Chapter 2
Understanding CSP

In the previous chapter we learned how to build simple CSP descriptions of sequential systems. We also saw that the specifications of CSP processes are frequently other CSP processes. In this chapter we see some of the methods we can use to prove processes equal to each other and understand how they behave.

Since the early 1980s there have been three complementary approaches to understanding the semantics of CSP programs. These are *algebra*, where we set out laws that the syntax is assumed to satisfy, *behavioural models* such as traces that form the basis of refinement relations and other things, and *operational models*, which try to understand all the actions and decisions that process implementations can make as they proceed. This chapter introduces the reader to the basics of each of these.

2.1 Algebra

An algebraic law is the statement that two expressions, involving some operators and identifiers representing arbitrary processes (and perhaps other things such as events) are equal. By 'equal', we mean that the two sides are essentially the same: for CSP this means that their communicating behaviours are indistinguishable by the environment.

Everyone with the most basic knowledge of arithmetic or set theory is familiar with the sort of algebra we are now talking about. There are a number of basic patterns that many laws conform to; the following are a few familiar examples illustrating these:

$x + y = y + x$	a *commutative*, or *symmetry* law
$x \times y = y \times x$	ditto
$x \cup y = y \cup x$	ditto
$(x + y) + z = x + (y + z)$	*associativity*
$(x + y) \times z = (x \times z) + (y \times z)$	(right) *distributive* law
$0 + x = x$	*unit* law (0 is a left unit of $+$)

A.W. Roscoe, *Understanding Concurrent Systems*, Texts in Computer Science,
DOI 10.1007/978-1-84882-258-0_2, © Springer-Verlag London Limited 2010

$$\emptyset \cap x = \emptyset \qquad \textit{zero} \text{ law } (\emptyset \text{ is a left zero of } \cap)$$

$$x \cup x = x \qquad \textit{idempotence}$$

We will find all of these patterns and more amongst the laws of CSP. Let us now consider what laws ought to relate the CSP operators we have met so far: prefixing, external choice, nondeterministic choice, and conditionals.

We would expect the choice between P and itself to be the same as P, the choice between P and Q the same as that between Q and P, and the choice between three processes P, Q and R to be the same however bracketed. And this will all apply whether we are talking about internal (nondeterministic) choice or external choice. In other words, these properties all hold whether the environment or the process gets to decide which path is chosen. Thus there are idempotence,[1] symmetry and associative laws for both $\Box$ and $\sqcap$:

$$P \Box P = P \qquad \langle \Box\text{-idem}^* \rangle \quad (2.1)$$

$$P \sqcap P = P \qquad \langle \sqcap\text{-idem} \rangle \quad (2.2)$$

$$P \Box Q = Q \Box P \qquad \langle \Box\text{-sym} \rangle \quad (2.3)$$

$$P \sqcap Q = Q \sqcap P \qquad \langle \sqcap\text{-sym} \rangle \quad (2.4)$$

$$P \Box (Q \Box R) = (P \Box Q) \Box R \qquad \langle \Box\text{-assoc} \rangle \quad (2.5)$$

$$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R \qquad \langle \sqcap\text{-assoc} \rangle \quad (2.6)$$

These three laws[2] (idempotence, symmetry and associativity) are just what is needed to ensure that the nondeterministic choice operator over sets, $\bigsqcap S$, makes sense (see Exercise 2.5). For what we mean by this (for finite $S = \{P_1, \ldots, P_n\}$) must be the same as $P_1 \sqcap \cdots \sqcap P_n$, and since sets are oblivious to the repetition and order of their elements, we need $\sqcap$ to be idempotent (ignoring repetitions), symmetric (ignoring order) and associative (so that bracketing is not required). Clearly the operator $\bigsqcap S$ has laws that we could write down too, but these would not follow such conventional forms as it is not an ordinary binary operator. We will always feel at liberty to rewrite $\bigsqcap\{P_1, \ldots, P_n\}$ as

$$P_1 \sqcap \cdots \sqcap P_n$$

and similar, without formal recourse to laws.

[1] The reader will notice that $\langle \Box$ -idem$^* \rangle$ has a $*$ in its name. This is because, while this law is true in the most standard models of CSP, there are models in the hierarchy discussed in Chaps. 11 and 12 where it is not true. This point is examined in Chap. 13. We will follow the $*$-ing convention throughout the book: any law without one is true throughout the hierarchy of behavioural equivalences for CSP; sometimes $*$ will be replaced by an indication of the class of models in which the law is true.

[2] We have given each law a descriptive name and a number. The many laws that are common to this book and TPC are given the same name in each, but not usually the same number. In each book the number refers to the chapter in which the law was first asserted, and the number within that chapter.

Notice that each law has been given a name and a number to help us refer to it later. Laws will be quoted throughout this section and Chap. 13. A consolidated list can be obtained from this book's web-site.

If we have any operator or construct $F(\cdot)$ which, in any 'run', takes at most one copy of its argument, then it is natural to expect that $F(\cdot)$ will be *distributive*, in that

$$F(P \sqcap Q) = F(P) \sqcap F(Q)$$

$$F(\bigsqcap S) = \bigsqcap\{F(P) \mid P \in S\}$$

(i.e., the operator distributes over $\sqcap$ and distributes through $\bigsqcap$). In the first of these, this is because the argument on the left-hand side can act like P or like Q, so the effect of running $F(P \sqcap Q)$ must be either like running $F(P)$ or like running $F(Q)$. Since that is precisely the set of options open to $F(P) \sqcap F(Q)$, the two sides are equal. The second is just the same argument applied to an arbitrary, rather than two-way, choice. All of the operators, other than recursion, which we have described so far fall into this category. The distributive laws for some of the constructs seen to date are:

$$P \,\square\, (Q \sqcap R) = (P \,\square\, Q) \sqcap (P \,\square\, R) \qquad \langle\square\text{-dist}\rangle \quad (2.7)$$

$$P \,\square \bigsqcap S = \bigsqcap\{P \,\square\, Q \mid Q \in S\} \qquad \langle\square\text{-Dist}\rangle \quad (2.8)$$

$$a \to (P \sqcap Q) = (a \to P) \sqcap (a \to Q) \qquad \langle\text{prefix-dist}\rangle \quad (2.9)$$

$$a \to \bigsqcap S = \bigsqcap\{a \to Q \mid Q \in S\} \qquad \langle\text{prefix-Dist}\rangle \quad (2.10)$$

$$?x : A \to (P \sqcap Q) = (?x : A \to P) \sqcap (?x : A \to Q) \qquad \langle\text{input-dist}\rangle \quad (2.11)$$

$$?x : A \to \bigsqcap S = \bigsqcap\{?x : A \to Q \mid Q \in S\} \qquad \langle\text{input-Dist}\rangle \quad (2.12)$$

Note that there is a pair for each. In fact, of course, the second of each pair implies the first. An operator that distributes over $\bigsqcap S$ might be called *fully* distributive, whereas one that distributes over $\sqcap$ might be called *finitely* distributive. In future, we will generally only quote one of each pair of distributive laws explicitly, to save space. It may be assumed in CSP that they both hold if either does, noting that they are always equivalent when infinite (unbounded) nondeterminism is banned.

In general, an operator $F(P)$ should be distributive unless it has the chance, in a single run, to compare two different copies of P. If it can make such a comparison then $F(P \sqcap Q)$ may be different from $F(P) \sqcap F(Q)$. In the first case the two copies it compares may be different (one P and one Q) whereas, in the second, they must be the same (whichever they are). This is why recursion is not distributive. We only have to consider a simple example like

$$\mu p.((a \to p) \sqcap (b \to p)) \quad \text{and} \quad (\mu p.a \to p) \sqcap (\mu p.b \to p)$$

where the left-hand side can perform any sequence of as and bs (at its own choice) while the right-hand side has to be consistent: once it has communicated one a it must keep on doing as.

⟨□-dist⟩ is actually only one of the two distributive laws for □ over ⊓. The other one (the right distributive law) follows from this one and ⟨□-sym⟩. There are also (left and right) distributive laws for ⊓ over itself—provable from the existing set of laws for ⊓ (see Exercise 2.1 below).

Distributivity of most operators, and in particular $a \to \cdot$, over ⊓ is something that is true in CSP but frequently not in other process algebras. The main reason for this is the type of models—traces etc.—that we use for CSP. Each observation these make of a process records only a single evolution through time, with no branching. It should not be too hard to see that if we could *freeze* a process after a non-empty trace and run several copies of the result, then we could distinguish between $(a \to b \to \mathit{STOP}) \sqcap (a \to c \to \mathit{STOP})$ and $a \to (b \to \mathit{STOP} \sqcap c \to \mathit{STOP})$. If we froze the second version after a and ran it twice then one copy might perform b and the other c, but since the nondeterministic choice in the first version gets committed before a, the two frozen copies made after a are bound to behave the same. We will learn more about this in Chaps. 9 and 10. Observations of a single evolution through time are sometimes called a *linear* or *linear time* observation, in contrast to *branching time* where we can freeze or backtrack.

There is a close relationship between this last argument and the earlier one that an operator which uses several copies of an argument need not be distributive. We can conclude that copying arguments is incompatible with distributivity whether it is done by an operator or by the underlying model.

There is a further law relating the two forms of choice whose motivation is much more subtle. Consider the process $P \sqcap (Q \,\square\, R)$. It may *either* behave like P *or* offer the choice between Q and R. Now consider

$$(P \sqcap Q) \,\square\, (P \sqcap R)$$

a process which the distributive laws of □ can expand to

$$(P \,\square\, P) \sqcap (P \,\square\, R) \sqcap (Q \,\square\, P) \sqcap (Q \,\square\, R)$$

The first of these four equals P by ⟨□-idem*⟩. It follows that the first and last alternatives provide all the options of the first process. Every behaviour of the second and third is possible for one of the other two: every set of events they reject initially is also rejected by P (for they offer the choice of the first actions of P and another process), and every subsequent behaviour belongs to one of P, Q and R. It is therefore reasonable[3] to assert that the processes on the left- and right-hand sides below are equal. In other words, ⊓ distributes over □, at least in the most used models of CSP.

$$P \sqcap (Q \,\square\, R) = (P \sqcap Q) \,\square\, (P \sqcap R) \qquad \langle\sqcap\text{-}\square\text{-dist}^*\rangle \quad (2.13)$$

[3] In some sense this is the most important law that characterises the most standard equivalences for CSP. As we can see from the $*$, there are models where it is not true.

The following is the chief law relating prefixing and external choice. It says that if we give the environment the choice between processes offering A and B, then we get a process offering $A \cup B$ whose subsequent behaviour depends on which of A and B the first event belongs to:

$$\begin{aligned}&(?x : A \to P) \,\Box\, (?x : B \to Q)\\&\quad = ?x : A \cup B \to ((P \sqcap Q) \lessdot x \in A \cap B \gtrdot (P \lessdot x \in A \gtrdot Q))\end{aligned} \qquad \langle\Box\text{-step}\rangle \quad (2.14)$$

We have called this a *step* law because it allows us to compute the first step of the combination's behaviour (i.e., the selection of initial actions plus the process that succeeds each action) from the first-step behaviour of the processes we are combining.[4]

STOP is the process that offers no choice of initial actions. This can of course be written as the prefix-choice over an empty set:

$$STOP = ?x : \emptyset \to P \qquad \langle STOP\text{-step}\rangle \quad (2.15)$$

It is an immediate consequence of the last two laws that

$$STOP \,\Box\, (?x : A \to P) = ?x : A \to P$$

Of course we would expect that the external choice of *any* process with *STOP* would have no effect. This gives us our first unit law:

$$STOP \,\Box\, P = P \qquad \langle\Box\text{-unit}\rangle \quad (2.16)$$

(There is no need for a right unit law as well as this one, since it is easily inferred from this one and the symmetry law ⟨□-sym⟩.)

Conditional choice is idempotent and distributive:

$$P \lessdot b \gtrdot P = P \qquad \langle\lessdot\cdot\gtrdot\text{-idem}\rangle \quad (2.17)$$

$$(P \sqcap Q) \lessdot b \gtrdot R = (P \lessdot b \gtrdot R) \sqcap (Q \lessdot b \gtrdot R) \qquad \langle\lessdot\cdot\gtrdot\text{-dist-l}\rangle \quad (2.18)$$

$$R \lessdot b \gtrdot (P \sqcap Q) = (R \lessdot b \gtrdot P) \sqcap (R \lessdot b \gtrdot Q) \qquad \langle\lessdot\cdot\gtrdot\text{-dist-r}\rangle \quad (2.19)$$

Left and right distributive laws are required here because conditional choice is not symmetric.

[4] From here on, in quoting laws about prefixed processes, we will usually refer only to the form $?x : A \to P$. The others, namely $a \to P$ and $c?x : A \to P$ (and the more complex forms for multi-part events discussed above) can be transformed into this form easily, and so quoting a lot of extra laws to deal with them would serve no particular purpose.

The conditional behaviour is brought out by the following pair of laws:

$$P \lhd \mathit{true} \rhd Q = P \qquad \langle \lhd\text{true}\rhd\text{-id}\rangle \tag{2.20}$$

$$P \lhd \mathit{false} \rhd Q = Q \qquad \langle \lhd\text{false}\rhd\text{-id}\rangle \tag{2.21}$$

There are other laws in which conditional choice interacts with boolean operators on the condition(s), but we do not attempt to enumerate them here (though see the exercises below). One interesting class of laws is that almost all operators distribute over this form of choice as well as over $\sqcap$. The only ones that do not are ones (in particular prefix-choice) that may modify the bindings of identifiers used in the boolean condition. An example of a law that does hold is

$$P \Box (Q \lhd b \rhd R) = (P \Box Q) \lhd b \rhd (P \Box R) \qquad \langle \lhd \cdot \rhd\text{-}\Box\text{-dist}\rangle \tag{2.22}$$

while the failure of this distribution in the presence of binding constructs is illustrated by

$$\begin{aligned} &?x : \mathbb{N} \to ?x : \mathbb{N} \to (P \lhd x \text{ is even} \rhd Q) \neq \\ &?x : \mathbb{N} \to ((?x : \mathbb{N} \to P) \lhd x \text{ is even} \rhd (?x : \mathbb{N} \to Q)) \end{aligned}$$

since the distribution of $\lhd x$ is even$\rhd$ through the inner prefix-choice results in x being bound to the first input rather than the second.

The fundamental law of *recursion* is that a recursively defined process satisfies the equation defining it. Thus the law is (in the case of equational definitions) just a part of the program. For the μ form of recursion this law is

$$\mu p.P = P[\mu p.P/p] \qquad \langle \mu\text{-unwind}\rangle \tag{2.23}$$

where the notation $Q[R/p]$ means the substitution of the process R for all free (i.e., not bound by some lower-level recursion) occurrences of the process identifier p. (We will sometimes denote the set of *all* a term's free identifiers, both process and non-process, as $\mathit{fv}(P)$.)

We have already noted that recursion fails to be distributive.

Laws of the sort seen in this section serve several functions: they provide a useful way of gaining understanding and intuition about the intended meaning of constructs, they can (as we will see later) be useful in proofs about CSP processes, and finally, if presented and analysed highly systematically, they can be shown to completely define the meaning, or *semantics* of language constructs (in a sense we are not yet in a position to appreciate but which is fully explained in Chap. 13). Whenever we introduce a new operator in later chapters, we will use some of its laws to help explain how it behaves.

Exercise 2.1 Using the laws quoted in the text for $\sqcap$, prove that it distributes over itself (i.e., that $P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap (P \sqcap R)$).

Exercise 2.2 Suggest some laws for $\bigsqcap S$ and how it relates to $\sqcap$.

Exercise 2.3 Write down the left and right distributive laws of $\cdot \lhd b \rhd \cdot$ through $\sqcap$.

Exercise 2.4 Use $\langle \Box\,\textsf{-step}\rangle$ and other laws given above to prove that

$$(?x : A \to P) \,\Box\, (?x : A \to Q) = (?x : A \to P) \sqcap (?x : A \to Q)$$

Exercise 2.5 Suppose we try to extend the binary operator $\oplus$ (e.g. $\sqcap$) to finite non-empty sets by defining

$$\bigoplus\{P_1, \ldots, P_n\} = P_1 \oplus (P_2 \oplus \cdots (P_{n-1} \oplus P_n) \cdots)$$

Show that this makes sense (i.e., the value of $\bigoplus S$ is independent of the way S is written down) only if $\oplus$ is idempotent, symmetric and associative. For example, it must be idempotent because $\{P, P\} = \{P\}$, and hence $P \oplus P = \bigoplus\{P, P\} = \bigoplus\{P\} = P$.

In this case prove that $\bigoplus(A \cup B) = (\bigoplus A) \oplus (\bigoplus B)$ for any non-empty A and B.

What additional algebraic property must $\oplus$ have to make $\bigoplus \emptyset$ well defined in such a way that this union law remains true? [*Hint*: $\Box$ *has this property but* $\sqcap$ *does not*.] What is then the value of $\bigoplus \emptyset$?

Exercise 2.6 Complete the following laws of the conditional construct by filling in the blank(s) $(\ldots)$ in each

(a) $P \lhd \neg b \rhd Q = \cdots \lhd b \rhd \cdots$
(b) $P \lhd b \rhd (Q \lhd b \wedge c \rhd R) = \cdots \lhd b \rhd R$
(c) $(P \lhd c \rhd Q) \lhd b \rhd R = \cdots \lhd c \rhd \cdots$

2.2 The Traces Model and Traces Refinement

Recall that we defined that P *trace-refines* Q, written $Q \sqsubseteq_T P$, to mean that every finite trace of P is one of Q. Note that a trace is a sequence of visible actions in the order they are observed, but that the actual time these actions happen are not recorded.

It is natural to model an untimed CSP process by the set of all traces it can perform. It turns out that recording only *finite* traces is sufficient in the majority of cases—after all, if u is an infinite trace then all its finite *prefixes* (initial subsequences) are finite traces—and for the time being we will only record finite ones. In Sect. 9.5.2 and Chap. 12 we will see the subtle distinctions infinite traces can make in some cases—though at some cost in terms of theoretical difficulty. As we will see later in this book, for example Chaps. 6, 11 and 12, there are many other choices of behaviours we can use to model processes.

2.2.1 Working out *traces*(P)

Recall that, any process P, $\mathit{traces}(P)$ is the set of all its finite traces—members of Σ^*, the set of finite sequences of events. For example:

- $\mathit{traces}(\mathit{STOP}) = \{\langle\rangle\}$—the only trace of the process that can perform no event is the empty trace;
- $\mathit{traces}(a \to b \to \mathit{STOP}) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\}$—this process may have communicated nothing yet, performed an a only, or an a and a b;
- $\mathit{traces}((a \to \mathit{STOP}) \mathbin{\Box} (b \to \mathit{STOP})) = \{\langle\rangle, \langle a\rangle, \langle b\rangle\}$—here there is a choice of first event, so there is more than one trace of length 1;
- $\mathit{traces}(\mu p.((a \to p) \mathbin{\Box} (b \to \mathit{STOP}))) = \{\langle a\rangle^n, (\langle a\rangle^n)\hat{}\langle b\rangle \mid n \in \mathbb{N}\}$—this process can perform as many as as its environment likes, followed by a b, after which there can be no further communication.

Note the use of finite sequence notation here: $\langle a_1, a_2, \ldots, a_n\rangle$ is the sequence containing a_1, a_2 to a_n in that order. Unlike sets, the order of members of a sequence *does* matter, as does the number of times an element is repeated. Thus $\langle a, a, b\rangle$, $\langle a, b\rangle$ and $\langle b, a\rangle$ are all different. $\langle\rangle$ denotes the empty sequence. If s and t are two finite sequences, then $s\hat{}t$ is their *concatenation*: the members of s followed by those of t: for example $\langle a, b\rangle\hat{}\langle b, a\rangle = \langle a, b, b, a\rangle$. If s is a finite sequence and $n \in \mathbb{N}$, then s^n means the n-fold concatenation of s: $s^0 = \langle\rangle$ and $s^{n+1} = (s^n)\hat{}s$. If s is an initial subsequence, or *prefix* of t, in that there is a (possibly empty) sequence w with $t = s\hat{}w$, then we write $s \leq t$. We will meet more sequence notation later when it is required.

For any process P, $\mathit{traces}(P)$ will always have the following properties:

- $\mathit{traces}(P)$ is non-empty: it always contains the empty trace $\langle\rangle$;
- $\mathit{traces}(P)$ is prefix-closed: if $s\hat{}t$ is a trace then at some earlier time during the recording of this, the trace was s.

There are two important things we can do with $\mathit{traces}(P)$: give a meaning, or semantics, to the CSP notation, and specify the behaviour required of processes. The set of all non-empty, prefix-closed subsets of Σ^* is called the *traces model*—the set of all possible representations of processes using traces. It is written $\mathcal{T}$ and is the first—and simplest—of a number of models for CSP processes we will be meeting in this book: the rest are introduced in Chap. 6, Part II and (in a timed context) Chap. 15

Hopefully, given the earlier explanations of what the various constructs of CSP 'meant', the example sets of traces are all obviously correct in the sense that they are the only possible sets of traces that the various processes might have. We can, in fact, *calculate* the trace-set of any CSP process by means of a set of simple rules—for in every case we can work out what the traces of a compound process (such as $a \to P$ or $P \mathbin{\Box} Q$) are in terms of those of its components (P and Q). Thus the traces of any process can be calculated by following its syntactic construction.

The rules for the prefixing and choice constructs are all very easy:

1. $\mathit{traces}(\mathit{STOP}) = \{\langle\rangle\}$

2. $traces(a \to P) = \{\langle\rangle\} \cup \{\langle a\rangle\hat{\ }s \mid s \in traces(P)\}$—this process has either done nothing, or its first event was a followed by a trace of P.
3. $traces(?x : A \to P) = \{\langle\rangle\} \cup \{\langle a\rangle\hat{\ }s \mid a \in A \wedge s \in traces(P[a/x])\}$—this is similar except that the initial event is now chosen from the set A and the subsequent behaviour depends on which is picked: $P[a/x]$ means the substitution of the value a for all free occurrences of the identifier x.
4. $traces(c?x : A \to P) = \{\langle\rangle\} \cup \{\langle c.a\rangle\hat{\ }s \mid a \in A \wedge s \in traces(P[a/x])\}$—the same except for the use of the channel name.
5. $traces(P \,\square\, Q) = traces(P) \cup traces(Q)$—this process offers the traces of P and those of Q.
6. $traces(P \sqcap Q) = traces(P) \cup traces(Q)$—since this process can behave like either P or Q, its traces are those of P and those of Q.
7. $traces(\bigsqcap S) = \bigcup\{traces(P) \mid P \in S\}$ for any non-empty set S of processes.
8. $traces(P \not< b \not> Q) = traces(P)$ if b evaluates to *true*; and $traces(Q)$ if b evaluates to *false*.[5]

The traces of a guarded alternative can be computed by simply re-writing it as an external choice (i.e., replacing all |s by □s).

Notice that the traces semantics of internal and external choice are indistinguishable. What this should suggest to you is that $traces(P)$ does not give a complete description of P, since we certainly want to be able to tell $P \,\square\, Q$ and $P \sqcap Q$ apart. We will see the solution to this problem later, but its existence should not prevent you from realising that knowledge of its traces provides a great deal of information about a process.

This style of calculating the semantics of a process is known as *denotational*. We will see other denotational semantics for CSP in Chap. 6 and study this style in detail in Chaps. 10–12. The final construct we need to deal with is recursion. Think first about a single, non-parameterised, recursion $p = Q$ (or equivalently $\mu p.Q$), where Q is any process expression possibly involving the identifier p. This means the process which behaves like Q when the whole recursively defined object is substituted for the process identifier p in its body: $Q[\mu p.Q/p]$ as in the law $\langle\mu\text{-unwind}\rangle$. The way traces have been calculated through the other constructs means that a term, like Q, with a free process identifier p, represents a function F from sets of traces to sets of traces: if p has set of traces X, then Q has traces $F(X)$. For example, if Q is $a \to p$, $F(X) = \{\langle\rangle\} \cup \{\langle a\rangle\hat{\ }s \mid s \in X\}$. $traces(\mu p.Q)$ should be a set X that solves the equation $X = F(X)$.

Now it turns out that the functions F over $\mathcal{T}$ that can arise from CSP process descriptions always have a least *fixed point* in the sense that $X = F(X)$ and $X \subseteq Y$

[5] *Technical note*: The treatment of identifiers representing input values and process parameters, and appearing in boolean expressions, is very lightweight here. This treatment implicitly assumes that the only terms for which we want to compute $traces(P)$ are those with no free identifiers—so that for example any boolean expression must evaluate to *true* or *false*. The advantage of this approach is that it frees us from the extra notation that would be needed to deal with the more general case, but there is certainly no reason why we could not deal with processes with free identifiers as ‘first class objects’ if desired.

whenever $Y = F(Y)$—this least fixed point always being the appropriate value to pick for the recursion (see Chap. 10). Two separate mathematical theories can be used to demonstrate the existence of these fixed points, namely metric spaces and partial orders. There are introductions to each of these and their most important fixed point methods in Appendix A of TPC. Readers interested in getting full background in this area should study that.

The case of parameterised and other mutual recursions is little different, though the greater generality makes it somewhat harder to formulate. In this case we have a definition for a collection of processes, where the definition of each may invoke any or all of the others. This defines what we might term a *vector* of process names (where, in the case of a parameterised family, the parameter value is part of the name, meaning that there are as many names as there are parameter values) to be equal to a vector of process expressions. The problem of determining the trace-set of one of these mutually defined processes then comes down to solving an equation $\underline{X} = F(\underline{X})$ where $\underline{X}$ is a vector of trace-sets—one for each process name as above—and $F(\cdot)$ is now a function which both takes and delivers a vector of trace-sets. For example, in the mutual recursion

$$\begin{aligned}&P = (a \to P) \,\Box\, (b \to Q)\\&Q = (c \to Q) \,\Box\, (b \to P)\end{aligned}$$

all the vectors have length 2—one component corresponding to each of P and Q. Given a vector $\underline{X} = \langle X_P, X_Q \rangle$, the function F produces a vector, $\langle Y_P, Y_Q \rangle$ say, where

- $Y_P = \{\langle\rangle\} \cup \{\langle a\rangle\hat{}s \mid s \in X_P\} \cup \{\langle b\rangle\hat{}s \mid s \in X_Q\}$ i.e., the result of substituting $\underline{X}$ into the recursive definition of P, and
- $Y_Q = \{\langle\rangle\} \cup \{\langle c\rangle\hat{}s \mid s \in X_Q\} \cup \{\langle b\rangle\hat{}s \mid s \in X_P\}$ i.e., the result of substituting $\underline{X}$ into the recursive definition of Q.

In the case of *COUNT*, the vectors would be infinite—with one component for each natural number. B^∞ will also produce infinite vectors, but this time there is one component for each finite sequence of the type being transmitted.

The extraction of fixed points is mathematically the same whether the functions are on single trace-sets or on vectors. The only difference is that the intended process value in the case of a mutual recursion will be one of the components of the fixed point vector, rather than the fixed point itself.

All of the recursions we have seen to date (and almost all recursions one meets in practice) have a property that makes them easier to understand—and reason about. They are *guarded*, meaning that each recursive call comes after (i.e., is prefixed by) a communication that is introduced by the recursive definition[6] rather than being exposed immediately. Examples of *non*-guarded recursions are $\mu p.p$ (perhaps the archetypal one), $\mu p.p \,\Box\, (a \to p)$, and the parameterised mutual recursion (over the

[6] This definition will be modified later to take account of language constructs we have not met yet.

natural numbers)[7]

$$P(n) = (a \to P(1)) \lhd n = 1 \rhd P((3n+1)\,\mathsf{div}\,2 \lhd n\ \mathsf{odd} \rhd n\,\mathsf{div}\,2)$$

The point about a guarded recursion is that the first-step behaviour does not depend at all on a recursive call, and when a recursive call is reached, the first step of its behaviour, in turn, can be computed without any deeper calls, and so on. In other words, we are guaranteed to have communicated at least n events before a recursive call is made at depth n.

2.2.2 Traces and Laws

In Sect. 2.1 we introduced the notion of equality between processes provable by a series of laws. One can have two quite different processes, textually, which are provably equal by a series of these laws. Whatever the text of a process, the previous section gives us a recipe for computing its set of traces. We should realise that these two theories have to be squared with each other, since it would be a ridiculous situation if there were two processes, provably equal in the algebra, that turned out to have different trace-sets.

Of course this is not so, since all the laws quoted are easily shown to be valid in the sense that the traces of the processes on the left- and right-hand sides are always the same. For example, since the traces of $P \Box Q$ and $P \sqcap Q$ are both given by union, the trace-validity of their idempotence, symmetry and associative laws follow directly from the same properties of set-theoretic union ($\cup$). Since $\sqcap$ and $\Box$ are indistinguishable from each other in traces, their distributive laws over each other are equally simple, for example

$$\begin{aligned}
traces(P \sqcap (Q \Box R)) &= traces(P) \cup (traces(Q) \cup traces(R)) \\
&= (traces(P) \cup traces(Q)) \\
&\quad \cup (traces(P) \cup traces(R)) \\
&= traces((P \sqcap Q) \Box (P \sqcap R))
\end{aligned}$$

On the other hand, since there are distinctions we wish to make between processes that we know are not made by traces, we would expect that there are processes P and Q such that $traces(P) = traces(Q)$ (which we can abbreviate $P =_T Q$) but such that $P = Q$ is *not* provable using the laws. This is indeed so, as our investigations of more refined models in later chapters will show.

Clearly the validity of the various laws with respect to traces means we can prove the equality of $traces(P)$ and $traces(Q)$ by transforming P to Q by a series of

[7] The interesting thing about this particular example is that it is not known whether or not the series (of parameters) generated by an arbitrary starting value will always reach 1, so in fact we *do not know* whether all the components of this recursion will always be able to communicate an a. Of course not nearly this amount of subtlety is required to give unguarded mutual recursions!

laws. The rather limited set of operators we have seen to date means that the range of interesting examples of this phenomenon we can discuss yet is rather limited. However, there is one further proof rule which greatly extends what is possible: the principle of unique fixed points for guarded recursions.

2.2.3 Unique Fixed Points

If $Z = F(Z)$ is the fixed-point equation generated by any guarded recursion (single, parameterised or mutual) for trace-sets and Y is a process (or vector of processes) whose trace-sets satisfy this equation, then $X =_T Y$ where X is the process (or vector) defined by the recursion. In other words, the equation has precisely one solution over[8] $\mathcal{T}$ or the appropriate space of vectors over $\mathcal{T}$. This rule is often abbreviated UFP; its theoretical justification can be found in Sect. 10.2, but in essence it is true because guardedness means that we can completely determine the traces of the fixed point P of a guarded function F, with length n or less, from the equation $P = F^n(P)$.

Recall for example, the first two recursive processes we defined:

$$P_1 = up \to down \to P_1$$

$$P_2 = up \to down \to up \to down \to P_2$$

We know, by unwinding the first of these definitions twice, that

$$P_1 = up \to down \to up \to down \to P_1 \qquad (\dagger)$$

Thus P_1 satisfies the equation defining P_2. Since P_2 is guarded we can deduce that $P_1 =_T P_2$—in other words, $traces(P_1)$ solves an equation with only one solution, namely $traces(P_2)$. Of course it was obvious that these two processes are equivalent, but it is nice to be able to prove this!

In applying this rule in future we will not usually explicitly extract the trace-sets of the process we are claiming is a fixed point. Instead, we will just apply laws to demonstrate, as in (†) above, that the syntactic process solves the recursive definition.

Most interesting examples of the UFP rule seem to derive from mutual recursions, where we set up a vector $\underline{Y}$ that satisfies some mutual recursion $\underline{X} = F(\underline{X})$. Indeed, the mutual recursion is usually in the form of a one-step tail recursion (precisely one event before each recursive call). The thing to concentrate on is how these vectors $\underline{Y}$ are constructed to model the state spaces that these tail recursions so clearly describe.

[8] IMPORTANT: though the UFP rule is stated here in terms of the traces model $\mathcal{T}$, because this is the only model we have seen so far, it applies equally to all models of CSP to be found in this book except for some introduced in Chap. 12.

As an easy but otherwise typical example, suppose our *COUNT* processes were extended so that the parameter now ranges over *all* the integers $\mathbb{Z}$ rather than just the non-negative ones $\mathbb{N}$:

$$ZCOUNT_n = up \rightarrow ZCOUNT_{n+1} \,\Box\, down \rightarrow ZCOUNT_{n-1}$$

The striking thing about this example, when you think about it, is that the value of the parameter n actually has no effect at all on the behaviour of $ZCOUNT_n$: whatever its value, this process can communicate any sequence of *up*s and *down*s. This might lead us to believe it was equal to the process

$$AROUND = up \rightarrow AROUND \,\Box\, down \rightarrow AROUND$$

and indeed we can use the UFP rule to prove $AROUND =_T ZCOUNT_n$ for all n. Let $\underline{A}$ be the vector of processes with structure matching the *ZCOUNT* recursion (i.e., it has one component for each $n \in \mathbb{Z}$) where every component equals *AROUND*. This is a natural choice since we conjecture that every $ZCOUNT_n$ equals *AROUND*. Applying the function F_{ZC} of the *ZCOUNT* recursion to this vector we get another, whose nth component is

$$\begin{aligned} F_{ZC}(\underline{A})_n &= up \rightarrow A_{n+1} \,\Box\, down \rightarrow A_{n-1} \\ &= up \rightarrow AROUND \,\Box\, down \rightarrow AROUND \\ &= AROUND \\ &= A_n \end{aligned}$$

(where the second line follows by definition of $\underline{A}$ and the third by definition of *AROUND*). Thus $\underline{A}$ is indeed a fixed point of F_{ZC}, proving our little result.

The basic principle at work here is that, in order to prove that some process P (in this case *AROUND*) is equivalent to a component of the tail recursion $\underline{X} = F(\underline{X})$ (in this case $\underline{ZCOUNT}$), you should work out what states P goes through as it evolves. Assuming it is possible to do so, you should then form a hypothesis about which of these states each component of $\underline{X}$ matches up with. In our case there is only one state of P, and *all* the components of *ZCOUNT* match up with it. You then form the vector $\underline{Y}$ by replacing each component of $\underline{X}$ by the state of P conjectured to be equivalent, and then try to prove that this creates a solution to the tail recursion: if you can do this, you have completed the proof.

Both in the text and the exercises, there will be a number of examples following basically this argument through the rest of Part I (see, for example, pp. 52, 101 and 133, and Exercises 3.9 and 7.3).

Exercise 2.7 Prove the validity in traces of the laws ⟨prefix-dist⟩ (2.9) and ⟨□-step⟩ (2.14).

Exercise 2.8 Recall the processes P_1, and P_u and P_d from Sect. 1.1.2. Prove that $P_u =_T P_1$ by the method above. [*Hint*: *show that a vector consisting of* P_1 *and one other process is a fixed point of the* $\langle P_u, P_d\rangle$ *recursion*.]

Exercise 2.9 Use laws and the UFP rule to prove that

$$Chaos_A \sqcap RUN_A =_T Chaos_A$$

for any alphabet A.

2.2.4 Specification and Refinement

Traces are not just a dry and abstract model of processes to help us decide equality, but give a very usable language in *specification*. A specification is some condition that we wish a given process to satisfy. Since a CSP process is, by assumption, characterised completely by its communicating behaviour, it is obviously the case that we will be able to formulate many specifications in terms of $traces(P)$. In fact, most trace specifications one meets in practice are what we term *behavioural* specifications: the stipulation that each $s \in traces(P)$ meets some condition $R(s)$. This is termed a behavioural specification because what we are doing is 'lifting' the specification R on the individual recorded behaviours (i.e., traces) to the whole process.

There are two different approaches to behavioural specifications and their verification. The first (which is that adopted in Hoare's book, where you can find many more details than here) is to leave R explicitly as a specification of traces (generally using the special identifier *tr* to range over arbitrary traces of P). The second, and essentially equivalent, method, and the one required for FDR, is to define a process *Spec* that has just those traces that can occur in a process satisfying the specification, and test $Spec \sqsubseteq_T P$. There are merits in both of these ways of expressing specifications, as well as others such as *temporal logics*. The functionality of FDR means, however, that the main emphasis of this book will be on processes-as-specifications.

In Hoare's notation

$$P\,\mathbf{sat}\,R(tr) \quad \text{means} \quad \forall tr \in traces(P).R(tr)$$

This is meaningful however R is constructed, though usually it is expressed in predicate logic using trace notation.

In order to be able to express this sort of property it is useful to extend our range of trace notation:

- If s is a finite sequence, $\#s$ denotes the *length* of s (i.e., the number of members).
- If $s \in \Sigma^*$ and $A \subseteq \Sigma$ then $s \upharpoonright A$ means the sequence s *restricted* to A: the sequence whose members are those of s which are in A. $\langle\rangle \upharpoonright A = \langle\rangle$ and $(s\hat{}\langle a\rangle) \upharpoonright A = (s \upharpoonright A)\hat{}\langle a\rangle$ if $a \in A$, $s \upharpoonright A$ otherwise.
- If $s \in \Sigma^*$ then $s \downarrow c$ can mean two things depending on what c is. If c is an *event* in Σ then it means the number of times c appears in s (i.e., $\#(s \upharpoonright \{c\})$), while if c is a *channel name* (associated with a non-trivial data-type) it means the sequence of values (without the label c) that have been communicated along c in s. For example,

$$\langle c.1, d.1, c.2, c.3, e.4\rangle \downarrow c = \langle 1, 2, 3\rangle$$

The following are some examples of specifications describing features of some of the processes we have already met.

- Various processes in Sect. 1.1.2 all satisfy the condition:

  $$tr \downarrow down \leq tr \downarrow up \leq tr \downarrow down + 1 \qquad (\ddagger)$$

  which states that they have never communicated more *down*s than *up*s, and neither do they fall more than one behind.
- The specification of $COUNT_n$ is similar but less restrictive:

  $$tr \downarrow down \leq tr \downarrow up + n$$

- $B^{\infty}_{\langle\rangle}$ and *COPY* both satisfy the basic *buffer* specification:

  $$tr \downarrow right \leq tr \downarrow left$$

  (noting that here $\leq$ means prefix and the things to its left and right are sequences of values). This is in fact the strongest trace specification that $B^{\infty}_{\langle\rangle}$ meets, but *COPY* meets further ones.

Hoare gives a set of proof rules for establishing facts of the form $P\,\mathbf{sat}\,R(tr)$—essentially a re-coding into logic of the rules we have already seen for computing $traces(P)$. The following rules cover the operators we have seen to date (bearing in mind the known equivalences between forms of prefixing).

$$STOP\,\mathbf{sat}(tr = \langle\rangle)$$

$$\frac{\forall a \in A.P(a)\,\mathbf{sat}\,R_a(tr)}{?a : A \to P\,\mathbf{sat}(tr = \langle\rangle \vee \exists a \in A.\exists tr'.\, tr = \langle a\rangle\hat{\ }tr' \wedge R_a(tr'))}$$

$$\frac{P\,\mathbf{sat}\,R(tr) \wedge Q\,\mathbf{sat}\,R(tr)}{P \,\Box\, Q\,\mathbf{sat}\,R(tr)}$$

$$\frac{P\,\mathbf{sat}\,R(tr) \wedge Q\,\mathbf{sat}\,R(tr)}{P \sqcap Q\,\mathbf{sat}\,R(tr)}$$

$$\frac{P\,\mathbf{sat}\,R(tr) \wedge \forall tr.R(tr) \Rightarrow R'(tr)}{P\,\mathbf{sat}\,R'(tr)}$$

$$\frac{P\,\mathbf{sat}\,R(tr) \wedge P\,\mathbf{sat}\,R'(tr)}{P\,\mathbf{sat}\,R(tr) \wedge R'(tr)}$$

The most interesting is that relating to recursion, and in fact Hoare's rule can usefully (and validly) be generalised in two ways: his assumption that the recursion is guarded is not necessary for this style of proof (though it is in many similar proof rules, some of which can be found later in this book or in TPC), and we can give a version for mutual recursion by attaching one proposed specification to each component of the vector of processes being defined.

PROOF RULE FOR RECURSION *Suppose* $\underline{X} = F(\underline{X})$ *is the fixed point equation for* (*vectors of*) *trace-sets resulting from some recursive definition, and that* $\underline{X}$ *is the* (*least*) *fixed point which it defines. Let* Λ *be the indexing set of the vectors, so that* $\underline{X} = \langle X_\lambda \mid \lambda \in \Lambda \rangle$. *Suppose that for each* λ *there is a specification* R_λ *such that*

- $\mathit{STOP}\,\mathbf{sat}\,R_\lambda(tr)$ *for all* $\lambda \in \Lambda$, *and*
- $\forall \lambda \in \Lambda . Y_\lambda\,\mathbf{sat}\,R_\lambda(tr) \Rightarrow \forall \lambda \in \Lambda . F(\underline{Y})_\lambda\,\mathbf{sat}\,R_\lambda(tr)$

then $X_\lambda\,\mathbf{sat}\,R_\lambda(tr)$ *for all* $\lambda \in \Lambda$.

Paraphrasing this: we attach a specification R_λ to each component of the mutual recursion, and provided all of these are satisfied by *STOP* and, on the assumption that they all hold of recursive calls, they hold of the body of the recursion, then we can infer they hold of the actual process(es) defined by the recursion. This rule is formally justified in Sect. 9.2 of TPC, which discusses a number of related proof rules collectively known as *recursion induction*.

The above can be used to prove that the *COUNT* processes meet the vector of specifications quoted for them above and, provided one can come up with appropriate specifications for the B_s^∞ processes for $s \neq \langle\rangle$, one can prove that $B_{\langle\rangle}^\infty$ meets its specification.

The most curious feature of this is the role played by *STOP*. It does not seem a very useful process and yet its satisfying R is a precondition to the above rule (and Hoare's). At first sight it seems unlikely that many useful specifications will be met by *STOP*, but in fact *any* behavioural trace specification which is satisfied by any process at all is satisfied by *STOP*. For $\mathit{traces}(\mathit{STOP}) = \{\langle\rangle\} \subseteq \mathit{traces}(P)$ for any P, and so if all the traces of P satisfy R, so do all those of *STOP*.

This shows precisely the limitation of trace specifications: while they can say that a process P cannot do anything stupid, they cannot force it to do anything at all. For this reason they are often termed *safety* or *partial correctness* conditions, while *liveness* or *total correctness* conditions are ones that additionally force a process to be able to do things. In later chapters we will develop models that allow us to build liveness specifications.

In order to satisfy '$\mathbf{sat}\,R(tr)$' a process's traces must be a *subset* of the traces which R allows. In fact, most of the example specifications given above have the property that the target process has the *largest possible* set of traces of any process satisfying it. This can be expressed in several different, but equivalent, ways (where P is the process and R the trace condition):

- $P =_T \sqcap\{Q \mid Q\,\mathbf{sat}\,R(tr)\}$ or, in other words, P is trace equivalent to the nondeterministic choice over all processes meeting the specification.
- $Q\,\mathbf{sat}\,R(tr) \Rightarrow \mathit{traces}(Q) \subseteq \mathit{traces}(P)$
- $\mathit{traces}(P) \subseteq \{s \mid \forall t \leq s . R(t)\}$, the largest prefix-closed set of traces satisfying R. (It is worth noting that the set of traces satisfying each of the trace specifications on p. 37 is *not* prefix-closed. For example, the trace $\langle \mathit{down}, \mathit{up}\rangle$ satisfies the specification (‡) there, but since the prefix $\langle \mathit{down}\rangle$ does not, the longer trace is not possible for a process *all* of whose traces satisfy (‡).)

Remember we defined that Q *refines* P, written $P \sqsubseteq Q$ if $P = Q \sqcap P$. Interpreted over the traces model, this leads to the concept of *traces refinement*

$$P \sqsubseteq_T Q \;\equiv\; P =_T Q \sqcap P \;\equiv\; traces(Q) \subseteq traces(P)$$

The above properties demonstrate that, for any satisfiable behavioural trace specification R there is always a process P_R (given by the formula in the first bullet point, and whose traces are the expression in the third) that is the most nondeterministic satisfying R and such that

$$Q\,\mathbf{sat}\,R(tr) \Leftrightarrow P_R \sqsubseteq_T Q$$

Let us say that P_R is the *characteristic* process of R. In other words, satisfaction (**sat**) can always be determined by deciding refinement against a suitably chosen process.

For example, $B^{\infty}_{\langle\rangle}$ is the (characteristic process of the) trace specification of a buffer, and a process will trace-refine it if, and only if, it meets the trace-based buffer specification. Thus $COPY \sqsupseteq_T B^{\infty}_{\langle\rangle}$, and all but the last of your answers to Exercise 1.7 should have the same property. (Here, we are taking the liberty of writing $P \sqsupseteq_T Q$ as the equivalent of $Q \sqsubseteq_T P$. We will do this for all order relations in future without comment, as the need arises.)

There are some major advantages in identifying each specification with the most nondeterministic process satisfying it.

- This is the form in which FDR codes specifications and allows them to be mechanically verified or refuted.
- Refinement has many properties that can be exploited, for example it is *transitive*:

  $$P \sqsubseteq Q \wedge Q \sqsubseteq T \Rightarrow P \sqsubseteq T$$

  and *monotone*: if $C[\cdot]$ is any process context, namely a process definition with a slot to put a process in, then

  $$P \sqsubseteq Q \Rightarrow C[P] \sqsubseteq C[Q]$$

  If $C[Q]$ is a process definition with component Q, with an overall target specification S, we might be able to factor the proof of $S \sqsubseteq C[Q]$ into two parts. First, find a specification P such that $S \sqsubseteq C[P]$. Second, prove $P \sqsubseteq Q$, which implies thanks to monotonicity that $C[P] \sqsubseteq C[Q]$. Transitivity then gives $S \sqsubseteq C[Q]$. This software engineering technique is known as *compositional development*.

  Note how the identification of processes and specifications allowed us to consider the object $C[P]$, which we might read as '$C[Q]$, on the assumption that the process Q satisfies the specification P'.
- It allows one to move gradually from specification to implementation, using the transitivity property quoted above, creating a series of processes

  $$Spec \sqsubseteq P_1 \sqsubseteq \cdots \sqsubseteq P_n \sqsubseteq Impl$$

where the first is the specification, and each is created by refining the previous one till an acceptable implementation is reached. This is known as *stepwise refinement*.

It is worth noting that, since the refinement $P \sqsubseteq Q$ is expressible as the equality $P \sqcap Q = P$, it makes sense to try to prove it algebraically. Recall Exercise 2.9.

Of course, the limitations of trace specification discussed earlier still apply here. It is worth noting that $\mathit{STOP} \sqsupseteq_T P$ and $\mathit{RUN} \sqsubseteq_T P$ for all processes P.

Our emphasis on refinement-based proof means that we will not give any of the **sat** rules for the further operators and models we introduce later in this book; the interested reader can, of course, find many of them in Hoare's text.

2.2.5 Afters and Initials

If P is any process, $\mathit{initials}(P)$ (abbreviated P^0 in some publications on CSP) is the set of all its initial events

$$\mathit{initials}(P) = \{a \mid \langle a\rangle \in \mathit{traces}(P)\}$$

This set is often used in specifications and other definitions.

For example, $\mathit{initials}(\mathit{STOP}) = \emptyset$ and $\mathit{initials}(?x : A \rightarrow P(x)) = A$.

If $s \in \mathit{traces}(P)$ then P/s (pronounced 'P *after* s') represents the behaviour of P after the trace s is complete. Over the traces model, P/s can be computed

$$\mathit{traces}(P/s) = \{t \mid s\hat{\ }t \in \mathit{traces}(P)\}$$

This operator should not be thought of as an ordinary part of the CSP language, rather as a notation for discussing behaviour of processes in fairly abstract contexts, to represent the behaviour of P on the *assumption* that s has occurred. The best reason for not including it as an operator you could use in programs is that it is not implementable in a conventional sense: the process

$$(\mathit{STOP} \sqcap a \rightarrow a \rightarrow \mathit{STOP})/\langle a\rangle$$

is equivalent to $a \rightarrow \mathit{STOP}$, but no reasonable implementation acting on the nondeterministic choice here can force it to do anything.

Over the traces model it is true that

$$P \;=_T\; ?x : \mathit{initials}(P) \rightarrow P/\langle x\rangle$$

but we will find that this is not true over more discriminating models.

Exercise 2.10

(a) Let $N \geq 0$. Give a trace specification for a process with events a, b and c which states that the absolute value of the difference between the number of as and the total number of bs and cs is at most N.

(b) Now find a CSP definition of a process D_N for which this is the strongest specification. [*Hint*: *give a parameterised recursion whose parameter is the present difference*.] D_0 is equivalent to a well-known simple process: what is it and why?

(c) What traces refinements hold between the D_N?

Exercise 2.11 Give the strongest trace specification satisfied by $\mathit{COPY} = \mathit{left}?x \rightarrow \mathit{right}!x \rightarrow \mathit{COPY}$. Use the proof rules for **sat** given above to prove that *COPY* meets it.

Exercise 2.12 See Exercise 1.6. Give a trace specification that a machine with events $\{\mathit{in}£1, \mathit{out}5p, \mathit{out}10p, \mathit{out}20p\}$ has never given out more money than it has received.

2.3 Operational Semantics and Labelled Transition Systems

Unlike algebraic laws and behavioural models, an *operational* semantics does not attempt to capture some sort of extensional[9] equivalence between processes. Rather it gives a formalisation of how a process can be implemented. We will see several styles of operational semantics for CSP in this book. The most practically important of these are based on the idea of *labelled transition systems*, usually abbreviated LTS.

A labelled transition system consists of a non-empty set of states S, with a designated initial state P_0, a set of labels L (which for us will always be the set of visible actions processes can perform plus a special, invisible action τ, the Greek letter tau), and a ternary relation $P \xrightarrow{x} Q$ meaning that the state P can perform an action labelled x and move to state Q. All the transition pictures we showed in Chap. 1 are examples of LTSs, for example Figs. 1.1 and 1.2. A further one, the behaviour of the process *Asleep* (p. 6) is shown in Fig. 2.1.

Many operational semantics consist of rules for extracting an LTS from the syntax of a program: from any program state P they provide rules for calculating what

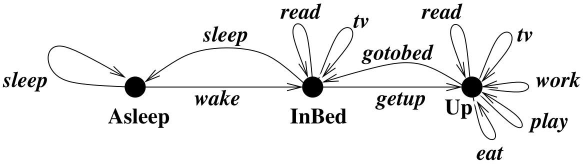


Fig. 2.1 Example LTS: the process *Asleep*

[9] Here, *extensional* means equality between objects based on the properties that can be observed of them. Thus the *axiom of extension* in set theory says that two sets are equal if and only if they have the same members. The opposite of extensional is *intensionality*, and intensional semantics capture not only what a program does but some element of how it does it. Thus operational semantics have at least an element of intensionality about them.

the initial labels are of actions that P can perform, together with the state the process moves to resulting from each action. In CSP these program states are usually just pieces of CSP syntax. A given state may have more than one result for a given label: for example the operational semantics of $P \sqcap Q$ has two different results of τ (namely P and Q) and $(a \to P) \Box (a \to Q)$ has the same two results for a. We will see in Chap. 9 that the rules for reducing a process description to an LTS can take a number of different forms.

FDR operates on the LTS representation of a process, which it calculates from the syntax. When we refer to a *state* or number of states in the context of an FDR check, we are always referring to states in the resulting LTS. While the nodes of an LTS are often marked with a representation of the processes they represent, these node markings are irrelevant: the behaviour comes only from the actions.

It is clear that we can calculate[10] the traces of a process from its LTS semantics: just start from the initial state and see what sequences of actions are possible from there, deleting the τs. Everything that FDR does with traces is based on this correspondence: the tool *never* calculates an explicit trace-set for any process.

We will not give the explicit rules of an operational semantics here: by and large you can rely on FDR calculating a process's transitions either to use itself or to display the transitions in graph mode. For most simple sequential processes (such as those whose transition systems we have already seen), the shape of the LTS is fortunately obvious: indeed it would make a lot of sense to have a graphical input mode for FDR and other tools that would allow the user to draw an LTS and automatically generate the corresponding CSP (we will later have an exercise (5.12) to show that every LTS can be implemented as a CSP process.

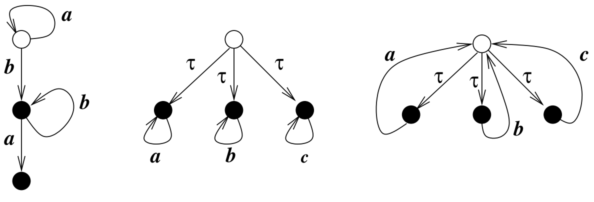
It is important for using FDR that you know what an LTS is and what our assumptions are about how they behave:

- Starting from the initial state, a process is always in one of the states of its LTS.
- When a process is in a state that has visible (outward-pointing) actions, it may allow the environment to synchronise with one of these.
- If it has no τ actions, it must agree to one of its visible actions if offered by the environment.
- If it does have one or more τ actions, then either it synchronises quickly on a visible action with the environment, or follows one of its τ options. Following a τ option is both unobservable and uncontrollable by the environment.

Two processes can easily be equivalent in every behavioural model and yet have different transition systems. This is clear from the two examples in Fig. 1.1. As another example, every process P is behaviourally equivalent to the one in which the initial node leads directly, by a τ action to the initial node of P. (Following some other process algebras it is convenient to write this process as τP.)

We will find in later chapters (particularly Chap. 9) that there are theories that allow us (and FDR) to compare and manipulate LTSs. It is not necessary to understand these now.

[10] It should also be clear to those familiar with automata theory that the trace set of any finite-state process is a *regular language*.



(i) (ii) (iii)

Fig. 2.2 See Exercise 2.13

Exercise 2.13 Give CSP definitions of processes whose natural LTS implementations are the diagrams shown in Fig. 2.2. In each case the initial node is indicated by the white circle.

2.4 Tools

As stated above, FDR works by calculating the LTS semantics of the specification and implementation processes of each refinement check `Spec T=[Imp` and then running an algorithm on these that produces the same answer we would have got by working out the traces of both sides and comparing them directly. You can find some details of this algorithm in Chap. 8. One obvious advantage of this approach is that it works for any finite-state system, even when they have infinite trace sets.

There is, unfortunately, no way of checking the various laws of CSP on FDR in their full generality. All we can do is check particular cases of the laws when all the processes and other parameters have been substituted by concrete values. This demonstrates the main weakness of the style of reasoning that FDR represents: excellent at concrete examples, but much more difficult to use for proving general results.

A number of techniques where, by doing a few carefully chosen FDR checks, one can establish general results, can be found in Chap. 17. None of these applies to the laws. Until recently the only option was to prove them by hand. Several groups, in particular that of Markus Roggenbach at Swansea [115, 116], and [156], have shown how to embed the theories of several CSP models in theorem proving tools such as Isabelle [103], and then prove both the laws in general and other coherency properties of the definitions of CSP operators over the models. This type of tool is not recommended for CSP beginners unless they happen to be experts in theorem proving technology.

http://www.springer.com/978-1-84882-257-3

Understanding Concurrent Systems
Roscoe, A.W.
2010, XVIII, 530 p., Hardcover
ISBN: 978-1-84882-257-3