (1.31) \\
((x \to P)\ \Box\ (y \to Q)) \setminus x & = & (P \setminus x) \sqcap ((P \setminus x)\ \Box\ (y \to (Q \setminus x))) & \\
& & \text{if} \quad x \neq y & (1.32) \\
f(\mathit{STOP}) & = & \mathit{STOP} & (1.33) \\
f(e \to P) & = & f(e) \to f(P) & (1.34) \\
f(P\ ;\ Q) & = & f(P)\ ;\ f(Q) \quad \text{if} \quad f^{-1}(\surd) = \{\surd\} & (1.35) \\
f(P\ |||\ Q) & = & f(P)\ |||\ f(Q) & (1.36) \\
f(P\ \Box\ Q) & = & f(P)\ \Box\ f(Q) & (1.37) \\
f(P \sqcap Q) & = & f(P) \sqcap f(Q) & (1.38) \\
f(P \setminus f^{-1}(x)) & = & f(P) \setminus x & (1.39)
\end{array}$$

1.2 The Failures-Divergences Model

In the preceding section the concept of communicating processes was introduced informally and the corresponding algebraic laws were stated without mathematical justification. In this section a precise semantic definition of CSP processes is given from which the laws can be deduced. This is known as the *Failures-Divergences* model. Here a process is defined in terms of important observable properties – *traces*, *failures* and *divergences*.

A *trace* of a process P is any finite sequence of events that it may initially perform. For instance

$$\{\langle \mathit{coffee}, \mathit{coffee}, \mathit{coffee}\rangle, \langle \mathit{coin}, \mathit{tea}\rangle\} \subset \mathit{traces}(TD)$$

The following useful operations are defined on traces

- Catenation: $s \frown t$

$$\langle s_1, s_2, .., s_m\rangle \frown \langle t_1, t_2, .., t_n\rangle = \langle s_1, .., s_m, t_1, .., t_n\rangle$$

- Restriction: $s \upharpoonright B$, trace s restricted to elements of set B

Example: $\langle a, b, c, d, b, d, a\rangle \upharpoonright \{a, b, c\} = \langle a, b, c, b, a\rangle$

- Replication: s^n trace s repeated n times.

Example: $\langle a, b\rangle^2 = \langle a, b, a, b\rangle$

- Count: $s \downarrow x$ number of occurrences of event x in trace s

Example: $\langle x, y, z, x, x\rangle \downarrow x = 3$

- Length: $|s|$ the length of trace s.

Example: $|\langle a, b, c\rangle| = 3$

- Merging: $\mathit{merge}(s, t)$ the set of all possible interleavings of trace s with trace t

Example: $\mathit{merge}(\langle a, b\rangle, \langle c\rangle) = \{\langle a, b, c\rangle, \langle a, c, b\rangle, \langle c, a, b\rangle\}$

A complication to trace interleaving is that the $\surd$ event requires the joint participation of both traces. This means that a trace which contains $\surd$ cannot be interleaved with one that does not.

Examples:
$$\begin{aligned} \mathit{merge}(\langle a, b, \surd\rangle, \langle c, \surd\rangle) &= \left\{ \begin{array}{l} \langle a, b, c, \surd\rangle, \\ \langle a, c, b, \surd\rangle, \\ \langle c, a, b, \surd\rangle \end{array} \right\} \\ \mathit{merge}(\langle a, b, \surd\rangle, \langle c\rangle) &= \{\} \end{aligned}$$

The *failures* of a process describe the circumstances under which it might deadlock. Each failure of a process P consists of a pair (s, X) where s is a trace of P and X is a set of events which if offered to P by its environment after it has performed trace s, might be completely refused. For instance

$$(\langle \mathit{coin,tea,coin,tea,coin,coin} \rangle, \{\mathit{tea,coin}\}) \in \mathit{failures}(\mathit{VM})$$

This describes a situation where the vending machine *VM* has dispensed two cups of tea and then accepted two coins. At this point the machine is willing only to dispense coffee. If a user arrives who wants tea, and is only prepared to take a cup of tea or to insert another coin then deadlock will ensue.

The concept of failures is commonly used to write specifications for the behaviour of CSP processes. Consider the following specification.

$$\forall (s, X) : \mathit{failures}(P). \quad s \downarrow \mathit{in} > s \downarrow \mathit{out} \Longrightarrow \mathit{out} \notin X$$

This states that whenever process P has performed the event *in* more often than the event *out* it must guarantee not to refuse event *out*. This might form part of the overall specification for a buffer.

The *divergences* of a process are a list of the traces after which it might diverge, *e.g.*

$$\langle\rangle \in \mathit{divergences}(\mathit{CLOCK} \setminus \mathit{tick})$$

There are several further aspects of notation that are needed in order to define the model which are as follows. The *Power-Set* of a set A, written $\mathbf{P}\, A$, consists of all subsets of A. The *Finite Power-Set* of A, written $p(A)$, consists of all finite subsets of A. The set of all finite sequences (including $\langle\rangle$) that may be formed from elements of A is written A^*.

The *Failures-Divergences* model is based on a universal set of events Σ. Each CSP process is uniquely defined by a pair of sets (F, D), corresponding to its *failures* and *divergences*, such that

$$\begin{aligned} F &\subseteq \Sigma^* \times \mathbf{P}\,\Sigma \\ D &\subseteq \Sigma^* \end{aligned}$$

There are seven axioms that such a pair of sets must satisfy in order to qualify as a process. (Note that there are several versions of these in existence in the literature. This version comes from [Brookes and Roscoe 1985a].)

(1) $(\langle\rangle, \{\}) \in F$

(2) $(s^\frown t, \{\}) \in F \Longrightarrow (s, \{\}) \in F$

(3) $(s, Y) \in F \wedge X \subseteq Y \Longrightarrow (s, X) \in F$

(4) $(s, X) \in F \wedge (\forall\, c \in Y.((s^\frown\langle c\rangle, \{\}) \notin F)) \Longrightarrow (s, X \cup Y) \in F$

(5) $(\forall\, Y \in p(X).(s, Y) \in F) \Longrightarrow (s, X) \in F$

(6) $s \in D \wedge t \in \Sigma^* \Longrightarrow s^\frown t \in D$

(7) $s \in D \wedge X \subseteq \Sigma \Longrightarrow (s, X) \in F$

Putting the first four axioms into words tells us that every process starts off with an empty trace (axiom 1). In order to perform trace s, it must be able to perform any prefix of s (axiom 2). A subset of a refusal set is also a refusal set (axiom 3). If the process can refuse the events in X, and cannot perform any of the events in Y as its next step, then it may also refuse $X \cup Y$ (axiom 4). These are all basic intuitive properties of processes.

Axiom 5 states that a set may be refused if all its finite subsets may be refused. This is to allow for the possibility of Σ being an infinite set without complicating the theory.

Axioms 6 and 7 state that once a process diverges it may subsequently perform any trace imaginable and will behave in a totally nondeterministic manner. This is a rather harsh treatment of the phenomenon of divergence. If we put our *CLOCK* in a vacuum to hide its ticking we would not expect such dramatic behaviour. It is, however, a convenient means to make the theory work better based on the assumption that the possibility of divergence is catastrophic (see [Roscoe 1994]).

There is a natural *partial order* (see appendix A) on the set of all processes given by

$$(F_1, D_1) \sqsubseteq (F_2, D_2) \iff F_1 \supseteq F_2 \wedge D_1 \supseteq D_2$$

The interpretation of this is that process P_1 is worse than P_2 if it can deadlock or diverge whenever P_2 can. This ordering is in fact a *complete partial order*. The bottom, or worst, element $\perp$ represents the process which always diverges, corresponding to the decision to treat this form of behaviour as the least desirable. It is a chaotic process which can do absolutely anything in a totally unpredictable manner. It is defined as follows.

$$\begin{aligned} \mathit{failures}(\perp) &= \Sigma^* \times \mathbf{P}\,\Sigma \\ \mathit{divergences}(\perp) &= \Sigma^* \end{aligned}$$

The *failures* and *divergences* of the fundamental CSP terms are defined in figures 1.3 and 1.4. (These are the same as in [Brookes and Roscoe 1985a], except that the definitions of parallel composition and interleaving are modified to reflect the fact that in the modern version of CSP these operators implicitly require the cooperation of both processes in performing the $\surd$ event.) This covers all closed, non-recursive CSP terms.

All of the CSP operators can be shown to be *well-defined*. In other words, if you apply any of them to existing CSP processes, the resulting object will itself be a process: its failures and divergences obeying the seven axioms of the model. They are also *continuous*, with respect to $\sqsubseteq$. This is important because it means that any recursive CSP equation of the form $X = F(X)$ has a solution, by Tarski's fixed point theorem (see appendix A). The *least* solution is given by

$$\mu X \bullet F(X) = \sqcup\{F^n(\perp) | n \in \mathrm{N}\}$$

This means that if you want to find the solution to $X = F(X)$ you start at the bottom $\perp$ and repeatedly apply the function F to it. For instance *CLOCK* is the limit of the series

$$\perp, \mathit{tick} \rightarrow \perp, \mathit{tick} \rightarrow \mathit{tick} \rightarrow \perp, \mathit{tick} \rightarrow \mathit{tick} \rightarrow \mathit{tick} \rightarrow \perp, ..$$

The failures and divergences of $\mu\, X \bullet F(X)$ may be calculated as follows

$$\begin{aligned} \mathit{divergences}(\mu\, X \bullet F(X)) &= \bigcap_{n \in \mathbb{N}} \mathit{divergences}(F^n(\perp)) \\ \mathit{failures}(\mu\, X \bullet F(X)) &= \bigcap_{n \in \mathbb{N}} \mathit{failures}(F^n(\perp)) \end{aligned}$$

This is how we define the meaning of recursion in CSP.

This approach may be extended to mutual recursion, where a number of processes are defined by a system of simultaneous equations. The trick here is to let X be a vector of processes, satisfying an equation of the form $X = F(X)$. The solution is then defined as the least fixed point of F in the same way as before.

Whilst the fact that recursion is well-defined in CSP is crucial to the theory, it is really only of technical interest to a designer of concurrent systems. Basically it allows him to specify processes recursively, assured in the knowledge that it is a sound practice.

The failures-divergences model of CSP is used for formal reasoning about the behaviour of concurrent systems defined by CSP equations. The partial ordering of nondeterminism is very important to the stepwise refinement of concurrent systems. Starting from an abstract non-deterministic definition, details of components may be independently fleshed out whilst preserving important properties of the overall system such as freedom from deadlock. This will be explained in more detail later.

Figure 1.3: Denotational Semantics for CSP I

$$
\begin{array}{rcl}
\mathit{divergences}(\mathit{STOP}) & = & \{\} \\
\mathit{failures}(\mathit{STOP}) & = & \{\langle\rangle\} \times \mathbf{P}\,\Sigma \\
\mathit{divergences}(\mathit{SKIP}) & = & \{\} \\
\mathit{failures}(\mathit{SKIP}) & = & (\{\langle\rangle\} \times \mathbf{P}(\Sigma - \surd)) \\
 & \cup & (\{\langle\surd\rangle\} \times \mathbf{P}\,\Sigma) \\
\mathit{divergences}(x \to P) & = & \{\langle x\rangle^\frown s | s \in \mathit{divergences}(P)\} \\
\mathit{failures}(x \to P) & = & \{(\langle\rangle, X) | X \subseteq \Sigma - \{x\}\} \\
 & \cup & \{(\langle x\rangle^\frown s, X) | (s, X) \in \mathit{failures}(P)\} \\
\mathit{divergences}(P\,;\,Q) & = & \mathit{divergences}(P) \\
 & \cup & \left\{ \begin{array}{c} s^\frown t | s^\frown\langle\surd\rangle \in \mathit{traces}(P) \wedge s\surd\text{-}\mathit{free} \\ \wedge t \in \mathit{divergences}(Q) \end{array} \right\} \\
\mathit{failures}(P\,;\,Q) & = & \{(s, X) | s\surd\text{-}\mathit{free} \wedge (s, X \cup \langle\surd\rangle) \in \mathit{failures}(P)\} \\
 & \cup & \left\{ \begin{array}{c} (s^\frown t, X) | s^\frown\langle\surd\rangle \in \mathit{traces}(P) \wedge s\surd\text{-}\mathit{free}\wedge \\ (t, X) \in \mathit{failures}(Q) \end{array} \right\} \\
 & \cup & \{(s, X) | s \in \mathit{divergences}(P\,;\,Q)\} \\
\begin{array}{c} \mathit{divergences} \\ (P \;|[\, A \,|\, B \,]|\; Q) \end{array} & = & \left\{ \begin{array}{l} s^\frown t | s \in (A \cup B \cup \{\surd\})^* \wedge \\ \left(\begin{array}{l} \left(\begin{array}{c} s \upharpoonright (A \cup \{\surd\}) \in \mathit{divergences}(P) \wedge \\ s \upharpoonright (B \cup \{\surd\}) \in \mathit{traces}(Q) \end{array} \right) \\ \vee \left(\begin{array}{c} s \upharpoonright (B \cup \{\surd\}) \in \mathit{divergences}(Q) \wedge \\ s \upharpoonright (A \cup \{\surd\}) \in \mathit{traces}(P) \end{array} \right) \end{array} \right) \end{array} \right\} \\
\mathit{failures}(P \;|[\, A \,|\, B \,]|\; Q) & = & \left\{ \begin{array}{c} (s, X \cup Y \cup Z) | s \in (A \cup B \cup \{\surd\})^* \\ \wedge X \subseteq (A \cup \{\surd\}) \wedge Y \subseteq (B \cup \{\surd\}) \wedge \\ Z \subseteq (\Sigma - (A \cup B \cup \{\surd\})) \\ \wedge (s \upharpoonright (A \cup \{\surd\}), X) \in \mathit{failures}(P) \\ \wedge (s \upharpoonright (B \cup \{\surd\}), Y) \in \mathit{failures}(Q) \end{array} \right\} \\
 & \cup & \{(s, X) | s \in \mathit{divergences}(P \;|[\, A \,|\, B \,]|\; Q)\}
\end{array}
$$

Figure 1.4: Denotational Semantics for CSP II

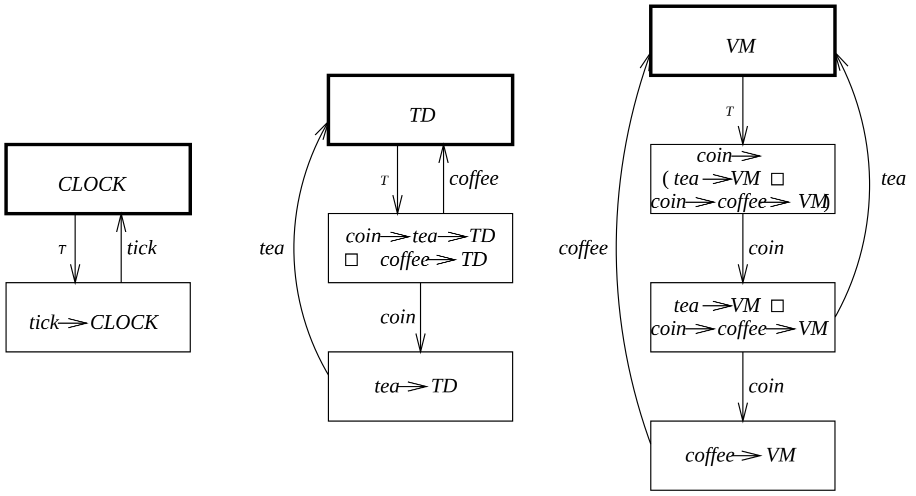
$$\begin{array}{rcl}
\mathit{divergences}(P \mid\mid\mid Q) & = & \left\{ \begin{array}{c} \exists\, s, t. \quad u \in \mathit{merge}(s,t) \wedge \\ \left(\begin{array}{c} (s \in \mathit{divergences}(P) \wedge t \in \mathit{traces}(Q)) \vee \\ (s \in \mathit{traces}(P) \wedge t \in \mathit{divergences}(Q)) \end{array} \right) \end{array} \right\} \\
\mathit{failures}(P \mid\mid\mid Q) & = & \left\{ \begin{array}{c} (u, X) \mid \exists\, s, t. \\ \left(\begin{array}{c} \left(\begin{array}{c} (s, X - \{\surd\}) \in \mathit{failures}(P) \wedge \\ (t, X) \in \mathit{failures}(Q) \end{array} \right) \vee \\ \left(\begin{array}{c} (s, X) \in \mathit{failures}(P) \wedge \\ (t, X - \{\surd\}) \in \mathit{failures}(Q) \end{array} \right) \end{array} \right) \wedge \\ u \in \mathit{merge}(s,t) \end{array} \right\} \\
& \cup & \{(s, X) \mid s \in \mathit{divergences}(P \mid\mid\mid Q)\} \\
\mathit{divergences}(P \sqcap Q) & = & \mathit{divergences}(P) \cup \mathit{divergences}(Q) \\
\mathit{failures}(P \sqcap Q) & = & \mathit{failures}(P) \cup \mathit{failures}(Q) \\
\mathit{divergences}(P \,\square\, Q) & = & \mathit{divergences}(P) \cup \mathit{divergences}(Q) \\
\mathit{failures}(P \,\square\, Q) & = & \left\{ \begin{array}{c} (s, X) \mid (s, X) \in \mathit{failures}(P) \cap \mathit{failures}(Q) \vee \\ \left(\begin{array}{c} s \neq \langle\rangle \wedge \\ (s, X) \in \mathit{failures}(P) \cup \mathit{failures}(Q) \end{array} \right) \end{array} \right\} \\
& \cup & \{(s, X) \mid s \in \mathit{divergences}(P \,\square\, Q)\} \\
\mathit{divergences}(P \setminus x) & = & \left\{ \begin{array}{c} (s \upharpoonright (\Sigma - \{x\}))^\frown t \mid \\ \left(\begin{array}{c} s \in \mathit{divergences}(P) \\ \vee (\forall\, n. s^\frown \langle x \rangle^n \in \mathit{traces}(P))) \end{array} \right) \end{array} \right\} \\
\mathit{failures}(P \setminus x) & = & \{(s \upharpoonright (\Sigma - \{x\}), X) \mid (s, X \cup \{x\}) \in \mathit{failures}(P)\} \\
& \cup & \{(s, X) \mid s \in \mathit{divergences}(P \setminus \{x\})\} \\
\mathit{divergences}(f(P)) & = & \{f(s)t \mid s \in \mathit{divergences}(P)\} \\
\mathit{failures}(f(P)) & = & \{(f(s), X) \mid (s, f^{-1}(X)) \in \mathit{failures}(P)\} \\
& \cup & \{(s, X) \mid s \in \mathit{divergences}(f(P))\}
\end{array}$$

1.3 Operational Semantics

So far we have encountered two ways of looking at communicating processes: firstly as algebraic expressions and secondly in terms of abstract mathematical sets based on their observable behaviour. There is no obvious way of seeing from either of these representations how our processes might be realised on a machine. We need a more concrete approach – an operational model. The operational semantics of CSP is a mapping from CSP expressions to *state transition systems*. A state transition system is a labelled digraph where each vertex represents a *state* in which the process may rest. The outgoing arcs from each vertex represent the events that the process is ready to perform when in the associated state. The destination vertex of each of these arcs represents the new state that the process attains by performing the associated event. There is one particular vertex that is marked as the initial state of the process. A special event τ is used to represent concealed events or internal decisions. States which have outgoing τ-labelled arcs are called *unstable*. Those which do not are called *stable*.

Transition systems for certain processes that we have previously encountered are shown in figure 1.5. Note that recursion is represented here by the presence of circuits in the digraphs.

Figure 1.5: State Transition Systems


CLOCK
τ
tick
tick → CLOCK
TD
τ
coffee
tea
coin → tea → TD □ coffee → TD
coin
tea → TD
VM
τ
coin → (tea → VM □ coin → coffee → VM)
coffee
tea
coin
tea → VM □ coin → coffee → VM
coin
coffee → VM

The operational semantics of CSP is defined by a set of inference rules which define a mapping from closed CSP terms to transition systems. Each clause consists of a (possibly empty) set of assertions $\{A_1, .., A_n\}$ and a conclusion C presented in the form

$$\frac{A_1, .., A_n}{C}$$

Consider, for example, the rules which define sequential composition.

$$\frac{P \xrightarrow{a} P'}{(P\ ;\ Q) \xrightarrow{a} (P'\ ;\ Q)}\, a \neq \surd$$

$$\frac{P \xrightarrow{\surd} P'}{(P\ ;\ Q) \xrightarrow{\tau} Q}$$

The first clause states that if a process P can perform a certain event a, where a can be any event except for $\surd$, and its subsequent behaviour is then described by the process P', then process $P\,;Q$ can also perform a and its subsequent behaviour is described by $P'\ ;\ Q$. The second clause tells us that if P can terminate straight away, by performing event $\surd$, then $P\ ;\ Q$ can perform an internal event τ and then behave like Q.

The full set of operational rules for the subset of the CSP language that we are using is given in figures 1.6 and 1.7. These clauses are taken from [Roscoe 1988a] and [Formal Systems 1993]. They may be used to systematically construct transition digraphs from systems of CSP equations, as is done by the refinement checking program FDR [Formal Systems 1993]. As an example, let us consider how the transition digraph for process *TD*, figure 1.5, is constructed. First of all the defining CSP equation is converted into a syntax tree as follows

$$\begin{tikzcd}
 & \boxed{\mathit{TD}} \arrow[d, phantom, "="] & \\
 & \boxed{\mathit{coin} \to \mathit{tea} \to \mathit{TD} \;\square\; \mathit{coffee} \to \mathit{TD}} \arrow[d, phantom, "\square"] & \\
\boxed{\mathit{coin} \to \mathit{tea} \to \mathit{TD}} \arrow[d, phantom, "\mathit{coin} \to"] & & \boxed{\mathit{coffee} \to \mathit{TD}} \arrow[d, phantom, "\mathit{coffee} \to"] \\
\boxed{\mathit{tea} \to \mathit{TD}} \arrow[d, phantom, "\mathit{tea} \to"] & & \boxed{\mathit{TD}} \\
\boxed{\mathit{TD}} & &
\end{tikzcd}$$

The syntax tree shows how the defining CSP term for *TD* is composed from operators acting on sub-processes. Each framed process term represents a potential state of *TD* or a state of one of its sub-processes. We can expand some of these straight away using the operational rule for event prefixing.

$$\boxed{\mathit{coin} \to \mathit{tea} \to \mathit{TD}} \xrightarrow{\mathit{coin}} \boxed{\mathit{tea} \to \mathit{TD}}$$

$$\boxed{\mathit{tea} \to \mathit{TD}} \xrightarrow{\mathit{tea}} \boxed{\mathit{TD}}$$

$$\boxed{\mathit{coffee} \to \mathit{TD}} \xrightarrow{\mathit{coffee}} \boxed{\mathit{TD}}$$

We are now ready to expand the external choice construct which gives us

$$\boxed{\mathit{coin} \to \mathit{tea} \to \mathit{TD} \;\square\; \mathit{coffee} \to \mathit{TD}} \xrightarrow{\mathit{coin}} \boxed{\mathit{tea} \to \mathit{TD}}$$

$$\boxed{coin \rightarrow tea \rightarrow TD \;\Box\; coffee \rightarrow TD} \overset{coffee}{\rightarrow} \boxed{TD}$$

The rule for recursion enables us to make the following connection.

$$\boxed{TD} \overset{\tau}{\rightarrow} \boxed{coin \rightarrow tea \rightarrow TD \;\Box\; coffee \rightarrow TD}$$

It may not be immediately obvious how this follows from the rule for recursion, which is phrased in terms of the μ operator. The reason that it does follow is that we are actually using *TD* as an abbreviation for

$$\mu X.coin \rightarrow tea \rightarrow X \;\Box\; coffee \rightarrow X$$

It is now the case that every state reachable from *TD* has been expanded, and together they constitute a state-transition system for *TD*, which is

$$\left\{ \begin{array}{c} \boxed{TD} \overset{\tau}{\rightarrow} \boxed{coin \rightarrow tea \rightarrow TD \;\Box\; coffee \rightarrow TD} \\ \boxed{coin \rightarrow tea \rightarrow TD \;\Box\; coffee \rightarrow TD} \overset{coin}{\rightarrow} \boxed{tea \rightarrow TD} \\ \boxed{coin \rightarrow tea \rightarrow TD \;\Box\; coffee \rightarrow TD} \overset{coffee}{\rightarrow} \boxed{TD} \\ \boxed{tea \rightarrow TD} \overset{tea}{\rightarrow} \boxed{TD} \end{array} \right\}$$

(Note that states *coin* $\rightarrow$ *tea* $\rightarrow$ *TD* and *coffee* $\rightarrow$ *TD* are not reachable from *TD*.) This gives us the finite state machine shown in figure 1.5. It is important to note that not all CSP expressions have finite operational representations. Some simple examples of infinite state processes are given in [Roscoe 1994].

It is straightforward to derive the failures and divergences of a process from its state transition system. However there may be many operational representations of a single process, just as there may be many algebraic representations. It is shown in [Roscoe 1988a] that the denotational semantics of CSP, *i.e.* the failures-divergences model, and the operational semantics are *congruent*. This means that if Φ is the mapping from operational semantics to failures and divergences, and **op** is the representation of a CSP operation in the operational model, and *op* is the representation of the same CSP operation in the denotational model then for any process P in the operational model we have

$$\Phi(\mathbf{op}(P)) = op(\Phi(P))$$

This means that the behaviour of a process predicted by its failures and divergences will be the same as that which can be observed of its operational representation. So we may use the operational semantics of CSP in order to prove properties of process behaviour which are phrased in the Failures-Divergences model. This feature turns out to be particularly useful when the operational representation of a process is finite although its failures and divergences are infinite, as is usually the case in practice. More on this in chapter 3.

Figure 1.6: Operational Semantics for CSP I

Primitive processes:

$$\frac{}{\mathit{SKIP} \xrightarrow{\checkmark} \mathit{STOP}}$$

Prefix:

$$\frac{}{(a \rightarrow P) \xrightarrow{a} P}$$

External choice:

$$\frac{P \xrightarrow{a} P'}{(P \,\square\, Q) \xrightarrow{a} P'} a \neq \tau$$

$$\frac{Q \xrightarrow{a} Q'}{(P \,\square\, Q) \xrightarrow{a} Q'} a \neq \tau$$

$$\frac{P \xrightarrow{\tau} P'}{(P \,\square\, Q) \xrightarrow{\tau} (P' \,\square\, Q)}$$

$$\frac{Q \xrightarrow{\tau} Q'}{(P \,\square\, Q) \xrightarrow{\tau} (P \,\square\, Q')}$$

Internal choice:

$$\frac{}{(P \sqcap Q) \xrightarrow{\tau} P}$$

$$\frac{}{(P \sqcap Q) \xrightarrow{\tau} Q}$$

Sequential Composition:

$$\frac{P \xrightarrow{a} P'}{(P\,;\,Q) \xrightarrow{a} (P'\,;\,Q)} a \neq \checkmark$$

$$\frac{P \xrightarrow{\checkmark} P'}{(P\,;\,Q) \xrightarrow{\tau} Q}$$

Figure 1.7: Operational Semantics for CSP II

Parallel Composition:

$$\frac{P \xrightarrow{a} P'}{P \;|[\, A \mid B \,]|\; Q \xrightarrow{a} P' \;|[\, A \mid B \,]|\; Q}\, a \in (A - B - \{\surd\}) \cup \{\tau\}$$

$$\frac{Q \xrightarrow{a} Q'}{P \;|[\, A \mid B \,]|\; Q \xrightarrow{a} P \;|[\, A \mid B \,]|\; Q'}\, a \in (B - A - \{\surd\}) \cup \{\tau\}$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \;|[\, A \mid B \,]|\; Q \xrightarrow{a} P' \;|[\, A \mid B \,]|\; Q'}\, a \in (A \cap B) \cup \{\surd\}$$

Interleaving:

$$\frac{P \xrightarrow{a} P'}{P \;|||\; Q \xrightarrow{a} P' \;|||\; Q}\, a \neq \surd$$

$$\frac{Q \xrightarrow{a} Q'}{P \;|||\; Q \xrightarrow{a} P \;|||\; Q'}\, a \neq \surd$$

$$\frac{P \xrightarrow{\surd} P' \quad Q \xrightarrow{\surd} Q'}{P \;|||\; Q \xrightarrow{\surd} P' \;|||\; Q'}$$

Hiding:

$$\frac{P \xrightarrow{a} P'}{(P \setminus A) \xrightarrow{\tau} (P' \setminus A)}\, a \in A \cup \{\tau\}$$

$$\frac{P \xrightarrow{a} P'}{(P \setminus A) \xrightarrow{a} (P' \setminus A)}\, a \notin A \cup \{\tau\}$$

Alphabet Transformation:

$$\frac{P \xrightarrow{a} P'}{f(P) \xrightarrow{f(a)} f(P')}$$

Recursion:

$$\frac{}{\mu\, X \bullet F(X) \xrightarrow{\tau} F(\mu\, X \bullet F(X))}$$

1.4 Language Extensions

The core CSP syntax described above is very abstract, and lacks certain useful features found in conventional sequential and parallel programming languages. The extensions outlined below are useful for writing more detailed specifications.

Sometimes we define processes with parameters, such as

$$\mathit{BUFF}(\mathit{in}, \mathit{out}) = \mathit{in} \rightarrow \mathit{out} \rightarrow \mathit{BUFF}(\mathit{in}, \mathit{out})$$

This is a process-schema, rather than an actual process. It defines a CSP process for each combination of parameter values. CSP parameters may be integers, real numbers, events, sets, matrices, *etc.*

A *communication* is a special type of event described by a pair $c.v$, where c is the name of the channel on which the event takes place, and v is the value of the message that is passed.

The set of messages communicable on channel c is defined

$$\mathit{type}(c) = \{v | c.v \in \Sigma\}$$

Input and output are defined as follows. A process which first outputs v on channel c, then behaves like P is defined

$$(c!v \rightarrow P) = (c.v \rightarrow P)$$

Outputs may involve expressions of parameters such as $P(x) = c!x^2 \rightarrow Q$. The expressions are evaluated according to the appropriate laws.

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

$$(c?x \rightarrow P(x)) = \Box_{v:\mathit{type}(c)}\,(c.v \rightarrow P(v))$$

It is usual for a communication channel to be used by at most two processes at any time: one for input and the other for output. However this restriction is not enforced in the modern version of CSP.

Another important aspect to real programming languages is the use of conditionals. Let b be a boolean expression (either true or false). Then

$$P \lhd b \rhd Q \qquad (\text{“}P \text{ if } b \text{ else } Q\text{”})$$

is a process which behaves like P if the value of expression b is true, or like Q otherwise.

These extensions are useful for specifying fine detail during the later stages of program refinement. At the design stage we shall tend to stick to abstract, non-deterministic definitions of processes. The deadlock issue will be addressed at this point. In this way we shall build robust programs for which deadlock-freedom cannot be compromised by implementation decisions made at a later stage.

1.5 Deadlock Analysis

Terminology and Fundamental Results

The problem of the “deadly embrace” was first reported by E. W. Dijkstra relating to resource sharing[Dijkstra 1965]. It has proved a popular topic of research ever since. Most of the early work was presented in an informal manner, for instance [Chandy and Misra 1979], largely due to the lack of a suitable mathematical model for concurrency at the time. But in 1985-86, S.D.Brookes, A.W.Roscoe and N.Dathi presented some powerful techniques for reasoning about deadlock based upon the solid mathematical foundations of CSP. A major benefit of their approach is that it relies only on local analysis of pairs of processes, and simple topological properties of the network configuration. This makes it suitable for analysing networks of arbitrary size. The terminology introduced here is taken from the following sources: [Brookes and Roscoe 1985b], [Roscoe and Dathi 1986], and [Brookes and Roscoe 1991].

We consider a *network*, V, which is a list of processes $\langle P_1, .., P_n \rangle$. Associated with each process P_i is an alphabet αP_i. The corresponding process, $\|_{i=1}^{n} \ (P_i, \alpha P_i)$, is denoted *PAR*$(V)$.

We view a network as consisting of a static collection of everlasting components. Parallel programs do not need to terminate to produce useful results, and deadlock analysis is simplified if we can cast termination aside. Henceforth we shall only consider processes which are non-terminating, *i.e.* they never perform the event $\surd$ (although they may still be constructed from sub-processes which do terminate).

A process P can deadlock after trace s if and only if $(s, \Sigma) \in$ *failures*(P). We say P is *deadlock-free* if

$$\forall\, s : \mathit{traces}(P). \quad (s, \Sigma) \notin \mathit{failures}(P)$$

Note that this definition of deadlock-freedom also excludes any process which can diverge (by axiom 7 of the failures model), which seems reasonable as divergence is every bit as undesirable a phenomenon as deadlock. Network V is said to be deadlock-free if the process *PAR*(V) is deadlock-free.

The following lemma describes how individual sequential processes may be constructed free of deadlock. Used in conjunction with the algebraic laws of CSP, it also enables us to prove deadlock-freedom for certain small networks of processes by manipulation into a sequential form. Unfortunately this technique does not scale at all well to large networks because the resulting CSP terms usually increase in length in a manner exponentially proportional to the number of processes which constitute the network.

Lemma 1 (Roscoe-Dathi 1986) *Suppose the definition of the process P uses only the following syntax*

$$\begin{array}{lll} \mathit{Process} & :== & \mathit{SKIP} \quad \Big| \\ & & \mathit{event} \rightarrow \mathit{Process} \quad \Big| \end{array}$$

$$\begin{array}{ll} \textit{Process} \,;\, \textit{Process} & | \\ \textit{Process} \sqcap \textit{Process} & | \\ \textit{Process} \,\square\, \textit{Process} & | \\ f(\textit{Process}) & | \\ \textit{name} & | \\ \mu\, \textit{name} \bullet \textit{Process} & \end{array}$$

where "name" denotes a process variable, but P contains no free process variables, is divergence-free, and every occurrence of SKIP in P is directly or indirectly followed by a ";" to prevent successful termination. Then P is deadlock-free$\Box$

If every component process P_i of a network is deadlock-free we say that the network is *busy*. A network is *triple-disjoint* if no event requires the participation of more than two processes. We shall restrict our attention to networks which are both busy and triple disjoint. This will enable us to analyse networks for deadlock-freedom purely by the local analysis of neighbouring pairs of processes.

We observe the convention that communication channels are used in only one direction and between only two processes. We call this the *I/O* convention. This guarantees that whenever two processes are ready to communicate on a particular channel then the communication can go ahead. Sometimes, when we are not concerned about the data which is communicated, it is convenient to substitute a channel name for communication events in a process description. For instance, we might write $a \rightarrow \textit{SKIP}$ instead of $a?x \rightarrow \textit{SKIP}$. This is known as *abstraction*. If we can prove freedom from deadlock for an abstracted version of a network then the property will also hold for the original. A formal treatment of this is given in [Roscoe 1995].

A *network state* of V is defined as a trace s of *PAR*(V), together with a sequence $\langle X_1, .., X_n \rangle$ of refusal sets X_i, such that for each i,

$$(s \upharpoonright \alpha P_i, X_i) \in \textit{failures}(P_i)$$

We say that a network state is *maximal* if each of its refusal sets is maximal, *i.e.*, if $(s, \langle X_1, .., X_n \rangle)$ is a maximal state of V then for each process P_i there is no failure $(s \upharpoonright \alpha P_i, Y)$ such that $X_i \subset Y$.

When we consider deadlock properties we find that all the relevant information is carried by the maximal network states, as the more events that an individual process refuses, the more likely deadlock becomes. So from now on all network states will be taken to be maximal, as this simplifies the analysis.

There is a close relationship between a network state and the operational states of the processes within. Suppose we visualise a network as a collection of state transition systems – one representing each process. A network state is then rather like a cross-section of the network. The trace s tells us what each process has done so far, and each refusal set X_i corresponds to a particular stable state of process P_i, telling us exactly what it is refusing to do on the next step. For instance the network

$$\langle \mathit{VM}, \mathit{TD}\rangle$$

for which the transition systems are illustrated in figure 1.5, has a network state

$$(\langle \mathit{coin}\rangle, \langle\{\mathit{coffee}\}, \{\mathit{coin,coffee}\}\rangle)$$

which corresponds to the situation where the tea drinker has inserted a coin into the vending machine. The vending machine is then in operational state *tea* $\rightarrow$ *VM* $\Box$ *coin* $\rightarrow$ *coffee* $\rightarrow$ *VM*, refusing event *coffee* and prepared to accept $\{$*coin, tea*$\}$. The tea drinker is in operational state *tea* $\rightarrow$ *TD*, refusing $\{$*coin, coffee*$\}$ and prepared only to accept *tea*.

The following lemma characterises network states where deadlock is present.

Lemma 2 (Roscoe-Dathi 1986) *PAR(V) can deadlock after trace s if and only if there is a network state $(s, \langle X_1, .., X_n\rangle)$ such that*

$$\bigcup_{i=1}^{n}(X_i \cap \alpha P_i) = \bigcup_{i=1}^{n} \alpha P_i$$

$\Box$

This follows easily from the definitions. Such a state is called a *deadlock state*.

Suppose that, in a particular state $\sigma = (s, \langle X_1, .., X_n\rangle)$ there is a process P_i which is ready to communicate with P_j, *i.e.*

$$(\alpha P_i - X_i) \cap \alpha P_j \neq \{\}$$

We say that P_i is making a *request* to P_j in state σ, which is written

$$P_i \xrightarrow{\sigma} P_j.$$

We say that this request is *ungranted* if also P_j refuses to respond to P_i's request: *i.e.*

$$\alpha P_i \cap \alpha P_j \subseteq X_i \cup X_j$$

This is written

$$P_i \xrightarrow{\sigma} \bullet P_j.$$

The set of shared events within a network is known as its *vocabulary*, written Λ.

$$\Lambda = \bigcup_{i \neq j}(\alpha P_i \cap \alpha P_j)$$

Sometimes we are only interested in ungranted requests from P_i to P_j when neither process is able to communicate outside the vocabulary of the network, *i.e.* in addition to the above

$$(\alpha P_i - X_i) \cup (\alpha P_j - X_j) \subseteq \Lambda.$$

Then we say that P_i is making an ungranted request to P_j *with respect to* Λ. We write

$$P_i \overset{\sigma,\Lambda}{\to} \bullet P_j.$$

We say that P_i is *blocked* in network state σ of V if

$$\exists\, j.\quad P_i \overset{\sigma}{\to} P_j \quad \text{and} \quad P_i \overset{\sigma}{\to} P_k \Longrightarrow P_i \overset{\sigma,\Lambda}{\to} \bullet P_k$$

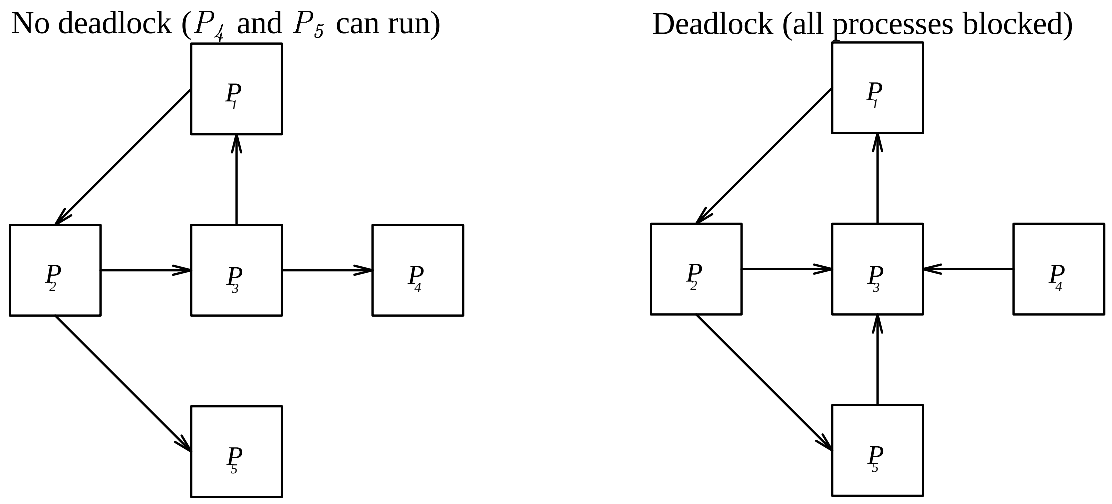
which means that P_i is ready to engage in a communication with at least one other process, but no process that P_i wishes to communicate with is able to do so. Neither P_i nor any process that it wishes to communicate with is able to perform any event outside the vocabulary of V.

The following lemma is derived from the above definitions:

Lemma 3 (Roscoe-Dathi 1986) *If σ is a state of a triple disjoint, busy network V, then σ is a deadlock state if, and only if, every process in V is blocked in state σ*$\square$

This result may be interpreted graphically. We define the *wait-for digraph* of a network state as follows. It is a digraph which has a vertex for every process P_i, and arcs from any blocked process to each process for which it is waiting. Figure 1.8 shows examples of wait-for digraphs, which illustrate lemma 3.

Figure 1.8: Wait-for Digraphs


No deadlock (P_4 and P_5 can run)
Deadlock (all processes blocked)
P_1
P_2
P_3
P_4
P_5
P_1
P_2
P_3
P_4
P_5

We may deduce an interesting feature of deadlock states. Consider a deadlock state of a busy, triple-disjoint network V. By lemma 3 there is at least one ungranted request from every process, with respect to the vocabulary of V. So starting with any process P_{i_1}, we may build an arbitrarily long sequence of ungranted requests as follows:

$$P_{i_1} \overset{\sigma,\Lambda}{\to} \bullet P_{i_2} \overset{\sigma,\Lambda}{\to} \bullet P_{i_3} \ldots$$

As there are only a finite number of processes, P_i, this sequence must eventually arrive back at a process that it has already visited, *i.e* there is a *cycle of ungranted requests*

$$P_{i_m} \overset{\sigma,\Lambda}{\rightarrow} \bullet P_{i_{m+1}} \overset{\sigma,\Lambda}{\rightarrow} \bullet . . \overset{\sigma,\Lambda}{\rightarrow} \bullet P_{i_{m+k}} \overset{\sigma,\Lambda}{\rightarrow} \bullet P_{i_m}$$

Hence we have proved the following result.

Theorem 1 *In any deadlock state of a triple disjoint, busy network there is a cycle of ungranted requests with respect to its vocabulary*$\Box$.

Roscoe and Dathi made use of this fact to establish a method for investigating deadlock properties of networks which involves only local checking. The crucial idea behind this technique is as follows. If a function is defined on the states of processes in a network which is strictly decreasing along any chain of ungranted requests, then there can never be a cycle of ungranted requests and hence no deadlock. An example of using this technique will be given in the next chapter.

Theorem 2 (Roscoe-Dathi 1986) *Let* $V = \langle P_1, . ., P_n \rangle$ *be a busy, triple-disjoint network with vocabulary* Λ. *If there exist functions* f_i, *from the failures of each process* P_i *to a strict partial order* $(S, >)$ *such that whenever* $\sigma = (s, \langle X_i, X_j \rangle)$ *is a state of the subnetwork* $\langle P_i, P_j \rangle$

$$P_i \overset{\sigma,\Lambda}{\rightarrow} \bullet P_j \Longrightarrow f_i((s \upharpoonright \alpha P_i, X_i)) > f_j((s \upharpoonright \alpha P_j, X_j))$$

then V *is deadlock-free. Or if there exist similar functions* g_i, *such that*

$$P_i \overset{\sigma,\Lambda}{\rightarrow} \bullet P_j \Longrightarrow g_i((s \upharpoonright \alpha P_i, X_i)) \geq g_j((s \upharpoonright \alpha P_j, X_j))$$

then any deadlock state $\delta = (s, \langle X_1, . ., X_n \rangle)$ *of* V *contains a cycle of ungranted requests,*

$$P_{i_1} \overset{\delta,\Lambda}{\rightarrow} \bullet P_{i_2} \overset{\delta,\Lambda}{\rightarrow} \bullet . . P_{i_m} \overset{\delta,\Lambda}{\rightarrow} \bullet P_{i_1}$$

such that

$$g_{i_1}((s \upharpoonright \alpha P_{i_1}, X_{i_1})) = g_{i_2}((s \upharpoonright \alpha P_{i_2}, X_{i_2})) = . . = g_{i_m}((s \upharpoonright \alpha P_{i_m}, X_{i_m}))\Box$$

The existence of a cycle of ungranted requests does not always mean deadlock has occurred. The cycle might subsequently be broken by the intervention of a process from outside the cycle.

Deadlock-free networks exist that sometimes develop cycles of ungranted requests and this theorem is not sufficiently powerful to prove them so. Dathi's thesis contains a hierarchy of stronger techniques, together with a classification of different levels of deadlock-freedom which they may be used to establish [Dathi 1990].

By treating cycles of length two as a special case we can arrive at a useful extension to theorem 1. We say that two processes P_i and P_j are in *conflict* with respect to Λ in network state σ if each one is trying to communicate with the other, but cannot agree on which event to perform, *i.e.*

$$P_i \overset{\sigma,\Lambda}{\rightarrow} \bullet P_j \wedge P_j \overset{\sigma,\Lambda}{\rightarrow} \bullet P_i$$

A conflict is basically a cycle of ungranted requests of length two. It is said to be *strong* if one of the processes is able to communicate *only* with the other process. *i.e.*

$$((\alpha P_i - X_i) \subseteq \alpha P_j) \vee ((\alpha P_j - X_j) \subseteq \alpha P_i)$$

We call a network where strong conflict can never occur *strong conflict-free*.

Theorem 3 (Brookes-Roscoe 1991) *In any deadlock state of a triple disjoint, busy, strong conflict-free network there is a cycle of ungranted requests with respect to its vocabulary of length greater than two.*

Proof. Consider the wait-for digraph of a deadlock state σ of such a network. Starting at any node P_{i_i} we can form a sequence of arbitrary length

$$P_{i_1} \overset{\sigma,\Lambda}{\rightarrow} \bullet P_{i_2} \overset{\sigma,\Lambda}{\rightarrow} \bullet P_{i_3} ..$$

with the property that P_{i_j}, $P_{i_{j+1}}$, and $P_{i_{j+2}}$ are all distinct for each j. For if $P_{i_{j+1}}$ has an ungranted request back to P_{i_j} the two processes are in conflict and as this cannot be a strong conflict $P_{i_{j+1}}$ must also have an ungranted request to some other process, which may then be selected as $P_{i_{j+2}}$. This sequence will eventually cross itself which means that there must be a cycle of ungranted requests of length greater than two□.

The property of strong conflict-freedom may be established by pairwise analysis of processes in the network and in this way may be checked for networks of arbitrary size.

Brookes and Roscoe used this result to develop another technique for proving deadlock freedom by showing that a cycle of ungranted requests cannot occur. This relies on the processes in the network obeying a rather special condition and so is somewhat in the nature of a *design rule*.

Theorem 4 (Brookes-Roscoe 1991) *Let $V = \langle P_1, ..P_N \rangle$ be a busy, triple-disjoint, strong-conflict-free network such that whenever a process P has an ungranted request to another process Q then Q has previously communicated with P, and has done so more recently than with any other process. It follows that V is deadlock-free.*

Proof. Consider a deadlock state σ of a strong-conflict-free network V. By theorem 3 there must exist a cycle of ungranted requests, of length at least three, as follows:

$$P_{i_1} \overset{\sigma,\Lambda}{\rightarrow} \bullet P_{i_2} \overset{\sigma,\Lambda}{\rightarrow} \bullet \, .. \, P_{i_k} \overset{\sigma,\Lambda}{\rightarrow} \bullet P_{i_1}$$

Now suppose that the most recent communication between two consecutive elements of this cycle was between P_{i_h} and $P_{i_{h+1}}$ (where addition is modulo k – the length of the cycle). Consider the ungranted request from $P_{i_{h-1}}$ to P_{i_h}. P_{i_h} has communicated with $P_{i_{h+1}}$ more recently than it has with $P_{i_{h-1}}$. This means that any strong-conflict-free network which deadlocks does not satisfy the conditions of the theorem. It follows that a network which obeys the conditions of the theorem is deadlock-free□.

This result has been used by Roscoe to develop a complex and sophisticated message routing algorithm [Roscoe 1988b]. A generalisation of the theorem is given in [Roscoe 1995].

Livelock

In high level concurrent programming languages, such as occam, it is conventional for communication channels between two processes to be concealed from the environment. This can potentially cause a form of divergence known as *livelock*. We say that a network is *livelock-free* if it can never perform an infinite sequence of internal or hidden actions, *i.e.*

$$\mathit{divergences}(\mathit{PAR}(V) \setminus \Lambda) = \{\}$$

Roscoe discovered a useful technique (detailed in [Dathi 1990]) for establishing this important property. It is described here in a slightly simplified form.

Theorem 5 (Roscoe 1982) *Suppose* $V = \langle P_1, .., P_N \rangle$ *is a triple-disjoint network of non-divergent processes such that for every* P_i *in* V

$$P_i \setminus (\cup_{j<i}(\alpha P_i \cap \alpha P_j)) \quad \text{is divergence-free}$$

then $\mathit{PAR}(V) \setminus \Lambda$ *is divergence-free*□

In other words, if no process in a network can ever perform an infinite sequence of communications with its predecessors then the network is livelock-free. (This can be proved by induction.) This theorem is found to be useful in many cases, although it requires careful ordering of the processes within the network to be effective.

Network Decomposition

The communication architecture of a triple-disjoint network may be represented by a *communication graph*. This consists of a vertex to represent each process and an edge to connect each pair of processes with overlapping alphabets. The next theorem describes how deadlock analysis of a network may be broken down into the analysis of a collection of smaller components by the removal of disconnecting edges (see appendix B) from the communication graph.

Theorem 6 (Brookes-Roscoe 1991) *Consider the communication graph of a network V with a set of disconnecting edges which separates the network into components*

$$\langle V_1, .., V_k \rangle$$

If each pair of processes joined by a disconnecting edge is conflict-free with respect to Λ and each subnetwork V_j is deadlock-free, then so is V□.

A proof of this theorem is given in [Brookes and Roscoe 1991].

This result is useful for the hierarchical construction of networks. It offers a safe way of connecting subsystems together without introducing any risk of deadlock.

Hiding

An important feature of reasoning with CSP is the use of the concealment operator, which enables us to hide those events that we are not interested in. This can greatly simplify deadlock analysis of a network.

Lemma 4 *If $P \setminus C$ is deadlock-free, then P is deadlock-free*□

Used with CSP law 1.30, this result enables us to add extra external communications to the component processes of a deadlock-free network. Deadlock freedom will be preserved as long as the behaviour of each component is unchanged when these events are concealed.

Lemma 5 *Suppose V is a network $\langle P_1, .., P_n \rangle$. Let V' be a network $\langle {P_1}', .., {P_n}' \rangle$, such that*

$$P_i' \setminus (\alpha P_i' - \alpha P_i) = P_i$$
$$i \neq j \implies (\alpha P_i' - \alpha P_i) \cap \alpha P_j' = \{\}$$

then

$$PAR(V) = PAR(V') \setminus \bigcup_{i=1}^{n} (\alpha P_i' - \alpha P_i)$$

Furthermore, if V is deadlock-free then so is V'.

Proof.

$$\begin{aligned} PAR(V) &= \|_{i=1}^{n} (P_i, \alpha P_i) \\ &= \|_{i=1}^{n} (P_i' \setminus (\alpha P_i' - \alpha P_i), \alpha P_i) \\ &= (\|_{i=1}^{n} (P_i', \alpha P_i')) \setminus \bigcup_{i=1}^{n} (\alpha P_i' - \alpha P_i) \quad \text{by application of law 1.30} \\ &= PAR(V') \setminus \bigcup_{i=1}^{n} (\alpha P_i' - \alpha P_i) \end{aligned}$$

It now follows from lemma 4 that V' inherits deadlock-freedom from V□

Refinement

CSP processes are related by a complete partial order $\sqsubseteq$, which we described in section 1.2. $Q \sqsubseteq P$ means that every behaviour pattern that is possible for P is also possible for Q. We say that Q is a *specification* for P, and that P is a *refinement* of Q.

The operation of parallel composition with any particular process is known to be *monotonic, i.e.* order-preserving, with respect to this partial ordering (in fact all CSP operations are). This leads us to the following observation.

Lemma 6 *Suppose that* $V = \langle P_1, .., P_N \rangle$ *and* $V' = \langle P'_1, .., P'_N \rangle$ *are networks where*

$$\forall\, i : \{1, .., N\}. \quad P_i \sqsubseteq P'_i$$

then $\mathit{PAR}(V) \sqsubseteq \mathit{PAR}(V')\Box$

In particular this means that if V is deadlock-free then so is V'. Similarly if V is livelock-free then so is V'.

This result makes an important statement about the way in which we should design and build concurrent systems, which has already been hinted at. At the design stage we should specify each of our components in as abstract a manner as is possible. Important properties of the system as a whole which are shown to hold at this stage, such as freedom from deadlock and divergence, will be preserved as we gradually refine each component into the finished product.