CMU-CS-83-140

A MODEL FOR

COMMUNICATING SEQUENTIAL PROCESSES

Stephen D. Brookes

Thesis submitted for the degree of Doctor of Philosophy.
University College
Oxford University
January 1983

ABSTRACT

This thesis describes the construction and mathematical properties of a model for communicating sequential processes. We define a semantic model for processes based on *failures,* which encapsulate certain finite aspects of process behaviour and allow an elegant treatment of nondeterminism. We define a set of process operations, including nondeterministic choice, conditional composition, and various forms of parallel composition. These process operations enjoy many interesting mathematical properties, which allow us to prove many process identities. The failures model is well suited to reasoning about deadlock properties of processes, and some examples are given to illustrate this.

The failures model does not treat in a reasonable way the phenomenon of divergence, which occurs when a process performs a potentially infinite sequence of internal actions and never interacts with its environment. We offer an extension of the model in Chapter 5, which treats divergence more satisfactorily. We also give a complete proof system for proving semantic equivalence of terms.

The thesis also contains some results on the relationship of these models to other models of processes in the literature, specifically relating to the work of Milner, Kennaway, Hennessy and de Nicola. Chapter 3 gives an alternative formulation of the failures model, allowing comparison with Kennaway's work. Chapter 4 uses Milner's synchronisation trees as the basis for representing processes. In Chapter 6 we show how links can be established between Kennaway's model, the work of Hennessy and de Nicola, and our failures model; this chapter focusses on modal assertions of various kinds about process behaviour. We show that by varying the class of assertions we can characterise different semantic models.

Contents

Introduction . . . . . Page 1

Acknowledgements 5

Chapter 1: A Domain of Processes 6

Chapter 2: Process operations 18

Chapter 3: Implementations 75

Chapter 4: Relationship with Milner's CCS 90

Chapter 5: A proof system for CSP 108

Chapter 6: Testing processes 129

Chapter 7: Some examples 153

Chapter 8: Operational semantics 165

Conclusions 173

Appendix A: A technical lemma 176

References 178

Introduction

The decreasing cost and diminishing size of powerful computers, and the opportunities offered in speed and efficiency by distributed computer systems are encouraging the use of programming languages in which the programmer can make use of *concurrency*. By allowing a program's task to be distributed when it is not logically necessary for a single processor to perform it, significant gains can be made in efficiency. A job shared by several concurrent processors can be a job halved in execution time. Accordingly, many new programming languages have been proposed over the last few years, each involving a form of *parallel* construct.

One of the most widely known of these languages is C.A.R.Hoare's CSP [H1], first published in 1978. In this language input and output, i.e., communication between concurrently active processes, was taken as a primitive, much in the way that conventional sequential imperative languages take assignment and sequencing as primitives. The most important feature of CSP was that it emphasised programmer-specified communication between processes and used *synchronization* as the mechanism for communication. The language, based on Dijkstra's guarded commands, allowed input requests to be used in guards, so that communications could direct the process's behaviour in a very elegant way. Moreover, a process in CSP can exhibit nondeterministic behaviour, notably when attempting to perform a guarded command in which more than one guard is satisfied. Depending on which guard is chosen in such circumstances the future behaviour of the process and its communication capabilities with other concurrent processes may vary. This was all evident in Hoare's paper. Nondeterminacy is often a feature in parallel execution of programs, because one usually wishes to abstract away from the relative speeds of concurrently executing processors.

It is well known that programs written in parallel programming languages can sometimes exhibit pathological behaviour. One of the most important examples of undesirable behaviour is called *deadlock*. A system of processes is said to be deadlocked if none of the processes in the system (assumed to be

operating in parallel) is capable of communicating with any other: nothing can happen, typically because no process is able to make a communication until another has done so. This situation can arise in practice when many processes are competing to share some scarce resource. When writing a program it is often vital to guarantee freedom from deadlock.

When using any programming language it is important to know what the syntactic constructs are intended to *denote*. One cannot understand a program properly and be able to reason effectively about its execution unless one has some sound ideas about the *semantics* of the language. For conventional sequential programming languages such as PASCAL, and the ALGOL family, there is a well-established body of work dealing with semantics. In general, programs here can be taken to denote *input-output* functions or *state transformations,* and logical proof systems can be built using Hoare-style assertions or Dijkstra's weakest preconditions. Partial and total correctness arguments or proofs can be given for programs in such languages in a reasonably straightforward way.

In the case of a parallel programming language the situation is not nearly so satisfactory. There is, for a start, no general agreement on what class of mathematical entities is suitable for modelling the meaning of programs written in a concurrent language. Even for one language, it is conceivable that more than one reasonable model exists. As remarked above, parallel programming can lead to pathological behaviour, of a kind not occurring with sequential programs. It is important to know to what extent any model for parallel processes satisfactorily captures behavioural properties, such as deadlock. Different semantic models often seem to focus on particular classes of properties. For example, an early semantics for CSP, the so-called *trace semantics* of Hoare, was ideally suited to reasoning about potential communication sequences but was insensitive to the possibility of deadlock.

This thesis is primarily concerned with the construction of a mathematical model of communicating processes and a collection of process operations. The model enjoys a number of interesting and elegant mathematical properties, and seems adequate to serve as a semantic model for languages such as CSP. We investigate in some detail the properties of our model, proving many process identities and other relations between processes; these results enable us to specify and prove a class of semantic properties of individual processes. Roscoe's thesis [R] contains a detailed account of some general proof techniques applicable to this model; Roscoe has made significant contributions during the development of the model.

In Chapter 1 we begin with the definition of a semantic domain for processes over an alphabet Σ. The elements of Σ, termed *events* can be thought

of as communications or other atomic actions. Instead of representing a CSP process as a set of traces, as in the trace model, we use a set of *failures*. A failure is a generalisation of a trace to include an indication of the potential for deadlock at the next step in execution. There is a natural ordering on failure sets corresponding to the fact that the nondeterminacy of a process is mirrored in its failure set: if one process is more nondeterministic than another, then its failure set should be larger. This order turns the set of processes (identified with failure sets) into a complete semi-lattice.

In Chapter 2 we give a denotational semantics to an abstract version of CSP, using the semantic domain just described. As usual in denotational semantics, the meaning of a combination of processes is defined from the meanings of the component processes. In effect, we define a set of operations on processes, each corresponding to a syntactic operation on terms in an abstract language derived from CSP. With one notable exception, these process operations are continuous with respect to the nondeterminism ordering.

Chapter 3 contains an alternative construction of the domain of processes, in which a nondeterministic process is modelled as a set of deterministic processes. The basic idea is to identify a process with its set of possible *implementations,* each implementation being a deterministic process to which the original process is an approximation. All of the process operations of Chapter 2 can be defined in this new formulation. They all possess an intuitively appealing property termed *implementability*. This work points the way to some later remarks about the relationship between our model for CSP and Kennaway's model [K1,2], which also modelled nondeterminism with sets of deterministic processes.

In Chapter 4 we introduce for comparison the semantic model used by Milner for his language CCS (Calculus of Communicating Systems) [M1]. This language, also widely used in theoretical investigations, involves a different form of parallel composition from that used in CSP. Milner's semantics, based on *synchronisation trees* and an *observational equivalence* relation, is able to express more subtle behavioural properties than the failures model. We define a failure equivalence relation on synchronisation trees and show how it can be axiomatized in a similar way to Milner's equivalence. We also show how the CSP operations of Chapter 2 can be defined on synchronisation trees so that they preserve failure equivalence: thus we obtain a faithful semantic representation of a process as a synchronisation tree.

Chapter 5 extends the failures model to allow a more pleasing treatment of the phenomenon of *divergence*. This leads to a semantics for processes based on failure sets and divergence sets, from which we can recover the earlier

failure semantics by ignoring divergence. The main results of this chapter concern a complete and consistent proof system for semantic equivalence of terms. Again, by restricting attention to divergence-free terms we obtain a complete proof system for the failure semantics.

Chapter 6 discusses in some detail the connections between our model and the work of Hennessy and de Nicola [HN], as well as Kennaway. It appears that, by beginning with synchronisation trees and certain types of modal assertions which, when interpreted, state *necessary* or *possible* attributes of process executions, we can obtain various different models: notably, our model and Kennaway's. This work shows very elegantly the subtle differences in the characterizations of these models: by varying the class of assertions and identifying processes which satisfy precisely the same set of assertions we obtain some interesting results.

By now we will have listed an enormous number of process properties and proof rules. Chapter 7 is an attempt to consolidate a little, by illustrating a treatment of some classical concurrent programming problems in our framework. We discuss briefly the Dining Philosophers problem of Dijkstra, the well known mutual exclusion problem, and give some examples involving networks of processes which communicate along named channels.

In Chapter 8, we indicate briefly how an operational semantics for our language can be given, following the general lines of Plotkin [P1,2] and the particular examples of Hennessy and Milner [HM,HP].

The final section contains some conclusions, and suggestions for future research.

Acknowledgements

The author would like to thank Professor C.A.R.Hoare, who has been a continuing source of inspiration and encouragement during the development of this work. Many thanks also to Bill Roscoe, with whom I have had many fruitful discussions and collaborative results. The other main stimulus to this work has been the research of Robin Milner and his colleagues at Edinburgh University. I have benefitted from discussions with Matthew Hennessy, Robin Milner, David Park, Bill Rounds, Joe Stoy, and Zhou Chaochen.

This research was supported by a grant from the Science and Engineering Research Council of Great Britain. The author is also grateful to the Computer Science Department of Carnegie-Mellon University, whose facilities were used in the preparation of this thesis. Thanks to William L. Scherlis for help with the type-setting of the manuscript.

Chapter 1

A Domain of Processes

0. Introduction

This chapter introduces a mathematical model of communicating sequential processes. This model will be used throughout the thesis as the basis for a semantics for an abstract version of CSP. In the model we abstract from the details of inter-process communication by assuming that we may adequately represent behaviour using the primitive notion of *event*. Moreover, although we will use the model exclusively for CSP the work in this chapter uses the term *process* in an abstract sense, without specifying any intended interpretation of processes and events. Several fairly well known possible interpretations come to mind, and may be helpful in motivating the ideas. For instance, an event might stand for the acceptance of a symbol by a non-deterministic automaton; or an event could represent a synchronized exchange of values by two concurrently active components in a communicating network.

For the purposes of this chapter, it suffices to think of an event as an instantaneous, atomic action; we make no assumptions about the structure of an event other than this. We will describe an abstract process in terms of its ability to perform or refuse to perform events.

1. Processes

To begin, let a set Σ of *events* be given; this set will be called the (universal) *alphabet*. In all examples of this thesis, Σ will be countable, since it seems reasonable to insist that any conceivable process be capable of performing only finitely many distinct events in any finite time. Let $\mathcal{P}(\Sigma)$ be the powerset of Σ, and $p\Sigma$ be the set of finite subsets of Σ. We use a, b, c to range over Σ and X, Y, Z to range over $p\Sigma$.

A *trace* is an element of Σ^*, the set of finite sequences of events. The *empty* trace, of length 0, is denoted $\langle\,\rangle$, and in general a trace of length n will have the form

$$\langle c_1, \ldots, c_n \rangle,$$

where each $c_i \in \Sigma$. Where no confusion can arise, we may omit the braces and write $c_1 \ldots c_n$ for such a trace. We use s, t, u to range over Σ^*. The *concatenation* of traces s and t is denoted st, and we say that s is a *prefix* of u, written $s \leq u$, if $u = st$ for some t. Thus, the empty trace is a prefix of any other trace, and any trace is a prefix of itself. In the case of a proper prefix we write $s < u$, when $s \leq u \;\&\; s \neq u$.

The behaviour of a process may be described by giving the set of events which it may perform initially, and for each of these events describing the subsequent behaviour. This is, of course, a recursive definition of behaviour. In any case, when the process exhibits non-deterministic behaviour, this is not an adequate description. The ability of a non-deterministic process to perform events may depend on the outcome of a non-deterministic choice. We adopt the simple view that the effect of each such choice is to remove a finite set of events from the set of events which the process might otherwise have performed next. This means that a non-deterministic decision amounts to, or can be modelled by, the choice of a *refusal set*, which must be a finite set of events; we are supposing that a process is only able to refuse finitely many events in a finite amount of time. If a process P can perform the sequence of actions s and then refuse all events in the set X, we say that the pair (s, X) is a *failure* of the process. Thus, a failure captures some finite aspect of process behaviour. For us, the behaviour of a process will be characterized by its set of possible failures. The following properties of failure sets capture the intuition behind our notion of process.

Definition 1.1.1: A *process* is a subset P of $\Sigma^* \times p\Sigma$ such that the following conditions hold:

(P0) $\quad (\langle\,\rangle, \emptyset) \in P$

(P1) $\quad (st, \emptyset) \in P \Rightarrow (s, \emptyset) \in P$

(P2) $\quad (s, X) \in P \;\&\; Y \subseteq X \Rightarrow (s, Y) \in P$

(P3) $\quad (s, X) \in P \;\&\; (s\langle c \rangle, \emptyset) \notin P \;\Rightarrow (s, X \cup \{ c \}) \in P.$ ∎

Definition 1.1.2: Let $P \subseteq \mathrm{PROC}$.

(1) $\quad \mathrm{traces}(P) = \{ t \mid \exists X.(t, X) \in P \}$

(2) $\quad \mathrm{initials}(P) = \{ c \mid \langle c \rangle \in \mathrm{traces}(P) \}$

(3) $\quad \mathrm{refusals}(P) = \{ X \mid (\langle\,\rangle, X) \in P \}.$ ∎

We explain conditions (P0)–(P3) as follows. The domain of the relation P consists of the set of all conceivable sequences of actions by the process.

Since the empty trace $\langle\rangle$ corresponds to the state of affairs before P has yet performed any action, the empty trace should be included in this set; this is the content of (P0). For (P1) we merely observe that when a process has performed a sequence u it must be the case that each prefix of u was performed earlier. Thus the first two conditions state that the set of traces of a process is non-empty and prefix-closed. Such a set of traces will be called a *tree*. Now suppose that, after performing a sequence s, P can (as the result of a non-deterministic choice) refuse all events in the set X. Then it is evident that the same decision results in the refusal of any subset $Y \subseteq X$; this explains (P3). Finally, if at some stage an event c is *impossible* it is reasonable to allow that event to be included in any refusal set, as in (P4).

Several elementary properties of processes are derived easily from this definition. For instance, an immediate corollary of Definition 1.1.1 is:

$$\text{traces}(P) = \{\, s \mid (s, \emptyset) \in P \,\},$$

since if s is a possible trace for P then P can at least refuse the empty set of events after performing this sequence of events. Similarly, by finitely many applications of $(P4)$ we can show:

$$(s, X) \in P \;\&\; sY \cap \text{traces}(P) = \emptyset \quad \Rightarrow \quad (s, X \cup Y) \in P,$$

where $sY = \{\, sx \mid x \in Y \,\}$. Thus, finite sets of impossible events can always be included in a refusal set.

Notation: The variables P, Q, R range over processes; S, T, U range over trees. The trivial tree $\{\langle\rangle\}$ will be denoted NIL.

The following definition shows the existence of a wide variety of processes. In particular, we see that for any tree T there is a largest and a smallest set of failures having trace set T and satisfying conditions (P0)–(P3). Of course, in general there will be many other processes with this trace set, but these special processes will be important later.

Theorem 1.1.3: For any tree T, the following definitions give processes with trace set T:

(1) $\text{chaos}(T) = \{\, (t, X) \mid t \in T \;\&\; X \in p\Sigma \,\}$

(2) $\text{det}(T) = \{\, (t, X) \mid t \in T \;\&\; X \in p\Sigma \;\&\; tX \cap T = \emptyset \,\}$

Proof. Trivial. ∎

Notice that chaos(T) includes all possible failures in which the trace belongs to T, and is therefore the largest failure set with traces T. In contrast,

$\det(T)$ contains only those failures which must belong to any process with trace set T, by condition (P3); it is thus the smallest such process.

Next we show that processes "persist" in the sense that the future behaviour of a process P after it has first performed a trace s is still represented by a process (denoted $P \underline{\text{after}}\ s$).

Definition 1.1.4: Let T be a tree, P a process and s a trace of P.

(i) $T \underline{\text{after}}\ s = \{ t \mid st \in P \}$

(ii) $P \underline{\text{after}}\ s = \{ (t, X) \mid (st, X) \in P \}$

Then $T \underline{\text{after}}\ s$ is a tree and $P \underline{\text{after}}\ s$ is a process.

Proof. Trivial. ∎

Theorem 1.1.5: For all processes P and traces s, t,

(1) $\text{traces}(P \underline{\text{after}}\ s) = \text{traces}(P) \underline{\text{after}}\ s$

(2) $P \underline{\text{after}} \langle\rangle = P$

(3) $(P \underline{\text{after}}\ s) \underline{\text{after}}\ t = P \underline{\text{after}}\ st$

(4) $\text{refusals}(P \underline{\text{after}}\ s) = P(s)$.

Proof. Trivial. ∎

We end this section with examples to illustrate our ideas on processes. It is hoped that these examples will serve to clarify the concepts introduced so far.

Examples.

Example 1. The process STOP has the following specification:

$$\begin{aligned} \text{STOP} &= \det(\text{NIL}) \\ \text{traces(STOP)} &= \{\langle\rangle\} \\ \text{refusals(STOP)} &= p\Sigma \\ \text{STOP} \underline{\text{after}} \langle\rangle &= \text{STOP}. \end{aligned}$$

This is the only process with trace set consisting only of the empty trace. It must refuse any event, since there is no event in the initials of STOP. Indeed, STOP represents *deadlock,* because it cannot participate in any action.

Example 2. RUN is the process satisfying:

$$\begin{aligned} \text{RUN} &= \det(\Sigma^*) \\ \text{traces(RUN)} &= \Sigma^* \\ \text{refusals(RUN)} &= \{\emptyset\} \\ \text{RUN} \underline{\text{after}}\ s &= \text{RUN} \quad \text{for all } s. \end{aligned}$$

This process cannot ever refuse to perform an event. At each stage it is possible for RUN to perform any event in Σ, and the only refusal set is the trivial one.

Example 3. The process CHAOS has the following definition:

$$\begin{aligned} \text{CHAOS} &= \text{chaos}(\Sigma^*) \\ \text{traces}(\text{CHAOS}) &= \Sigma^* \\ \text{refusals}(\text{CHAOS}) &= p\Sigma \\ \text{CHAOS}\ \underline{\text{after}}\ s &= \text{CHAOS} \quad \text{for all } s. \end{aligned}$$

Although CHAOS has the same traces as RUN, it also has the ability to refuse any set of events at any stage. Thus CHAOS is more non-deterministic than RUN.

Example 4. The process CHOOSE is given by

$$\begin{aligned} \text{CHOOSE} &= \{(s, X) \mid s \in \Sigma^* \ \&\ X \neq \Sigma\} \\ \text{traces}(\text{CHOOSE}) &= \Sigma^* \\ \text{refusals}(\text{CHOOSE}) &= p\Sigma - \{\Sigma\} \\ \text{CHOOSE}\ \underline{\text{after}}\ s &= \text{CHOOSE} \quad \text{for all } s. \end{aligned}$$

This process can refuse any finite set of events except Σ, so it must perform some event at each stage in its execution; every time it performs an event it can choose arbitrarily which event to perform next. If Σ is infinite, this process has the same failures as CHAOS.

2. Non-determinism

So far we have introduced processes as sets of *failures* built from traces and refusal sets. Each failure (s, X) of a process P represents a possible result of a non-deterministic choice which the process may make; the effect of this choice is to render the events in X impossible on the next step after s has been performed. It makes sense to speak of P as being more non-deterministic than another process Q if every non-deterministic decision available to Q is also possible for P. In this section we investigate the order induced on the space of processes by this idea.

Definition 1.2.1: The order $\sqsubseteq$ on PROC is the superset relation:

$$P \sqsubseteq Q \Leftrightarrow P \supseteq Q.$$

By definition, every process is a subset of PROC. This means that CHAOS is the most non-deterministic process of all, as befits its name. In addition, the set of all processes is obviously partially ordered by $\sqsubseteq$. These facts are recorded in the following lemma:

Lemma 1.2.2: The space (PROC, $\sqsubseteq$) is a partial order with least element CHAOS, i.e., for all processes P, Q, R

$$\begin{array}{ll} (1) & P \sqsubseteq Q \,\&\, Q \sqsubseteq P \Rightarrow P = Q \\ (2) & P \sqsubseteq Q \,\&\, Q \sqsubseteq R \Rightarrow P \sqsubseteq R \\ (3) & \text{CHAOS} \sqsubseteq P. \end{array}$$

Proof. Trivial. ∎

Of particular interest are the processes $\text{det}(T)$, $\text{chaos}(T)$ introduced in the previous section. For the *maximal* processes with respect to the ordering turn out to be those of the form $\text{det}(T)$, and we will call these the *deterministic* processes; by contrast, $\text{chaos}(T)$ is the *most* non-deterministic of all the processes with trace set T.

Lemma 1.2.3: For any process P with trace set T

$$\text{chaos}(T) \sqsubseteq P \sqsubseteq \text{det}(T).$$

Proof. By elementary set-theoretic properties, using the definitions of Theorem 1.1.3. ∎

Definition 1.2.4: A process P is *maximal* if, for all processes Q,

$$P \sqsubseteq Q \Rightarrow P = Q.$$

Lemma 1.2.5: P is maximal iff $P = \mathrm{det}(\mathrm{traces}(P))$.

Proof. Suppose $P = \mathrm{det}(\mathrm{traces}(P))$ and let $T = \mathrm{traces}(P)$. Let Q be any process such that $P \sqsubseteq Q$. We must show that $Q = \mathrm{det}(T)$. This is achieved by establishing that $\mathrm{traces}(Q) = T$ and invoking Lemma 1.2.3.

By hypothesis,

$$Q \subseteq \mathrm{det}(T),$$

and so the traces of Q are contained in T :

$$\mathrm{traces}(Q) \subseteq T.$$

If this inclusion is proper, there must be a trace $t \in T - \mathrm{traces}(Q)$. Without loss of generality we may assume that t is the shortest such trace, so that all proper prefixes of t belong to $\mathrm{traces}(Q)$. Since the empty trace is known to be a trace of Q there must be a trace s and event c such that $t = s\langle c\rangle$. But then s must be a trace of Q, and $s\langle c\rangle$ is not. This implies

$$(s, \{\, c \,\}) \in Q.$$

But $Q \subseteq \mathrm{det}(T)$ now gives us that $(s, \{\, c \,\}) \in \mathrm{det}(T)$. This contradicts the definition of $\mathrm{det}(T)$, since $s\langle c\rangle \in T$. Hence $\mathrm{traces}(T) = Q$. Now by Lemma 1.2.3, we have

$$Q \sqsubseteq \mathrm{det}(T) = P,$$

which, together with $P \sqsubseteq Q$ gives the desired result, i.e., $P = Q$. That completes the proof. ∎

Next we consider some structural properties of PROC and its partial order $\sqsubseteq$. We show that PROC is closed under arbitrary *non-empty unions*, and under *directed intersections*. This means that every non-empty subset of PROC has a greatest lower bound (infimum) and that every directed subset has a least upper bound (supremum). Technically, this means that the space of processes forms a *complete semi-lattice*. Moreover, we show that each process is uniquely characterized as the infimum of a set of *deterministic* processes. Finally, each process is the supremum of a naturally defined directed sequence of approximating processes.

Theorem 1.2.6: Let $\mathcal{D}$ be a non-empty set of processes. Then $\bigcup\mathcal{D}$ is a process.

Proof. Let $P = \bigcup\mathcal{D}$. We check conditions (P0)–(P3). By definition,

$$(s, X) \in P \Leftrightarrow \exists Q \in \mathcal{D}.(s, X) \in Q.$$

Hence, the trace set of P is a tree, being simply

$$\mathrm{traces}(P) = \bigcup\{\, \mathrm{traces}(Q) \mid Q \in \mathcal{D} \,\}.$$

Condition (P2) is similarly verified. Finally, suppose

$$(s, X) \in P \;\&\; s\langle c\rangle \notin \mathrm{traces}(P).$$

There must be some $Q \in \mathcal{D}$ such that $(s, X) \in Q$; but clearly $s\langle c\rangle \notin \text{traces}(Q)$, since $\text{traces}(Q) \subseteq \text{traces}(P)$. Hence, using (P3) for the process Q we find $s\langle c\rangle \notin \text{traces}(P)$, which in turn implies that $(s, X \cup \{c\}) \in P$, as required. This completes the proof. ∎

Theorem 1.2.7: Every process is the infimum of a set of deterministic processes.

Proof. Let P be a process and define a set of deterministic processes,

$$\mathcal{D} = \{ \det(T) \mid P \sqsubseteq \det(T) \}.$$

By Theorem 1.2.3,

$$\det(\text{traces}(P)) \in \mathcal{D}.$$

Thus, $\mathcal{D} \neq \emptyset$ and so $\bigsqcup \mathcal{D}$ is a process, by Theorem 1.2.6. Clearly, by definition,

$$P \sqsubseteq \bigsqcup \mathcal{D}.$$

To establish the converse, i.e., $\bigsqcup \mathcal{D} \sqsubseteq P$, we let (s, X) be a failure of P. Let T be the tree obtained by removing from $\text{traces}(P)$ all proper extensions of s. Formally, what is required can be defined as follows:

$$sX\Sigma^* = \{ s\langle c\rangle t \mid c \in X \;\&\; t \in \Sigma^* \}$$

$$T = \text{traces}(P) - sX\Sigma^*.$$

It is trivial to check that T is a tree; and, by construction, $(s, X) \in \det(T)$. Moreover, for all $t \neq s \in T$ we see that

$$\text{initials}(P \underline{\text{after}}\, t) = \{ c \in \Sigma \mid t\langle c\rangle \in T \},$$

and

$$\text{initials}(P \underline{\text{after}}\, s) - X = \{ c \in \Sigma \mid s\langle c\rangle \in T \}.$$

Hence, for all $Y \in p\Sigma$ and $t \in T$

$$(t, Y) \in \det(T) \Rightarrow (t, Y) \in P.$$

Thus, $(s, X) \in \det(T)$, as required. ∎

Definition 1.2.8: A set of processes $\mathcal{D}$ is *directed* iff it is non-empty and for all $P, Q \in \mathcal{D}$ there is an $R \in \mathcal{D}$ such that $P \sqsubseteq R \;\&\; Q \sqsubseteq R$.

Theorem 1.2.9: Let $\mathcal{D}$ be a directed set of processes. Then $\bigsqcap \mathcal{D}$ is a process; this is the least upper bound of $\mathcal{D}$.

Proof. Let $P = \bigsqcap \mathcal{D}$. We must verify conditions (P0)–(P3), but we give details only for (P3); the other proofs are similar. Observe firstly that by definition of P,

$$\text{traces}(P) = \bigcap \{ \text{traces}(Q) \mid Q \in \mathcal{D} \}.$$

Assume that

$$(s, X) \in P \;\&\; s\langle c\rangle \notin \text{traces}(P).$$

We want to establish that $(s, X \cup \{ c \}) \in P$. If not, there would be some $Q \in \mathcal{D}$ with

$$(s, X \cup \{ c \}) \notin Q.$$

But $(s, X) \in P \Rightarrow (s, X) \in Q$, by definition of P. So $s\langle c \rangle$ must be a trace of Q, by (P3). Since $s\langle c \rangle$ is not a trace of P, there must be a process $Q' \in \mathcal{D}$ with

$$s\langle c \rangle \notin \text{traces}(Q').$$

Again, we have

$$(s, X) \in Q'.$$

Hence, using (P3), we must have

$$(s, X \cup \{ c \}) \in Q'.$$

By directedness, there is an $R \in \mathcal{D}$ with

$$Q \sqsubseteq R \;\&\; Q' \sqsubseteq R.$$

But then $(s, X) \in R$, by hypothesis. Since $s\langle c \rangle \notin \text{traces}(Q')$ and $Q' \sqsubseteq R$, $s\langle c \rangle$ cannot be a trace of R. This implies that $(s, X \cup \{ c \}) \in R$; however, this contradicts the fact that $Q \sqsubseteq R$ and $(s, X \cup \{ c \}) \notin Q$. We therefore conclude that no such Q can exist, and that $(s, X \cup \{ c \}) \in P$, as required. This completes the proof. ∎

Corollary 1.2.10: Let $\{ P_n \mid n \geq 0 \}$ be a chain of processes, i.e.,

$$\forall n \geq 0 \quad P_n \sqsubseteq P_{n+1}.$$

Then $\bigcap_n P_n$ is a process.

Proof. A chain is a simple form of directed set. ∎

A chain is simply a sequence of increasingly deterministic processes, and the supremum is the limit of the sequence. For any process P there is an obvious way to construct a sequence of approximations $\{ P^{(n)} \mid n \geq 0 \}$, by making the n^{th} process behave like P for its first n steps and thereafter like a restricted form of CHAOS, restricted to have the same traces as P.

Definition 1.2.11: Let P be a process. Define for $n \geq 0$

$$\begin{aligned} P^{(n)} = &\{ (s, X) \mid \text{length}(s) < n \;\&\; (s, X) \in P \} \\ &\cup \{ (s, X) \mid \text{length}(s) \geq n \;\&\; (s, \emptyset) \in P \;\&\; X \in p\Sigma \} \end{aligned}$$

It is easy to check that each $P^{(n)}$ is a process, and that

$$\begin{aligned} P^{(n)} \sqsubseteq P^{(n+1)} \sqsubseteq P, \\ \bigsqcap_n P^{(n)} = P. \end{aligned}$$

The following conditions characterise $P^{(n)}$ for each n :

$$\begin{array}{lll} \text{(i)} & \text{traces}(P^{(n)}) = \text{traces}(P), & \\ \text{(ii)} & \text{refusals}(P^{(n)} \underline{\text{after}}\ s) = \text{refusals}(P \underline{\text{after}}\ s), & (\text{length}(s) < n) \\ \text{(iii)} & \text{refusals}(P^{(n)} \underline{\text{after}}\ s) = p\Sigma, & (\text{length}(s) \geq n). \end{array}$$

Thus each $P^{(n)}$ is an n-step approximation to P, and these approximations form a chain converging to P. If P itself is a finite set of failures then the terms in the sequence $P^{(n)}$ will eventually be equal to P. Thus, the approximating processes used in this construction include the finite failure sets, as well as the processes whose behaviour is well-defined for finitely many steps and thereafter wholly arbitrary. It is convenient to refer to such processes as *finite*. Finite processes will be used in later sections in proving properties of processes: we have shown that the failure set of any process is uniquely determined as the limit of finite processes.

Summary.

The ordering $\sqsubseteq$ is a measure of non-determinacy, and it makes PROC into a complete semi-lattice: least upper bounds exist for directed sets, and greatest lower bounds exist for non-empty sets. The maximal processes are termed *deterministic* and every process is the greatest lower bound of a set of deterministic processes. Each process P is also definable as the limit of a chain of finite processes $\{ P^{(n)} \mid n \geq 0 \}$.

3. Functions on processes

Suppose now that we wish to construct new processes from old; in other words, we have a function F : PROC $\rightarrow$ PROC which allows us to build the process $F(P)$ from P. There are several constraints on the nature of F which it might be reasonable to impose if the construction is to be reasonable. For instance, when the argument P is made more deterministic we could reasonably expect the result $F(P)$ to become more deterministic (at least, no less deterministic.) This amounts to the *monotonicity* of F. If, in addition, F preserves the limits of chains, F is said to be *continuous*. A function which preserves infima of non-empty sets of processes is called *distributive*.

Definition 1.3.1: Let F : PROC $\rightarrow$ PROC be a function. F is *monotone* if, for all processes P and Q,

$$P \sqsubseteq Q \Rightarrow F(P) \sqsubseteq F(Q).$$

Definition 1.3.2: F is *continuous* iff, for all chains $\{ P_n \mid n \geq 0 \}$

$$F(\bigsqcap_n P_n) = \bigsqcap_n F(P_n).$$

Definition 1.3.3: F is *distributive* iff, for all non-empty sets $\mathcal{D}$ of processes

$$F(\bigsqcup \mathcal{D}) = \bigsqcup \{ F(P) \mid P \in \mathcal{D} \}.$$

Note that a continuous function is necessarily monotone, and that distributivity also implies monotonicity.

Examples.

Example 1.

$$\lambda P.\text{chaos}(\text{traces}(P))$$

This function is continuous and distributive.

Example 2.

$$\lambda P.\text{det}(\text{traces}(P))$$

This function is not monotone, and hence not continuous or distributive.

Example 3. Any function f : TREE $\rightarrow$ TREE can be used to define a function F : PROC $\rightarrow$ PROC as in

$$F(P) = \bigsqcup \{ \text{det}(f(T)) \mid P \sqsubseteq \text{det}(T) \}$$

This function will be monotone.

The second and third examples above are functions which preserve determinism in the sense that they yield deterministic results when applied to deterministic arguments. Such a function will be called *deterministic*.

Definition 1.3.4: F is *deterministic* if $F(P)$ is deterministic whenever P is deterministic.

It is a well known result that a monotone function on a complete semi-lattice has a least fixpoint. This fact is so well established that it is achieving the status of a "folk theorem" [LNS], to which paper the reader is referred for more details: there is disagreement on exactly who proved this result first. Since PROC is a complete semi-lattice, this result is applicable to the space of processes; it enables us to use recursive definitions of processes provided the constructions involved are monotone. Moreover, for continuous functions, the least fixpoint has an explicit formulation as the limit of a chain.

Theorem 1.3.5: Let $(D, \sqsubseteq)$ be a complete semi-lattice with least element $\perp$, and let $F : D \to D$ be a monotone function. Then F has a least fixpoint, i.e., there is an element $\mu F \in D$ such that

$$\begin{array}{ll} \text{(i)} & F(\mu F) = \mu F \\ \text{(ii)} & F(x) = x \Rightarrow \mu F \sqsubseteq x \end{array}$$

If F is continuous, the least fixpoint is given by:

$$\mu F = \bigsqcup_n F^n(\perp).$$

∎

In our domain of processes, the bottom element is CHAOS and the ordering is reverse inclusion. In the next chapter we introduce a collection of functions on processes; with one notable exception, all these functions are continuous. We can use this Theorem to assert that for such a function F the least fixpoint exists and is the intersection of the chain of processes obtained by iterating the application of F to CHAOS.

Chapter 2

Process operations

0. Introduction

In the previous chapter we introduced a domain PROC of processes over a universal alphabet Σ of events. These processes are intended to represent the possible behaviours of a class of non-deterministic communicating agents, and the events are taken to represent indivisible atomic actions of processes. In general a particular process only uses a subset of Σ in its traces. Correspondingly, we say that a subset $A \subseteq \Sigma$ is an alphabet for P if traces$(P) \subseteq A^*$. If we refer to *the* alphabet of P we intend the smallest such set.

Now we define a collection of operations on processes, motivated by intuitively reasonable methods of combining communicating agents; our choice of operators, while not claimed to be universally ideal, is based on those used by Hoare [H1] in his original paper on Communicating Sequential Processes, and in subsequent developments [H2],[HBR]. We will see that many interesting derived operators can be constructed, and that our process operations are powerful enough to allow expression of many parallel programming problems and examples.

We now introduce an abstract version of CSP based on a set of operations on our domain of processes. These operations will be referred to as *CSP operations*. If we wanted to separate semantic issues from syntax, we could begin with an abstract syntax for our version of the CSP language and define a semantic function mapping each term to its denotation as a failure set. We see no need to do so in this chapter, as the notation used to describe process operations is itself very suggestive of the abstract syntax we have in mind.

In an effort to stay faithful to Hoare's system we use a notation very similar to that of CSP. Each operation will be introduced with an informal explanation of the way the constructed process is intended to behave, as a function of the behaviours of the arguments.

0. Inaction

As we saw in Chapter 1, the process STOP is incapable of performing any event, and refuses every finite set of events. It has the failure set:

$$\text{STOP} = \{ (\langle\rangle, X) \mid X \in p\Sigma \}.$$

We regard STOP as representing *deadlock,* the inability to participate in any action.

1. Prefixing

Let P be a process and c be an event. The process $(c \to P)$ is first to perform event c and thereafter to behave like P; this operation is "prefixing" action c to the process P. Another interpretation is that the event c is a *guard* which must be passed before the process can begin to behave like P. We may refer to a process of this form as a guarded process. The first event in any non-empty trace of $(c \to P)$ must be c itself; and at the initial stage $(c \to P)$ cannot refuse to perform c. The construction has a simple formal definition and several obvious properties.

Definition 2.1.1:

$$(c \to P) = \{ (\langle\rangle, X) \mid c \notin X \} \cup \{ (\langle c \rangle t, X) \mid (t, X) \in P \}.$$

Theorem 2.1.2: For all processes P and events c, $(c \to P)$ is a process; it has the following properties:

(1) $\quad \text{traces}(c \to P) = \{ \langle\rangle \} \cup \{ \langle c \rangle t \mid t \in \text{traces}(P) \}$

(2) $\quad \text{refusals}(c \to P) = \{ X \mid c \notin X \}$

(3) $\quad (c \to P) \underline{\text{after}} \langle c \rangle t = P \underline{\text{after}}\, t.$

Proof. If P is a process, it is straightforward to verify that $(c \to P)$ satisfies conditions (P0)–(P3), and is therefore a process. The properties (1)–(3) are immediate, from Definition 2.1.1. ∎

Theorem 2.1.3: $\lambda P.(c \to P)$ is monotone, continuous, distributive and deterministic.

Proof. We give details only for continuity; the other properties are just as easy to establish. Let $\{ P_n \mid n \geq 0 \}$ be a chain of processes, so that

$$P_n \sqsubseteq P_{n+1}, \qquad (n \geq 0).$$

Let $P = \bigcap_n P_n$ be the limit of this chain. Assuming monotonicity, we know that the processes $(c \to P_n)$ also form a chain. We must show that the limit of this chain is the process $(c \to P)$: we need to establish the identity

$$\bigcap_n (c \to P_n) = (c \to \bigcap_n P_n).$$

This is done by proving that these two processes have the same failures, by induction on the length of traces. The base step is easy:

$$\begin{aligned}(\langle\rangle, X) \in (c \to P) &\Leftrightarrow c \notin X \\ &\Leftrightarrow \forall n.\, (\langle\rangle, X) \in (c \to P_n) \\ &\Leftrightarrow (\langle\rangle, X) \in \bigcap_n (c \to P_n).\end{aligned}$$

For the inductive step, it is clear from Theorem 2.1.2 that any non-empty trace of either process begins with the prefixed event c. It is enough, therefore, to consider a trace of the form $\langle c \rangle t$:

$$\begin{aligned}(\langle c \rangle t, X) \in (c \to P) &\Leftrightarrow (t, X) \in P \\ &\Leftrightarrow \forall n.\, (t, X) \in P_n \\ &\Leftrightarrow \forall n.\, (\langle c \rangle t, X) \in (c \to P_n) \\ &\Leftrightarrow (\langle c \rangle t, X) \in \bigcap_n (c \to P_n),\end{aligned}$$

as required. That completes the proof of continuity. ∎

Notice that the prefixing operation generalizes naturally to the case when a finite sequence of events is prefixed onto the behaviour of a process. We will use the notation $(s \to P)$ for the result of prefixing the trace s to P. The process $(s \to P)$ first must perform the sequence s and thereafter behaves like P.

Definition 2.1.4: For all processes P and traces s, the process $(s \to P)$ is defined by induction on the length of s :

$$\begin{aligned}&(1) \quad (\langle\rangle \to P) = P \\ &(2) \quad (\langle c \rangle t \to P) = (c \to (t \to P)).\end{aligned}$$

As a trivial consequence of this definition, we have the simple identity

$$(s \to (t \to P)) \doteq (st \to P).$$

Examples.

Example 1. The process $(a \to \text{STOP})$ performs a and then deadlocks, unable to perform any further action.

Example 2. Since the function $\lambda P.(a \rightarrow P)$ is continuous, it has least fixpoint $\bigcap_n P_n$, where:

$$P_0 = \text{CHAOS},$$
$$P_{n+1} = (a \rightarrow P_n).$$

An inductive proof shows that $P_n = (a^n \rightarrow \text{CHAOS})$. Notice that this recursion has only one fixpoint, since the least fixpoint turns out to be deterministic. This fact about fixpoints will be very useful.

2. Non-deterministic choice

Let P and Q be processes. We wish to construct a process which can behave either like P or like Q, the choice being arbitrary. The greatest lower bound of P and Q, denoted $P \sqcap Q$, has this ability. We call this combination "P or Q." This process should be able to behave like either of P and Q, and decides autonomously which one; thus, it can perform any sequence of events possible for either process, but similarly it may refuse any set of events that either process could have refused at the same stage.

Definition 2.2.1:

$$P \sqcap Q = P \cup Q.$$

Corollary 2.2.2:

$$\begin{array}{ll} (1) & \text{traces}(P \sqcap Q) = \text{traces}(P) \cup \text{traces}(Q) \\ (2) & \text{refusals}(P \sqcap Q) = \text{refusals}(P) \cup \text{refusals}(Q) \\ (3) & (P \sqcap Q) \underline{\text{after}}\, s = (P \underline{\text{after}}\, s) \sqcap (Q \underline{\text{after}}\, s). \end{array}$$

Note that the term $P \underline{\text{after}}\, s$ denotes the empty set when s is not a possible trace for P; thus, when c is an initial event of only one of the two constituent processes and $s = ct$, only one of the terms in the right-hand side of (3) contributes. It is convenient to adopt this as a general notational abbreviation, when strictly speaking we should have written out (3) in full as:

$$\begin{array}{lll} (P \sqcap Q) \underline{\text{after}}\, s = P \underline{\text{after}}\, s, & & \text{if } s \in \text{traces}(P) - \text{traces}(Q). \\ \quad = Q \underline{\text{after}}\, s, & & \text{if } s \in \text{traces}(Q) - \text{traces}(P). \\ \quad = (P \underline{\text{after}}\, s) \sqcap (Q \underline{\text{after}}\, s), & & \text{if } s \in \text{traces}(P) \cap \text{traces}(Q). \end{array}$$

The following result states that nondeterministic choice is idempotent, symmetric, associative, and has CHAOS as a zero element; it also gives the obvious connection between nondeterministic choice and the nondeterminism ordering. The proofs are straightforward and omitted.

Corollary 2.2.3:

$$\begin{array}{ll} (1) & P \sqcap P = P \\ (2) & P \sqcap Q = Q \sqcap P \\ (3) & P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R \\ (4) & P \sqcap \text{CHAOS} = \text{CHAOS} \\ (5) & P \sqcap Q = P \Leftrightarrow P \sqsubseteq Q. \end{array}$$

Theorem 2.2.4: $\lambda P, Q.P \sqcap Q$ is monotone, continuous and distributive.

Proof. Elementary properties of intersection and union. ∎

Examples.

Example 1. Let $a, b \in \Sigma$ be distinct events. The processes $(a \rightarrow \text{STOP})$ and $(b \rightarrow \text{STOP})$ are deterministic. Define the process P_{ab} as follows:

$$P_{ab} = (a \rightarrow \text{STOP}) \sqcap (b \rightarrow \text{STOP}).$$

This process is not deterministic, since on the first step each of the events a, b are possible but either one or the other can be refused.

$$\begin{aligned} \text{traces}(P_{ab}) &= \{ \langle\rangle, \langle a \rangle, \langle b \rangle \} \\ \text{initials}(P_{ab}) &= \{ a, b \} \\ \text{refusals}(P_{ab}) &= \{ X \mid a \notin X \vee b \notin X \}. \end{aligned}$$

Example 2. The function $F(P) = (a \rightarrow P) \sqcap (b \rightarrow P)$ is continuous, being built by composition from the continuous operations of prefixing and non-deterministic choice. The least fixpoint is the limit of the following chain:

$$\begin{aligned} P_0 &= \text{CHAOS} \\ P_{n+1} &= (a \rightarrow P_n) \sqcap (b \rightarrow P_n), \end{aligned}$$

and the limit process P clearly has the following properties:

$$\begin{aligned} \text{traces}(P) &= \{ a, b \}^* \\ \text{refusals}(P) &= \{ X \mid a \notin X \vee b \notin X \} \\ P \underline{\text{after}}\ s &= P \qquad \text{for all } s \in \text{traces}(P). \end{aligned}$$

Indeed, these conditions characterise any fixpoint of F. Since they determine completely the traces and refusals at each stage of the behaviour, we see that F has a unique fixpoint. Note also that P is able to choose at each stage to refuse either a or b, but not both. This makes P more non-deterministic than $\text{det}(a^*)$, which always refuses b; similarly for $\text{det}(b^*)$, which always refuses a; more non-deterministic than $\text{det}(\{ a, b \}^*)$, which refuses neither a nor b; but less so than $\text{chaos}(\{ a, b \}^*)$, which can refuse both a and b.

3. Controllable choice

The operation $P \sqcap Q$ formed a non-deterministic choice between alternatives. Next we introduce a form of conditional composition, for which we use the notation $P \Box Q$. The intention is that $P \Box Q$ should behave either like P or like Q, but the decision can be affected by its environment on the first step; whereas $P \sqcap Q$ could initially refuse a set X if either P refused it or Q refused it, we insist that $P \Box Q$ refuse a set if and only if both P and Q can refuse it. This means that $P \Box Q$ cannot commit itself in favour of (say) P. If the environment offers an initial event c which is acceptable only by P then $P \Box Q$ performs it and thereafter behaves like P; similarly the process behaves like Q if the environment offers an event initial only for Q. In the case where the environment's offer is possible for both P and Q, the choice is made non-deterministically.

Definition 2.3.1:

$$P \Box Q = \{(\langle\rangle, X) \mid (\langle\rangle, X) \in P \cap Q\} \cup \{(s, X) \mid s \neq \langle\rangle \,\&\, (s, X) \in P \cup Q\}.$$

Theorem 2.3.2: If P and Q are processes, then so is $P \Box Q$. It has the following properties:

(1) $\quad \text{traces}(P \Box Q) = \text{traces}(P) \cup \text{traces}(Q)$
(2) $\quad \text{refusals}(P \Box Q) = \text{refusals}(P) \cap \text{refusals}(Q)$.
(3) $\quad (P \Box Q) \underline{\text{after}}\ s = (P \underline{\text{after}}\ s) \sqcap (Q \underline{\text{after}}\ s), \quad \text{for } s \neq \langle\rangle$.

Proof. Straightforward, using standard set-theoretic properties of union and intersection. ∎

Like the other form of choice operator, this operation is idempotent, symmetric and associative. However, STOP is a unit for $\Box$, since a combination $P \Box \text{STOP}$ only refuses a set if P refuses it and only performs an event if P performs it: this process is identical to P. As we might expect, nondeterministic choice is in general more nondeterministic than controllable choice. These results are summarised below; we omit the proofs.

Theorem 2.3.3:

(i) $\quad P \Box P = P$
(ii) $\quad P \Box Q = Q \Box P$
(iii) $\quad P \Box (Q \Box R) = (P \Box Q) \Box R$
(iv) $\quad P \Box \text{STOP} = P$
(v) $\quad P \sqcap Q \sqsubseteq P \Box Q$.

We should also note some facts about $\Box$ applied to guarded processes. If B is a finite set of events, and if P_b is a process for each $b \in B$, then the combination

$$\Box_{b \in B}(b \to P_b)$$

denotes a process which initially accepts any event in B (and cannot refuse any of these events) and thereafter behaves like the corresponding component process. Note the structural similarity with the form of *guarded command* in Hoare's original CSP. We will refer to this conditional combination of guarded processes also as a guarded process. If the environment of a guarded process of this form wants to pass a particular guard it can do so because the process cannot refuse this event. If, however, we form a combination

$$(a \to P) \Box (a \to Q)$$

in which the two alternatives start with the same event, although the initial event cannot be refused it is nondeterministic which component performs it; this process is identical to the process

$$(a \to P) \sqcap (a \to Q).$$

There is an obvious connection here with Hoare's guarded construction in CSP.

Lemma 2.3.4: For all processes P, Q and all events a,

$$(a \to P) \Box (a \to Q) = (a \to P) \sqcap (a \to Q) = (a \to P \sqcap Q).$$

Next we establish that conditional choice enjoys the same continuity and distributivity properties as nondeterministic choice.

Theorem 2.3.5: $\lambda P, Q.P \Box Q$ is monotone, continuous and distributive.

Proof. As an example, we show distributivity. Let $\mathcal{D}$ be a non-empty set of processes. We will show that, for any P, the identity

$$P \Box \bigcup \mathcal{D} = \bigcup \{ P \Box Q \mid Q \in \mathcal{D} \}$$

holds. As usual, since the definition of this process operation has two clauses, we split the proof into two parts. For the empty trace, we have

$$\begin{aligned}
(\langle\rangle, X) \in P \Box \bigcup \mathcal{D} &\Leftrightarrow (\langle\rangle, X) \in P \ \& \ (\langle\rangle, X) \in \bigcup \mathcal{D} \\
&\Leftrightarrow \exists Q \in \mathcal{D}. (\langle\rangle, X) \in P \ \& \ (\langle\rangle, X) \in Q \\
&\Leftrightarrow \exists Q \in \mathcal{D}. (\langle\rangle, X) \in P \Box Q \\
&\Leftrightarrow (\langle\rangle, X) \in \bigcup \{ P \Box Q \mid Q \in \mathcal{D} \}
\end{aligned}$$

as required. A similar argument establishes that for all nonempty traces s,

$$(s, X) \in P \Box \bigcup \mathcal{D} \Leftrightarrow \exists Q. (s, X) \in P \Box Q,$$

from which the result follows. ∎

Examples.

Example 1. The process Q_{ab} has the same traces as P_{ab} above; however, Q_{ab} cannot refuse any of its initial events. Indeed, this process is deterministic.

$$Q_{ab} = (a \rightarrow \text{STOP}) \,\Box\, (b \rightarrow \text{STOP})$$
$$\text{traces}(Q_{ab}) = \{\langle\,\rangle, \langle a\rangle, \langle b\rangle\}$$
$$\text{refusals}(Q_{ab}) = \{X \mid a \notin X \;\&\; b \notin X\}.$$

Example 2. The function $G(P) = (a \rightarrow P) \,\Box\, (b \rightarrow P)$ is continuous, since prefixing and $\Box$ are both continuous and composition preserves continuity. Its least fixpoint is the limit of the chain:

$$Q_0 = \text{CHAOS}$$
$$Q_{n+1} = (a \rightarrow Q_n) \,\Box\, (b \rightarrow Q_n).$$

This limit, Q, has the following properties:

$$\text{traces}(Q) = \{a, b\}^*$$
$$\text{refusals}(Q) = \{X \mid a \notin X \;\&\; b \notin X\}$$
$$Q \,\underline{\text{after}}\, s = Q, \quad \text{for all } s \in \{a, b\}^*.$$

Clearly, $Q = \det(\{a, b\}^*)$.

Example 3. Despite Example 1, it is not always the case that $\Box$ preserves determinism; obviously the introduction of a non-deterministic choice takes place when the initials of the two processes have a non-empty intersection and when there is an event common to the initials after which each process would have different behaviour. An example of this type was mentioned in the above discussion of guarded processes. For another example, consider the process:

$$R = (a \rightarrow \text{STOP}) \,\Box\, (a \rightarrow (b \rightarrow \text{STOP})).$$

A simple calculation shows that

$$R \,\underline{\text{after}}\, \langle a\rangle = \text{STOP} \sqcap (b \rightarrow \text{STOP}).$$

This reflects the fact that, once the initial event a has occurred the process has made a nondeterministic choice between two alternatives, either of which could have started with the a event.

The next result shows that the two forms of choice operation *distribute* over each other.

Theorem 2.3.6:

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

Proof. Elementary properties of intersection and union. ∎

Although conditional choice is not a deterministic operation, it does preserve determinism in cases where the alternative processes have disjoint initials.

Theorem 2.3.7: Provided $S \cap T = \text{NIL}$,

$$\det(S) \,\square\, \det(T) = \det(S \cup T).$$

Proof. Trivial. ∎

We end this section by stating an identity for conditional composition of two guarded processes; it is easily proved from Lemma 2.3.4.

Lemma 2.3.8: For the guarded processes

$$P = \square_{b \in B}(b \to P_b),$$
$$Q = \square_{c \in C}(c \to Q_c),$$

the conditional composition $P \,\square\, Q$ is

$$P \,\square\, Q = \square_{a \in B \cup C}(a \to R_a),$$

where
$$R_a = P_a, \quad \text{if } a \in B - C,$$
$$= Q_a, \quad \text{if } a \in C - B,$$
$$= P_a \sqcap Q_a, \quad \text{if } a \in B \cap C.$$

4. Parallel composition

The process $P \parallel Q$ represents a form of parallel activity of P and Q. It performs an event only if both P and Q components agree to perform it; dually, it refuses an event if either component chooses to refuse it. Thus, the traces of the parallel combination are the common traces of the two components, while each refusal of $P \parallel Q$ is a union of a refusal of P with a refusal of Q. Once an event c has been performed, each component has performed it; the subsequent behaviour is that of P after $\langle c \rangle$ running in parallel with Q after $\langle c \rangle$. More generally, once $P \parallel Q$ has performed the sequence s its subsequent behaviour is that of the parallel composition of P after s and Q after s.

Definition 2.4.1:

$$P \| Q = \{ (s, X \cup Y) \mid (s, X) \in P \,\&\, (s, Y) \in Q \}.$$

Theorem 2.4.2: Whenever P and Q are processes, $P \| Q$ is a process and:

(i) $\text{traces}(P \parallel Q) = \text{traces}(P) \cap \text{traces}(Q)$
(ii) $\text{refusals}(P \| Q) = \{ X \cup Y \mid X \in \text{refusals}(P) \,\&\, Y \in \text{refusals}(Q) \}$
(iii) $(P \| Q) \text{ after } s = (P \text{ after } s) \| (Q \text{ after } s)$.

Proof. Elementary. ∎

Parallel composition is symmetric, associative and STOP is a zero. In general, parallel composition is not, however, idempotent; a process of the form $P \| P$ can generally refuse more sets than can P alone.

Theorem 2.4.3:

(1) $P \| P \sqsubseteq P$
(2) $P \| Q = Q \| P$
(3) $P \| (Q \| R) = (P \| Q) \| R$
(4) $P \| \text{STOP} = \text{STOP}$

Proof. These results follow immediately from the definition of $\|$. ∎

Theorem 2.4.4: $\lambda P, Q.P \| Q$ is monotone, continuous, distributive and deterministic.

Proof. We give details only for continuity and determinism, the other properties being straightforward.

Continuity. Let $\{ P_n \mid n \geq 0 \}$ be a chain of processes with limit P. We must show that, for any process Q, the processes $\{ (P_n \| Q) \mid n \geq 0 \}$ form a

chain with limit $P \| Q$. Monotonicity of $\|$ implies the chain property. Also,

$$P_n \sqsubseteq P \Rightarrow P_n \| Q \sqsubseteq P \| Q, \qquad \forall n \geq 0.$$

Hence, we get one half of the equation:

$$\bigsqcap_n P_n \| Q \sqsubseteq P \| Q.$$

For the converse relation, suppose $(s, Z) \in \bigsqcap_n P_n \| Q$. We want to show that $(s, Z) \in P \| Q$. For each $n \geq 0$ we know there are sets X_n, Y_n of events with

$$Z = X_n \cup Y_n, \qquad (s, X_n) \in P_n, \qquad (s, Y_n) \in Q.$$

Since Z is a finite set, in the list of pairs $\{ (X_n, Y_n) \mid n \geq 0 \}$ some pair, say (X_m, Y_m), must appear infinitely often. But this implies that $(s, Y_m) \in Q$, and that $(s, X_m) \in P_n$ for infinitely many n. By the chain condition this gives $(s, X_m) \in P$. Putting

$$X = X_m, \quad Y = Y_m,$$

we have

$$X \cup Y = Z, \qquad (s, X) \in P, \qquad (s, Y) \in Q$$

and hence

$$(s, Z) \in P \| Q.$$

Thus parallel composition is continuous.

Determinism. To show that $\|$ is deterministic, we prove that

$$\det(S) \| \det(T) = \det(S \cap T).$$

This is easy:

$$\begin{aligned}(s, Z) \in \det(S) \| \det(T) &\Leftrightarrow \exists X, Y.(s, X) \in \det(S) \,\&\, (s, Y) \in \det(T) \\ &\quad \& X \cup Y = Z \\ &\Leftrightarrow s \in S \,\&\, s \in T \,\&\, \forall c \in Z.(s\langle c \rangle \notin S \vee s\langle c \rangle \notin T) \\ &\Leftrightarrow s \in S \cap T \,\&\, \forall c \in Z.s\langle c \rangle \notin S \cap T \\ &\Leftrightarrow (s, Z) \in \det(S \cap T).\end{aligned}$$

That completes the proof of Theorem 2.4.4. ∎

The following results show how a parallel composition of guarded processes behaves:

Lemma 2.4.5: Let P_B and Q_C be the processes

$$P_B = \Box_{b \in B}(b \to P_b),$$
$$Q_C = \Box_{c \in C}(c \to Q_c).$$

Then the parallel composition $P_B \| Q_C$ is:

$$P_B \| Q_C = \Box_{a \in B \cap C}(a \to P_a \| Q_a).$$

Proof. Trivial. ∎

Corollary: For all processes P and Q, and all events a, b

$$\begin{aligned}(a \to P) \| (b \to Q) &= (a \to P \| Q), \quad \text{if } a \neq b. \\ &= \text{STOP}, \qquad\quad \text{if } a = b.\end{aligned}$$

∎

Examples.

Example 1. For any process P, the following equations hold:

$$\begin{aligned}P \| \text{STOP} &= \text{STOP} \\ P \| \text{RUN} &= P \\ P \| \text{CHAOS} &= \text{chaos}(\text{traces}(P)).\end{aligned}$$

Example 2. Let a and b be distinct events. Define processes P_{ab} and Q_{ab} :

$$\begin{aligned}P_{ab} &= (a \to \text{STOP}) \sqcap (b \to \text{STOP}) \\ Q_{ab} &= (a \to \text{STOP}) \,\Box\, (b \to \text{STOP}).\end{aligned}$$

Using distributivity and the result of Theorem 2.4.5,

$$\begin{aligned}P_{ab} \| (a \to \text{STOP}) &= ((a \to \text{STOP}) \| (a \to \text{STOP})) \\ &\quad \sqcap ((b \to \text{STOP}) \| (a \to \text{STOP})) \\ &= (a \to \text{STOP}) \sqcap \text{STOP}.\end{aligned}$$

Similarly,

$$P_{ab} \| (b \to \text{STOP}) = (b \to \text{STOP}) \sqcap \text{STOP}.$$

It follows that

$$\begin{aligned}P_{ab} \| P_{ab} &= (a \to \text{STOP}) \sqcap (b \to \text{STOP}) \sqcap \text{STOP} \\ &= P_{ab} \sqcap \text{STOP}.\end{aligned}$$

This is an example where $P \parallel P \neq P$. Thus parallel composition is not generally idempotent; of course, the proof of Theorem 2.4.4 shows that deterministic processes do have this property. Since Q_{ab} is deterministic, we have $Q_{ab} \| Q_{ab} = Q_{ab}$. Moreover,

$$\begin{aligned}Q_{ab} \| (a \to \text{STOP}) &= (a \to \text{STOP}) \\ Q_{ab} \| (b \to \text{STOP}) &= (b \to \text{STOP}).\end{aligned}$$

Thus, by distributivity,

$$Q_{ab} \| P_{ab} = (a \to \text{STOP}) \sqcap (b \to \text{STOP}) = P_{ab}.$$

These processes also show that $\|$ does not distribute over $\Box$, because

$$\begin{aligned}P_{ab} \| ((a \to \text{STOP}) \,\Box\, (b \to \text{STOP}) &= P_{ab} \| Q_{ab} \\ &= P_{ab}, \\ (P_{ab} \| (a \to \text{STOP})) \,\Box\, (P_{ab} \| (b \to \text{STOP})) &= P_{ab} \sqcap \text{STOP} \\ &\neq P_{ab}.\end{aligned}$$

Failures and parallel composition.

There is a simple connection between failure sets and parallel composition. Let s be any finite sequence of events and X be any finite set of events. The tree sX defines a very simple kind of deterministic process: one which can perform the sequence s and then perform any of the events in X before stopping. Intuitively, we can see that if P is a process for which (s, X) is a possible failure, then running P in parallel with the deterministic process $\det(sX)$ can result in deadlock after s has occurred. The converse is also true. More formally, this is reflected in the following fact, the proof of which is trivial:

Lemma 2.4.6: For all processes P and all failures (s, X),

$$(s, X) \in P \Leftrightarrow (P \,||\, \det(sX)) \sqsubseteq (s \rightarrow \text{STOP}).$$

Restriction: a derived operation.

Let B be any set of events, and P be any process. The combination

$$P \,||\, \det(B^*)$$

behaves like a *restricted* version of P; it can only perform events in the set B. This form of parallel composition merits a special notation, as it is very useful. We write

$$P\lceil B = P \,||\, \det(B^*),$$

and call this process "P restricted to B." If all events appearing in the trace set of P are already in B, restriction has no effect, and $P\lceil B = P$ in this case. Since $\det(\emptyset^*) = \text{STOP}$, we have already seen that $P\lceil\emptyset = \text{STOP}$, as we would expect. Some elementary properties of restriction follow from our previous results:

$$\begin{aligned}(P\lceil B)\lceil C &= (P \,||\, \det(B^*)) \,||\, \det(C^*) \\ &= P \,||\, (\det(B^*) \,||\, \det(C^*)) \\ &= P \,||\, \det(B^* \cap C^*) \\ &= P \,||\, \det((B \cap C)^*). \\ &= P\lceil(B \cap C)\end{aligned}$$

A special case is when we wish to prevent an event b, or more generally all events in a set B, from occurring. This can be achieved by restricting to the complement of B. Writing $P{-}B$ for the result of forbidding B in P,

$$P{-}B = P\lceil(\Sigma{-}B),$$

the following laws are obvious:

$$(P{-}B){-}C = (P{-}C){-}B = P{-}(B \cup C).$$

5. Interleaving

Another form of parallel composition allows arbitrary interleaving of the actions of the two component processes P and Q. We denote this combination $P|||Q$. Its traces are obtained by merging a trace of P with a trace from Q. At each stage we specify that an event can occur if either of the component processes can perform it; this means that $P|||Q$ can refuse a set X of events if both P and Q choose to refuse X. When an event occurs which was permissible for both components, the choice as to which component performs it is to be made non-deterministically. Of course, a given trace of $P|||Q$ may arise in many distinct ways as merges of traces of P and Q. The following definition makes precise the notion of merging two traces; induction on lengths of traces can be used to show that the definition is well-founded.

Definition 2.5.1: The function $\text{merge} : \Sigma^* \times \Sigma^* \to p\Sigma^*$ is given by:

$$\begin{array}{ll} \text{(i)} & \text{merge}(\langle\,\rangle, s) = \text{merge}(s, \langle\,\rangle) = \{\, s \,\} \\ \text{(ii)} & \text{merge}(\langle a\rangle s, \langle b\rangle t) = \{\, \langle a\rangle u \mid u \in \text{merge}(s, \langle b\rangle t \,\} \\ & \qquad\qquad \cup \{\, \langle b\rangle u \mid u \in \text{merge}(\langle a\rangle s, t) \,\}. \end{array}$$

For example, $\text{merge}(\langle a\rangle, \langle b\rangle) = \{\, \langle a, b\rangle, \langle b, a\rangle \,\}$. If $u \in \text{merge}(s, t)$ we say that s and t merge to u, or that u is a merge of s and t. Note that there are only a finite number of merges for any two traces. Some elementary properties of the merge function are now stated for reference. Proofs are omitted.

Corollary 2.5.2:

$$\begin{array}{ll} \text{(i)} & \text{merge}(s, t) = \text{merge}(t, s) \\ \text{(ii)} & u \in \text{merge}(s, t)\ \&\ u \in \text{merge}(s, t') \Rightarrow t = t' \\ \text{(iii)} & u \in \text{merge}(s, t) \Rightarrow \text{length}(u) = \text{length}(s) + \text{length}(t) \\ \text{(iv)} & u \in \text{merge}(s, t)\ \&\ u' < u \Rightarrow \exists s' \leq s, t' \leq t.u' \in \text{merge}(s', t'). \end{array}$$

Definition 2.5.3: For any sets S and T of traces, $S|||T$ denotes the set of all merges of a trace from S with a trace from T :

$$S|||T = \bigcup\{\, \text{merge}(s, t) \mid s \in S\ \&\ t \in T \,\}.$$

Lemma 2.5.4:

$$\begin{array}{ll} (1) & S|||T = T|||S \\ (2) & (S|||T)|||U = S|||(T|||U) \\ (3) & \text{NIL}|||T = T \\ (4) & \Sigma^*|||T = \Sigma^*. \end{array}$$

Proof. Trivial ∎

Definition 2.5.5: For any subsets P, Q of $\Sigma^* \times p\Sigma$, let $P|||Q$ be the set

$$P|||Q = \{(u,X) \mid \exists s,t.(s,X) \in P \,\&\, (t,X) \in Q \,\&\, u \in \text{merge}(s,t)\}.$$

Theorem 2.5.6: If P and Q are processes, so is $P|||Q$. This process is characterised uniquely by the following conditions:

(1) $\quad \text{traces}(P|||Q) = \text{traces}(P)|||\text{traces}(Q)$

(2) $\quad \text{refusals}(P|||Q) = \text{refusals}(P) \cap \text{refusals}(Q)$

(3) $\quad (P|||Q)\,\underline{\text{after}}\,u = \bigcup\{(P\,\underline{\text{after}}\,s)|||(Q\,\underline{\text{after}}\,t) \mid u \in \text{merge}(s,t)\}.$

Proof. Let P and Q be processes. First we see immediately from the above definition that condition (P2) holds:

$$\begin{aligned}(u,X) \in P|||Q \,\&\, Y \subseteq X &\Rightarrow \exists s,t.(s,X) \in P \,\&\, (t,X) \in Q \,\&\, u \in \text{merge}(s,t) \\ &\quad \&\, Y \subseteq X \\ &\Rightarrow u \in \text{merge}(s,t) \,\&\, (s,Y) \in P \,\&\, (t,Y) \in Q \\ &\Rightarrow (u,Y) \in P|||Q.\end{aligned}$$

In particular this allows us to deduce that the traces of $P|||Q$ are indeed obtained by merging:

$$\begin{aligned}\text{traces}(P|||Q) &= \{u \mid \exists X.(u,X) \in P|||Q\} \\ &= \{u \mid (u,\emptyset) \in P|||Q\} \\ &= \{u \in \text{merge}(s,t) \mid (s,\emptyset) \in P \,\&\, (t,\emptyset) \in Q\} \\ &= \bigcup\{\text{merge}(s,t) \mid s \in \text{traces}(P) \,\&\, t \in \text{traces}(Q)\} \\ &= \text{traces}(P)|||\text{traces}(Q).\end{aligned}$$

But traces(P) and traces(Q) are trees, since P and Q are processes. It is clear from Lemma 2.5.2 that merging two trees produces another tree; so we have established condition (P1), and also (1) above.

The refusals clearly satisfy (2), by Definition 2.5.5.

Finally, suppose $(u,X) \in P|||Q$, but $u\langle c\rangle \notin \text{traces}(P|||Q.)$ We must prove that $(u, X \cup \{c\}) \in P|||Q$. Let s,t be traces such that

$$(s,X) \in P, \quad (t,X) \in Q, \quad u \in \text{merge}(s,t).$$

Then $s\langle c\rangle$ cannot be a trace of P, because $u\langle c\rangle$ is not a trace of $P|||Q$. Similarly, $t\langle c\rangle$ cannot be a trace of Q. Hence, we can use $(P3)$ for the processes P and Q to deduce that

$$(s, X \cup \{c\}) \in P, \quad (t, X \cup \{c\}) \in Q, \quad u \in \text{merge}(s,t).$$

But this gives us the fact that $(u, X \cup \{c\}) \in P|||Q$, as required. That completes the proof that $P|||Q$ is a process. ∎

Some elementary properties of interleaving are stated next. The operation is symmetric, associative, and has STOP as a unit and RUN as a zero.

Theorem 2.5.7:

$$\begin{array}{ll} (1) & P|||Q = Q|||P \\ (2) & (P|||Q)|||R = P|||(Q|||R) \\ (3) & \text{STOP}|||Q = Q \\ (4) & \text{RUN}|||Q = \text{RUN}. \end{array}$$

Proof. Trivial. ∎

The next results show how interleaving of two guarded processes behaves.

Theorem 2.5.8: Let $P_B = \Box_{b\in B}(b \to P_b)$ and $Q_C = \Box_{c\in C}(c \to Q_c)$. Then interleaving these processes gives

$$P_B|||Q_C = \Box_{b\in B}(b \to (P_b|||Q_C)) \Box \Box_{c\in C}(c \to (P_B|||Q_c)).$$

Proof. Trivial. ∎

Corollary: For all processes P and Q,

$$(a \to P)|||(b \to Q) = (a \to P|||(b \to Q)) \Box (b \to (a \to P)|||Q).$$

Next we establish the important properties of monotonicity, distributivity and continuity.

Theorem 2.5.9: $\lambda P, Q.P|||Q$ is monotone, continuous and distributive.

Proof. It is enough to consider one variable at a time, because $|||$ is symmetric. Since distributivity implies monotonicity, we consider this case first.

Distributivity. Let $\mathcal{D}$ be a non-empty set of processes, let P be any process and let Q be the supremum of $\mathcal{D}$:

$$Q = \bigcup \mathcal{D}.$$

We want to show that $P|||Q$ is the supremum of the collection of interleaved processes:

$$P|||Q = \bigcup\{ P|||R \mid R \in \mathcal{D} \}.$$

This is straightforward:

$$\begin{aligned} (u, X) \in P|||Q &\Leftrightarrow \exists s, t.(s, X) \in P \,\&\, (t, X) \in Q \,\&\, u \in \text{merge}(s, t) \\ &\Leftrightarrow \exists R \in \mathcal{D}.\exists s, t.(s, X) \in P, (t, X) \in R, u \in \text{merge}(s, t) \\ &\Leftrightarrow \exists R \in \mathcal{D}.(u, X) \in P|||R \\ &\Leftrightarrow (u, X) \in \bigcup\{ P|||R \mid R \in \mathcal{D} \} \end{aligned}$$

as required. So interleaving is a distributive operation.

Continuity. For continuity of interleaving, let $\{ Q_n \mid n \geq 0 \}$ be a chain of processes with limit Q, and let P be any process. We know by monotonicity

that $\{ P|||Q_n \mid n \geq 0 \}$ is a chain; we want to prove that the limit of this chain is $P|||Q$. As usual, monotonicity provides one half of the result:

$$\bigsqcap_n P|||Q_n \sqsubseteq P|||Q.$$

For the reverse inequality, suppose $(u, X) \in \bigsqcap_n P|||Q_n$. This means that for each $n \geq 0$ there are traces s_n, t_n such that

$$(s_n, X) \in P, \quad (t_n, X) \in Q_n, \quad u \in \text{merge}(s_n, t_n).$$

But there are only finitely many pairs of traces which merge to u, since u has finite length. This means that some pair must appear infinitely often in the list $\{(s_n, t_n) \mid n \geq 0\}$. Let (s, t) be such a pair. Then $(t, X) \in Q_n$, for infinitely many values of n, which gives $(t, X) \in Q$, by the chain property. Since we already have $(s, X) \in P$ and $u \in \text{merge}(s, t)$, this completes the proof. ∎

Theorem 2.5.10:

$$\det(S)|||\det(T) \sqsubseteq \det(S|||T).$$

Proof. By Theorem 2.5.6, these two processes have the same traces:

$$\text{traces}(\det(S)|||\det(T)) = S|||T.$$

By Lemma 1.2.3,

$$P \sqsubseteq \det(\text{traces}(P),$$

for all processes P. The result follows on putting $\dot{P} = \det(S)|||\det(T)$. ∎

This inequality cannot in general be strengthened, as $|||$ is not a deterministic operation. Some examples will illustrate this point.

Examples.

Example 1. For the simple processes $(a \rightarrow \text{STOP})$ and $(b \rightarrow \text{STOP})$, we have:

(i) $(a \rightarrow \text{STOP})|||(a \rightarrow \text{STOP}) = (a \rightarrow (a \rightarrow \text{STOP}))$
(ii) $(a \rightarrow \text{STOP})|||(b \rightarrow \text{STOP}) = (a \rightarrow (b \rightarrow \text{STOP})) \,\Box\, (b \rightarrow (a \rightarrow \text{STOP}))$.

Thus the two events can occur in either order, and the interleaved process stops only when both components can continue no further.

Example 2. Let B and C be sets of events. Note that interleaving a trace from B^* with a trace from C^* always produces a trace in $(B \cup C)^*$, and that every trace in $(B \cup C)^*$ can be produced in at least one such way. In other words, $B^*|||C^* = (B \cup C)^*$. Let P and Q be the processes:

$$P = \det(B^*)|||\det(C^*)$$
$$Q = \det(B^*|||C^*).$$

We now show that $P = Q$. First consider the traces.

$$\begin{aligned}\text{traces}(P) &= \text{traces}(\det(B^*)|||\det(C^*)) \\ &= \text{traces}(\det(B^*))|||\text{traces}(\det(C^*)) \\ &= B^*|||C^* \\ &= \text{traces}(\det(B^*|||C^*)) \\ &= \text{traces}(Q).\end{aligned}$$

Next the refusals.

$$\begin{aligned}\text{refusals}(P) &= \text{refusals}(\det(B^*)) \cap \text{refusals}(\det(C^*)) \\ &= \{X \mid X \cap B = \emptyset \,\&\, X \cap C = \emptyset\} \\ &= \{X \mid X \cap (B \cup C) = \emptyset\} \\ &= \text{refusals}(\det(B \cup C)^*) \\ &= \text{refusals}(Q).\end{aligned}$$

Finally, since for all $s \in B^*, t \in C^*$,

$$B^* \underline{\text{after}}\, s = B^*$$

$$C^* \underline{\text{after}}\, t = C^*$$

we have, for all $u \in (B \cup C)^*$,

$$\begin{aligned}P \underline{\text{after}}\, u &= \bigcup\{\det(B^*) \underline{\text{after}}\, s|||\det(C^*) \underline{\text{after}}\, t \mid u \in \text{merge}(s,t)\} \\ &= \det(B^*)|||\det(C^*) \\ &= P.\end{aligned}$$

Similarly, $Q \underline{\text{after}}\, u = Q$. It follows that $P = Q$. ∎

A derived operation.

To end this section on interleaving let us introduce a useful derived operation. Let B be a set of events and P be a process which cannot ever perform any of the events in B. To be precise, we are assuming that the alphabet of P is disjoint from B :

$$\text{traces}(P) \subseteq \overline{B}^*.$$

In the composition $P|||\det(B^*)$, therefore, the P component performs the events from $\overline{B}$ and *ignores* events in B. We will refer to this form of composition as P *ignoring* B, and use a special notation $P\rceil B$.

Definition 2.5.11: Process P *ignores* a set B of events if $\text{traces}(P) \subseteq \overline{B}^*$.

Definition 2.5.12: For any process P and any set of events B,

$$P \rceil B = P ||| \det(B^*).$$

When P ignores B, this combination performs a sequence of events s when the P component performs $s \lceil \overline{B}$, all other events in s occurring for the other component. We will normally restrict use of this operation to the case when P does ignore B.

As immediate corollaries of earlier results, we have

$$\begin{aligned}(P \rceil B) \rceil C &= (P ||| \det(B^*)) ||| \det(C^*) \\ &= P ||| (\det(B^*) ||| \det(C^*)) \\ &= P ||| \det((B \cup C)^*) \\ &= P \rceil (B \cup C).\end{aligned}$$

Indeed, this equality holds even when P does not ignore $B \cup C$.

By its definition, $P \lceil B$ ignores $\overline{B}$. Finally, in the special case when $B = \emptyset$, we have $P \rceil \emptyset = P$, since $\det(\emptyset^*) = \text{STOP}$.

6. A derived parallel combinator

The previous two sections concerned extreme forms of parallel composition. We may think of $P \| Q$ as a purely *synchronous* composition, in which each action requires the simultaneous participation of the two component processes and a set of events can be refused if P can refuse part of the set and Q can refuse the rest. Similarly, $P|||Q$ is like an *asynchronous* composition in which each action requires participation by just one of the components, and a set is refused only if both components can refuse it.

Suppose that P and Q are processes such that

$$\text{traces}(P) \subseteq B^*$$
$$\text{traces}(Q) \subseteq C^*.$$

In other words, B and C are alphabets for P and Q respectively. Suppose we wish P and Q to operate concurrently, but to *synchronise* on events common to their alphabets. This would typically be the case when events in both alphabets represent communications between P and Q, in a system where communication is achieved by "handshakes," so that a communication can occur only when the sender and receiver are ready to perform it together. So events in $B \cap C$ are to require participation of both processes, while events in $B-C$ require only P and events in $C-B$ need Q alone. This is a *mixed* form of parallel composition, rather like the well known "shuffle" operator. Roughly speaking, we want a form of composition that is more constraining than the pure interleaving operation but not as restricting as pure parallel composition. We can defined a suitable derived operation in terms of the two pure operations. Before giving the formal definition, let us consider what properties the constructed process, which we will denote $P_B \|_C Q$, should have. We are certainly requiring that the traces of this process are built up from traces of P and Q, so that

$$\text{traces}(P_B \|_C Q) \subseteq (B \cup C)^*.$$

Moreover, such a trace should yield a trace of P when viewed through the alphabet of P, and a trace of Q when viewed through the alphabet of Q. This requirement is stated thus:

$$s \in \text{traces}(P_B \|_C Q) \Rightarrow s\lceil B \in \text{traces}(P) \;\&\; s\lceil C \in \text{traces}(Q).$$

As for refusals, an event in B can be refused if P chooses to refuse it, and an event in C can be refused if Q refuses it. This has the consequence that an event belonging to $B \cap C$ can be ruled out by either of the two component

processes, since it can occur if and only if they both allow it. Events outside $B \cup C$ are always impossible and can always be included in a refusal set:

$$\begin{aligned}\text{refusals}(P_B \|_C Q) = \{ X \cup Y \cup Z \mid X \subseteq B, Y \subseteq C, Z \subseteq \overline{B \cup C}, \\ X \in \text{refusals}(P) \,\&\, Y \in \text{refusals}(Q) \}.\end{aligned}$$

Finally, once $P_B \|_C Q$ has performed a sequence of events s it must be the case that the P component has engaged so far in the sequence $s\lceil B$, while the Q component has performed the sequence $s\lceil C$; the future behaviour should therefore be that of P after $(s\lceil B)$ running in parallel with Q after $(s\lceil C)$,

$$(P_B \|_C Q) \text{ after } s = (P \text{ after } s\lceil B)_B \|_C (Q \text{ after } s\lceil C).$$

We will verify that the following definition is satisfactory.

Definition 2.6.1: For any sets B, C of events, and processes P, Q

$$\begin{aligned}P_B \|_C Q = \{ (s, X \cup Y \cup Z) \mid X \subseteq B, Y \subseteq C, Z \subseteq \overline{B \cup C}, \\ s \in (B \cup C)^* \;\&\; (s\lceil B, X) \in P \,\&\, (s\lceil C, Y) \in Q \}.\end{aligned}$$

Theorem 2.6.2: For all processes P, Q and all sets of events B, C $P_B \|_C Q$ is a process.

Proof. It is a straightforward task to verify conditions (P0)–(P3). We give details for the most interesting of these, (P3):

Suppose

$$(s, W) \in P_B \|_C Q \;\&\; sa \notin \text{traces}(P_B \|_C Q).$$

We must show that $(s, W \cup \{ a \}) \in P_B \|_C Q$.

By assumption there are sets X, Y, Z such that

$$W = X \cup Y \cup Z,\ X \subseteq B,\ Y \subseteq C,\ Z \subseteq \overline{(B \cup C)},\ (s\lceil B, X) \in P,\ (s\lceil C, Y) \in Q.$$

There are four possible reasons why sa fails to be a valid trace, by inspection of Definition 2.6.1.

(i) If $a \notin (B \cup C)$, then we can simply include it in Z.

(ii) If $a \in B - C$, then $sa\lceil B = (s\lceil B)a$ and $sa\lceil C = s\lceil C$. Thus, $(s\lceil B)a$ cannot be a trace of P, so a is impossible for P after $s\lceil B$ and we can include a in X.

(iii) A similar argument shows that if $a \in C - B$ we can include a in Y.

(iv) Finally, if $a \in B \cap C$ we have $sa\lceil B = (s\lceil B)a$ and $sa\lceil C = (s\lceil C)a$. Since we are assuming that sa is not a trace of the compound process, one of the clauses in Definition 2.6.1 fails to hold: we must have

$$(s\lceil B)a \notin \text{traces}(P) \vee (s\lceil C)a \notin \text{traces}(Q).$$

Thus we can include a in either X or Y.

In each case we have shown that $(s, W \cup \{a\})$ is a failure, as required. ∎

Theorem 2.6.3: Provided P ignores $\overline{B}$ and Q ignores $\overline{C}$, the above definition can be characterised as:

$$P_B \|_C Q = (P\rceil\overline{B} \| Q\rceil\overline{C})\lceil(B \cup C).$$

Proof. Introduce the abbreviation $R = (P\rceil\overline{B} \| Q\rceil\overline{C})$, so that we are trying to prove that

$$P_B \|_C Q = R\lceil(B \cup C).$$

By definition of $\lceil$ and $\|$, and by assumption that $\text{traces}(P) \subseteq B^*$,

$$\begin{aligned} P\rceil B &= P \| \det(\overline{B}^*) \\ &= \{(s,X) \mid (s\lceil B, X) \in P \,\&\, (s\lceil\overline{B}, X) \in \det(\overline{B}^*)\} \\ &= \{(s,X) \mid (s,X) \in P \,\&\, X \subseteq B\}. \end{aligned}$$

By a similar argument,

$$Q\rceil C = \{(s,Y) \mid (s\lceil C, Y) \in Q \,\&\, Y \subseteq C\}.$$

Hence,

$$\begin{aligned} R = \{(s, X \cup Y) \mid (s\lceil B, X) \in P \,\&\, (s\lceil C, Y) \in Q \\ \&\, X \subseteq B \,\&\, Y \subseteq C\}. \end{aligned}$$

Now restricting to $B \cup C$ we get

$$\begin{aligned} R\lceil(B \cup C) &= R \| \det((B \cup C)^*) \\ &= \{(s, X \cup Y \cup Z) \mid (s\lceil B, X) \in P, (s\lceil C, Y) \in Q, \\ &\qquad X \subseteq B, Y \subseteq C, s \in (B \cup C)^*, Z \subseteq \overline{B \cup C}\}. \end{aligned}$$

The result follows from Definition 2.6.1. That completes the proof. ∎

It is clear from Definition 2.6.1 that, the traces and refusals of the composition $P_B \|_C Q$ do indeed have the properties suggested earlier. It only remains to verify that once an event has occurred the composition continues to behave like a parallel product.

Theorem 2.6.4: For all processes P, Q and all sets of events B, C, whenever u is a valid trace we have

$$(P_B \|_C Q) \underline{\text{after}}\, u = (P \underline{\text{after}}\, (u\lceil B))_B \|_C (Q \underline{\text{after}}\, (u\lceil C)).$$

Proof. We calculate as follows, using Definition 2.6.1:

$$\begin{aligned}(P_B \|_C Q) \underline{\text{after}}\, u &= \{(v,Z) \mid (uv,Z) \in P_B \|_C Q\} \\ &= \{(v, X \cup Y \cup Z) \mid X \subseteq B,\ Y \subseteq C,\ Z \subseteq \overline{B \cup C}, \\ &\qquad ((uv)\lceil B, X) \in P \,\&\, ((uv)\lceil C, Y) \in Q\} \\ &= \{(v, X \cup Y \cup Z) \mid X \subseteq B,\ Y \subseteq C,\ Z \subseteq \overline{B \cup C}, \\ &\qquad ((u\lceil B)(v\lceil B), X) \in P \ \&\ ((u\lceil C)(v\lceil C), Y) \in Q\} \\ &= \{(v, X \cup Y \cup Z) \mid X \subseteq B,\ Y \subseteq C,\ Z \subseteq \overline{B \cup C}, \\ &\qquad (v\lceil B, X) \in P \,\underline{\text{after}}\, (u\lceil B) \ \&\ (v\lceil C, Y) \in Q \,\underline{\text{after}}\, (u\lceil C)\} \\ &= (P \,\underline{\text{after}}\, (u\lceil B))_B \|_C (Q \,\underline{\text{after}}\, (u\lceil C)).\end{aligned}$$

∎

The mixed parallel combinator has been defined as a composition of pure parallel product and interleaving. Since both of these operations are monotone, continuous and distributive, it is clear that the mixed combinator also has these properties. We state this fact here for future reference.

Theorem 2.6.5: The mixed parallel combinator ${}_B\|_C$ is monotone, continuous and distributive:

$$\begin{aligned}&(1) && Q \sqsubseteq R \Rightarrow P_B \|_C Q \sqsubseteq P_B \|_C R \\ &(2) && P_B \|_C (\textstyle\bigsqcap_n Q_n) = \bigsqcap_n (P_B \|_C Q_n) \\ &(3) && P_B \|_C (Q \sqcap R) = (P_B \|_C Q) \sqcap (P_B \|_C R).\end{aligned}$$

Proof. By the corresponding properties of parallel composition and interleaving. ∎

This operator also enjoys an associativity property, as expressed in the following theorem.

Theorem 2.6.6:

$$P_A \|_{B \cup C} (Q_B \|_C R) = (P_A \|_B Q)_{A \cup B} \|_C R.$$

Proof. It suffices to show that each side of the equation can be rewritten as follows:

$$\begin{aligned}\{(s, X \cup Y \cup Z \cup W) \mid &X \subseteq A, Y \subseteq B, Z \subseteq C, W \subseteq \overline{A \cup B \cup C}, \\ &s \in (A \cup B \cup C)^*, (s\lceil A, X) \in P, (s\lceil B, Y) \in Q, (s\lceil C, Z) \in R\}.\end{aligned}$$

Indeed, since this expression is symmetric in P, Q, R and A, B, C, we need only reduce one of the terms to this form. We omit the details, as the calculation is straightforward. In the case when the alphabets of P, Q, R are contained in A, B, C respectively, we could prove instead that the process each side of the equation is expressible as

$$(P\lceil \overline{A} \| Q\rceil \overline{B} \| R\rceil \overline{C})\lceil (A \cup B \cup C).$$

We omit details. ∎

Examples.

Example 1.

Let P and Q be the recursively defined processes

$$P = (a \to c \to P)$$
$$Q = (b \to c \to Q)$$

Equivalently, $P = \det((ac)^*)$ and $Q = \det((bc)^*)$. The alphabets of these processes are $A = \{a, c\}$ and $B = \{b, c\}$ respectively. If we form the parallel product $P_A \|_B Q$ the only synchronizing event is c. Initially, neither component process is ready to synchronize, but each is willing to progress autonomously. Once they have each performed their first step, the next action must be a synchronization. After that we are again in the initial configuration. More formally,

$$P_A \|_B Q = (a \to b \to c \to P_A \|_B Q) \,\Box\, (b \to a \to c \to P_A \|_B Q).$$

The alphabet of this product process is $\{a, b, c\}$.

Example 2.

Let $P = (a \to b \to \text{STOP})$ and $Q = (b \to a \to \text{STOP})$. Both processes have alphabet $A = \{a, b\}$. If we form the parallel product in which both processes are required to synchronize on their entire alphabet, no action can occur; the initials of the two processes are disjoint. In this case, we have

$$P_A \|_A Q = \text{STOP}.$$

7. Hiding

The events performed by a process can be thought of as actions visible to and possibly requiring participation by the environment of the process. Events represent externally visible actions. It may be the case that, when P is a composite process built from a set of interacting component processes, some of the events are used only for communication between the components. In such a situation, we might feel it unrealistic to assume that all details of these essentially *internal* actions be observable by the environment of the process. We now introduce an operation which *conceals* all occurrences of a particular event from the environment of a process. The process formed by *hiding* all occurrences of b in P will be denoted $P\backslash b$, and the trace obtained from s by deleting all occurrences of b is denoted $s\backslash b$. Informally, the behaviour of $P\backslash b$ can be specified as follows. For each trace s of P, it should be possible for $P\backslash b$ to perform the trace $s\backslash b$. But whenever P had the ability to perform the *internal* action after doing the sequence s, $P\backslash b$ can decide autonomously whether to do the hidden action, once it has performed $s\backslash b$. Of course, the *ability* of P to perform the hidden event depends on P's choice of refusal set. Thus, if P can refuse X after s and $b \in X$, we will require that $P\backslash b$ be able to refuse X after $s\backslash b$. These two conditions are almost enough to define a process. However, there is one further possibility to consider. In the case where P can engage in an arbitrarily long sequence of internal actions at some stage, so that there is a trace s with $sb^n \in \text{traces}(P)$ for all n, then the process $P\backslash b$ may never interact with its environment. We will call this phenomenon *infinite internal chatter,* or *divergence.* There are two fairly reasonable interpretations of this behaviour in terms of failure sets. One is to identify such behaviour (or lack of behaviour) with deadlock, and to allow $P\backslash b$ to behave like STOP in such circumstances. The other identifies this behaviour with CHAOS, since the environment of a process engaging in infinite chatter cannot rule out the possibility that the process may, at some future point, stop chattering and start performing visible events again. We give, correspondingly, two alternative definitions of hiding; first let us consider the version in which infinite internal actions lead to deadlock. Afterwards we will discuss the "chaotic" version. In Chapter 5 we will extend the failures model to obtain a perhaps more satisfying treatment of infinite chatter.

Definition 2.7.1: The *hiding* function $\backslash b$ on traces is defined by induction on length:

$$\begin{aligned} \langle\rangle\backslash b &= \langle\rangle \\ (s\langle x\rangle)\backslash b &= (s\backslash b)\langle x\rangle \quad (x \neq b), \\ (s\langle x\rangle)\backslash b &= s\backslash b \qquad\quad (x = b). \end{aligned}$$

It is easy to see from the definition that the following identities hold:

Lemma 2.7.2:

$$\begin{array}{ll} \text{(i)} & (s\backslash b)\backslash b = s\backslash b \\ \text{(ii)} & (s\backslash b)\backslash c = (s\backslash c)\backslash b \\ \text{(iii)} & (st)\backslash b = (s\backslash b)(t\backslash b). \end{array}$$

Similarly, any prefix u of $s\backslash b$ must be obtained by hiding from a prefix of u,

$$u \leq s\backslash b \Rightarrow \exists t \leq s.u = t\backslash b.$$

Finally, if b does not occur in s then $s\backslash b = s$.

Definition 2.7.3: For a set S of traces we define

$$S\backslash b = \{\, s\backslash b \mid s \in S \,\}.$$

It is clear, from the above properties of hiding on traces, that when S is a tree so is $S\backslash b$.

Definition 2.7.4: For a process P we define

$$\begin{aligned} P\backslash b = &\{\, (s\backslash b, X) \mid (s, X \cup \{\, b \,\}) \in P \,\} \\ &\cup \{\, (s\backslash b, X) \mid X \in p\Sigma \;\&\; \forall n. sb^n \in \text{traces}(P) \,\}. \end{aligned}$$

This version of hiding represents infinite chatter by STOP, thus identifying deadlock with infinite internal activity.

Theorem 2.7.5: If P is a process so is $P\backslash b$, and

$$\begin{array}{ll} (1) & \text{traces}(P\backslash b) = \text{traces}(P)\backslash b \\ (2) & \text{refusals}(P\backslash b) = \{\, X \mid X \cup \{\, b \,\} \in \text{refusals}(P) \,\} \\ & \qquad\qquad \cup \{\, X \mid X \in p\Sigma \;\&\; \forall n. b^n \in \text{traces}(P) \,\} \\ (3) & (P\backslash b)\,\underline{\text{after}}\, u = \bigcup \{\, (P\,\underline{\text{after}}\, s)\backslash b \mid s\backslash b = u \,\}. \end{array}$$

Proof. The fact that hiding maps processes to processes will follow from (1)–(3), which can easily be used to establish the required properties (P0)–(P3). First let us establish (1). It is clear from the definition that

$$\text{traces}(P\backslash b) \subseteq \text{traces}(P)\backslash b.$$

If $s \in \text{traces}(P)$ and also $sb^n \in \text{traces}(P)$ for all $n \geq 0$, then again by definition we have $s\backslash b \in \text{traces}(P\backslash b)$.

If $s \in \text{traces}(P)$ but there is an integer n such that $sb^n \notin \text{traces}(P)$, then (choosing the smallest such n) we can use property (P4) of the process P to get $(sb^n, \{\, b \,\}) \in P$. But this gives $(s\backslash b, \emptyset) \in P\backslash b$, so that again $s\backslash b \in \text{traces}(P\backslash b)$.

This shows that

$$\text{traces}(P)\backslash b \subseteq \text{traces}(P\backslash b)$$

and completes the proof of (1).

The identity (2) is obvious, from Definition 2.7.4.

Finally, for (3), let u be a trace of $P\backslash b$. Let (v, X) be a failure of $(P\backslash b)$ after u, so that (uv, X) is a failure of $P\backslash b$. There are two possibilities:

$$\begin{array}{ll} \text{(i)} & \exists s.\ s\backslash b = uv \ \&\ (s, X \cup \{ b \}) \in P \\ \text{(ii)} & \exists s.\ s\backslash b = uv \ \&\ \forall n. sb^n \in \text{traces}(P). \end{array}$$

In both cases s can be written in the form $s = tw$, where

$$t\backslash b = u \ \&\ w\backslash b = v.$$

It is then clear that we have, respectively,

$$\begin{array}{ll} \text{(i)} & \exists t, w.\ t\backslash b = u \ \&\ w\backslash b = v \ \&\ (w, X \cup \{ b \}) \in P \text{ after } t, \\ \text{(ii)} & \exists t, w.\ t\backslash b = u \ \&\ w\backslash b = v \ \&\ \forall n. (wb^n, \emptyset) \in P \text{ after } t. \end{array}$$

Thus, we have one half of the condition:

$$P\backslash b \text{ after } u \subseteq \bigcup \{ (P \text{ after } t)\backslash b \mid t\backslash b = u \}.$$

The argument can be reversed to show the converse, from which the result follows. ∎

Next we show what the effect of hiding is on a guarded process. We will see that nondeterminism is introduced when one of the guards is hidden; in this case, hiding a guard removes control over the passing of that guard from the environment, and allows the process to decide autonomously whether to pass it.

Lemma 2.7.6: Let P_B be a guarded process:

$$P_B = \Box_{b \in B}(b \to P_b).$$

Then

$$\begin{array}{lll} P_B\backslash a = \Box_{b \in B}(b \to P_b\backslash a) & & \text{if } a \notin B, \\ \quad = P_a\backslash a \sqcap (P_a\backslash a \,\Box\, P_C\backslash a) & & \text{if } B = C \cup \{ a \}, a \notin C. \end{array}$$

Proof. Elementary. ∎

Corollary:

$$\begin{array}{lll} \text{(i)} & \text{STOP}\backslash b = \text{STOP} & \\ \text{(ii)} & (a \to P)\backslash b = (a \to P\backslash b) & (a \neq b) \\ \text{(iii)} & (b \to P)\backslash b = P\backslash b. & \end{array}$$

∎

Some examples will help to illustrate the properties of hiding.

Examples. Throughout these examples we assume that a and b are distinct events.

Example 1.

$$(a \to (b \to \text{STOP}))\backslash a = (b \to \text{STOP}).$$

Example 2.

$$(a \to (b \to \text{STOP}))\backslash b = (a \to \text{STOP}).$$

Example 3.

$$((a \to \text{STOP}) \,\Box\, (b \to \text{STOP}))\backslash b = (a \to \text{STOP}) \sqcap \text{STOP}.$$

In the third example, the original process was initially able to perform either the hidden event or the visible event a, and the environment of the process could control this choice. Hiding b allows the process to decide autonomously to perform it, a decision which would remove the possibility of performing a. Thus the resulting process has an initial non-deterministic choice between the hidden and the visible action. In the other two cases no such choice was necessary, because the event to be hidden was never possible at the same time as a visible event.

Example 4. Define a chain of processes P_n as follows:

$$P_0 = \text{chaos}(a^*b)$$
$$P_{n+1} = (a \to P_n)$$

The limit of this chain is $P = \bigcap_n P_n = \text{det}(a^*)$. Since each P_n can perform arbitrarily many hidden actions, we have

$$P_0\backslash a = \text{STOP} \sqcap (b \to \text{STOP})$$
$$P_{n+1}\backslash a = P_n\backslash a.$$

But the limit process P cannot perform a visible action, and

$$P\backslash a = \text{det}(a^*)\backslash a = \text{STOP}.$$

This shows that this version of hiding is not a continuous operation.

Example 5. Define the processes Q_n by:

$$Q_0 = (b \to \text{STOP})$$
$$Q_{n+1} = (a \to Q_n)$$

Let $Q = \bigcup_n Q_n$ be the greatest lower bound of these processes. Then a^n is a possible trace of Q, for each n. It is clear that

$$Q_n = (a^n b \rightarrow \text{STOP})$$
$$Q_n \backslash a = (b \rightarrow \text{STOP}).$$

But

$$Q \backslash a = \text{STOP} \sqcap (b \rightarrow \text{STOP}).$$

Here is an example where

$$(\bigcup_n Q_n) \backslash a \sqsubseteq \bigcup_n (Q_n \backslash a).$$

Hiding is not, therefore, generally distributive; it is, however distributive over *finite* unions.

Theorem 2.7.7: Hiding is finitely distributive and monotone.

Proof. Distributivity implies monotonicity, so we need only prove that hiding distributes over finite unions. It is enough to show that

$$(P \sqcap Q) \backslash b = P \backslash b \sqcap Q \backslash b,$$

for all processes P and Q. Let P and Q be processes with trace sets S and T respectively, so that the trace set of $P \sqcap Q$ is $S \cup T$. Since $(S \cup T) \backslash b = S \backslash b \cup T \backslash b$, we see that $(P \sqcap Q) \backslash b$ and $P \backslash b \sqcap Q \backslash b$ have the same traces. According to the definition of hiding,

$$\begin{array}{rll} (u, X) \in (P \sqcap Q) \backslash b \Leftrightarrow & \text{either} & u = w \backslash b \,\&\, (w, X \cup \{b\}) \in P \sqcap Q \\ & \text{or} & u = w \backslash b \,\&\, wb^n \in S \cup T \text{ for all } n. \end{array}$$

But if for each n we have $wb^n \in S \cup T$, then one of S and T must have infinitely many traces of the form wb^n. By prefix-closure, this means that one of S and T contains all of these traces: either every wb^n is a trace of P or every wb^n is a trace of Q. Hence, rewriting the above,

$$\begin{array}{rll} (u, X) \in (P \sqcap Q) \backslash b \Leftrightarrow & \text{either} & u = w \backslash b \,\&\, (w, X \cup \{b\}) \in P \\ & \text{or} & u = w \backslash b \,\&\, (w, X \cup \{b\}) \in Q \\ & \text{or} & u = w \backslash b \,\&\, wb^n \in \text{traces}(P) \text{ for all } n \\ & \text{or} & u = w \backslash b \,\&\, wb^n \in \text{traces}(Q) \text{ for all } n. \end{array}$$

But this is clearly equivalent to

$$\begin{array}{rl} (u, X) \in (P \sqcap Q) \backslash b \Leftrightarrow & (u, X) \in P \backslash b \text{ or } (u, X) \in Q \backslash b \\ \Leftrightarrow & (u, X) \in (P \backslash b) \sqcap (Q \backslash b). \end{array}$$

That completes the proof. ∎

The above examples also showed that hiding is not in general a deterministic operation. Hiding an event allows the process to decide autonomously whether or not to perform it, without letting the environment see it. In cases where the process was able to perform either a visible action or the hidden action, hiding could introduce non-determinism. Nevertheless, there is a special case when hiding does preserve determinism: when the action to be hidden is *unavoidable* or *exclusive* in the sense that, whenever it is possible it is the *only* possible action; there is never any other visible event which the process could perform instead of it, and the possibility of performing this event excludes all other events.

Definition 2.7.8: Let T be a tree and b be an event. Then b is *unavoidable* in T if, for all $t \in T, c \in \Sigma$,

$$t\langle b\rangle \in T \;\&\; t\langle c\rangle \in T \Rightarrow b = c.$$

Lemma 2.7.9: If b is unavoidable in T then for each trace $u \in T\backslash b$ there is a unique *minimal* trace $t \in T$ such that $t\backslash b = u$.

Proof. Let $u \in T\backslash b$. If $u = \langle\rangle$ then $t = \langle\rangle$ is the required trace. In the general case, let $u = \langle c_1, \ldots, c_n\rangle$ with $c_i \neq b$ for $i = 1 \ldots n$. By hypothesis there is a trace $w \in T$ such that $w\backslash b = u$. This trace can be written in the form

$$w = w_1\langle c_1\rangle \ldots w_{n-1}\langle c_n\rangle w_n,$$

for some traces $w_i \in b^*$. The trace

$$t = w_1\langle c_1\rangle \ldots w_{n-1}\langle c_n\rangle$$

is also in T, by prefix-closure. By definition, $t\backslash b = u$ and t is minimal with respect to this property. Finally, any other trace v in T satisfying this property must have the form

$$v = v_1\langle c_1\rangle \ldots v_{n-1}\langle c_n\rangle v_n$$

for some $v_i \in b^*$. It is easy to prove, from the definition of unavoidability, that

$$v_i = w_i,$$

for $i = 1 \ldots n-1$. That shows that the minimal trace t is unique. ∎

Theorem 2.7.10: If b is unavoidable in T then $\det(T)\backslash b = \det(T\backslash b)$.

Proof. By Lemma 1.2.3, and the fact that traces$(P\backslash b)$ = traces$(P)\backslash b$, we have

$$\det(T)\backslash b \sqsubseteq \det(T\backslash b).$$

For the converse, we need to establish that every failure of $\det(T)\backslash b$ is also a failure of $\det(T\backslash b)$. Let $(u, X) \in \det(T)\backslash b$. By Lemma 2.7.9, there is a unique minimal trace $t \in T$ such that $t\backslash b = u$. Moreover, any trace $s \in T$ such that $s\backslash b = u$ must have the form tb^n for some integer n. There are two cases to consider:

(a) $tb^n \in T$ for all $n \geq 0$,
(b) $tb^n \in T, tb^{n+1} \notin T$ for some $n \geq 0$.

In case(a), u must be a maximal trace of $T\backslash b$, because if there were an event c such that $u\langle c\rangle \in T\backslash b$ there would be a trace $w \in T$ with $w\backslash b = u\langle c\rangle$. There would then be a prefix $v < w$ with $v\backslash b = u$ and $v\langle c\rangle \in T$. But v must have the form tb^n for some n, and by hypothesis we also have $tb^{n+1} = v\langle b\rangle \in T$. This contradicts the assumption that b is unavoidable. So u is maximal in $T\backslash b$, and no event is possible for $\det(T\backslash b)$ after u. By (P3), therefore, $\det(T\backslash b)$ can refuse any set X after u. Thus, $(u, X) \in \det(T\backslash b)$ as required.

In case(b), let $s = tb^n$, so that $s \in T$ but $s\langle b\rangle \notin T$. In this case, the traces of T which produce u on hiding b are $t, t\langle b\rangle, \ldots, tb^n$. By definition of $\det(T)$, we have

$$(tb^i, \{\, b \,\}) \notin \det(T), \quad (i = 0 \ldots n-1)$$
$$(tb^n, \{\, b \,\}) \in \det(T).$$

It follows that the only trace contributing to the refusals of $\det(T)\backslash b$ after u is s. Since we are assuming that

$$(u, X) \in \det(T)\backslash b,$$

we must have

$$(s, X \cup \{\, b \,\}) \in \det(T).$$

This means that

$$\forall c \in X. \quad s\langle c\rangle \notin T.$$

We want to show that

$$\forall c \in X. \quad u\langle c\rangle \notin T\backslash b.$$

If there were an event $c \in X$ such that $u\langle c\rangle \in T\backslash b$, there would also be a trace $w \in T$ with $w\backslash b = u\langle c\rangle$. But then w would have a prefix v with $v\backslash b = u$ and $v\langle c\rangle \in T$. Thus v must be of the form $tb^i, i \leq n$. But by unavoidability of b, this forces v to be s; now we have a contradiction, since we are assuming $s\langle c\rangle \notin T$. Thus we must have $u\langle c\rangle \notin T\backslash b$. That completes the proof. ∎

8. Another version of hiding

As remarked in the previous section, hiding an event which may occur an unbounded number of times effectively allows a process to perform arbitrarily many hidden actions, without participating in any visible action. Previously we examined the consequences of identifying such behaviour with deadlock, arguing that a process forever engaging in internal actions appears deadlocked to its environment. However, another interpretation of infinite internal activity is possible. It can be argued that the future behaviour of a process engaging in internal actions is unpredictable until it stops performing hidden actions. As far as its environment is concerned, a process forever performing hidden events is totally unpredictable: there is no way of telling whether or not the process might at some future moment stop its internal activity and allow some visible action. It is consistent with this interpretation to identify infinite internal activity with CHAOS, the most nondeterministic of all processes. In this section we consider an alternative definition of a hiding operation; it differs from the earlier operation only in the treatment of infinite chatter. Most of its properties are analogous to those of the other operation, but this version of hiding is *continuous,* unlike the other version. We omit most of the proofs, in cases where they are comparable to the proofs given in the previous section. We include a proof of continuity.

Definition 2.8.1:

$$P/b = \{(s\backslash b, X) \mid (s, X \cup \{b\}) \in P\} \cup \{((s\backslash b)t, X) \mid \forall n.sb^n \in \text{traces}(P) \,\&\, (t, X) \in \text{CHAOS}\}.$$

Recall that the previous version of hiding was defined:

$$P\backslash b = \{(s\backslash b, X) \mid (s, X \cup \{b\}) \in P\} \cup \{((s\backslash b)t, X) \mid \forall n.sb^n \in \text{traces}(P) \,\&\, (t, X) \in \text{STOP}\}.$$

Theorem 2.8.2: If P is a process so is P/b. In the absence of infinite internal chatter, $P\backslash b = P/b$.

Proof. When there is no possibility of internal chatter, both definitions of hiding reduce to:

$$P\backslash b = P/b = \{(s\backslash b, X) \mid (s, X \cup \{b\}) \in P\}.$$

∎

Lemma 2.8.3: Let P_B be a guarded process: $P_B = \Box_{b \in B}(b \to P_b)$. Then

$$P_B/a = \Box_{b \in B}(b \to P_b/a) \quad \text{if } a \notin B,$$
$$= P_a/a \sqcap (P_a/a \,\Box\, P_C/a) \quad \text{if } B = C \cup \{a\}, a \notin C.$$

Corollary:

$$\begin{array}{ll} \text{(i)} & \text{STOP}/b = \text{STOP} \\ \text{(ii)} & (a \to P)/b = (a \to P/b) \qquad (a \neq b) \\ \text{(iii)} & (b \to P)/b = P/b. \end{array}$$

Examples.

Example 1.

$$(a \to (b \to \text{STOP}))/a = (b \to \text{STOP}).$$

Example 2.

$$(a \to (b \to \text{STOP}))/b = (a \to \text{STOP}).$$

Example 3.

$$((a \to \text{STOP}) \,\Box\, (b \to \text{STOP}))/b = (a \to \text{STOP}) \sqcap \text{STOP}.$$

Example 4. The processes P_n are defined as in *Example 4* of the previous section:

$$P_0 = \text{chaos}(a^*b)$$
$$P_{n+1} = (a \to P_n).$$

The limit process is $P = \text{det}(a^*)$. Since each P_n can perform arbitrarily many hidden actions when we hide a, for each n

$$P_n/a = \text{CHAOS}.$$

The limit process can also indulge in infinite internal action, so that

$$P/a = \text{CHAOS}.$$

This example is no longer a counterexample to continuity.

Example 5. The processes Q_n are defined as in *Example 5* of the previous section:

$$Q_0 = (b \to \text{STOP})$$
$$Q_{n+1} = (a \to Q_n).$$

Again let $Q = \bigcup_n Q_n$. Now we have

$$Q_n = (a^n b \to \text{STOP})$$
$$Q_n/a = (b \to \text{STOP})$$
$$Q/a = \text{CHAOS}.$$

Again this example shows that this version of hiding is not generally distributive.

Theorem 2.8.4: $\lambda P.P/b$ is monotone and finitely distributive.

Theorem 2.8.5: If b is unavoidable in T and T is free from internal chatter, then

$$\det(T)/b = \det(T\backslash b).$$

Next we prove that this formulation of the hiding operator defines a continuous function on processes. Roscoe [R] contains an alternative proof. First, we need a lemma; its proof has been relegated to the Appendix, as it is somewhat technical. The proof given in the appendix is for a slightly generalised version. Here we state only the particular form necessary for the continuity proof.

Lemma A: If $\{ s_n \mid n \geq 0 \}$ is a sequence of traces such that for the event b the traces $s_n\backslash b$ are uniformly bounded, i.e., there is a trace u such that

$$s_n\backslash b \leq u, \qquad \text{for all } n,$$

then either infinitely many s_n are identical or there is a trace s such that

$$\forall k.\exists n.sb^k \leq s_n.$$

Theorem 2.8.6: $\lambda P.P/b$ is continuous.

Proof. Let $\{ P_n \mid n \geq 0 \}$ be a chain of processes with limit P. We must show that hiding b in each term of this chain produces another chain, whose limit is P/b. By monotonicity, the chain property is obvious, and we have one half of the identity:

$$P/b \subseteq \bigcap_n (P_n/b).$$

For the converse, suppose $(u, X) \in \bigcap_n (P_n/b)$. We need to prove that (u, X) is also a failure of P/b. For this, we require one of the two following conditions to hold:

$$\begin{array}{lll} \text{either} & (1) & \exists s.\ s\backslash b = u \ \&\ (s, X \cup \{ b \}) \in P \\ \text{or} & (2) & \exists s.\ s\backslash b \leq u \ \&\ \forall n.(sb^n, \emptyset) \in P. \end{array}$$

But we are assuming for each n that (u, X) is a failure of P_n/b, so we have, for each n,

$$\begin{array}{lll} \text{either} & (1') & \exists s_n.\ s_n\backslash b = u \ \&\ (s_n, X \cup \{ b \}) \in P_n \\ \text{or} & (2') & \exists s_n.\ s_n\backslash b \leq u \ \&\ \forall m.(s_n b^m, \emptyset) \in P_n. \end{array}$$

One of these alternatives must hold for infinitely many n, and therefore for all n, using the chain condition. Without loss of generality, therefore, we can assume the existence of a sequence of traces s_n such that either $(1')$ holds for

all n or (2′) holds for all n. In both cases the traces s_n all satisfy the condition

$$s_n \backslash b \leq u.$$

Now we can use Lemma A. This guarantees either that the s_n are uniformly bounded in length, or that they contain arbitrarily long sequences of the event to be hidden, in the following sense:

$$\begin{array}{ll} \text{either} & \text{(i)} \quad \text{length}(s_n) \leq N, \text{ for some integer } N, \\ \text{or} & \text{(ii)} \quad \text{infinitely many } s_n \text{ have a prefix } sb^{k(n)}, \end{array}$$

for some integer $k(n)$ and a fixed trace s. In the first case, the sequence s_n must contain a constant subsequence, i.e., there is an increasing sequence of indices n_k and a trace s such that

$$s_{n_k} = s, \quad \text{for all } k.$$

In the second case, there is an increasing sequence of indices n_k and a trace s such that

$$sb^k \leq s_{n_k}, \quad \text{for all } k.$$

Recall that either (1′) or (2′) must hold. Now it is easy to see that the first alternative implies (1) if (1′) holds, and (2) if (2′) holds, by the chain condition; and the second alternative implies (1′) always. That completes the proof. ∎

The two hiding operations described so far enjoy, as we have seen many common properties: a notable exception is continuity, which only holds for the second version of hiding, where infinite chatter was identified with CHAOS. From now on, we will concentrate exclusively on this second, preferred version of hiding.

Compositions of hiding operations.

So far we have seen that hiding an event is a (finitely) distributive operation, and hence monotonic with respect to the ordering; and hiding, in the preferred version, is also continuous. In addition, hiding an event does not preserve determinism unless the event being hidden was unavoidable. Next we consider what happens if more than one event is hidden. We will see that the order of hiding does not matter.

When deleting a set of events from a trace, the order in which those events are deleted is irrelevant:

$$(s\backslash b)\backslash c = (s\backslash c)\backslash b.$$

Also, once an event has been deleted a further deletion of the same event has no effect:

$$(s\backslash b)\backslash b = s\backslash b.$$

We may, therefore, introduce the notation $s\backslash B$ for the result of hiding all occurrences of the events in the set B. When B is a finite set $\{ b_1, \ldots, b_n \}$,

$$s\backslash B = (\ldots(s\backslash b_1)\ldots\backslash b_n).$$

It will also be convenient to adopt the convention that

$$s\backslash\emptyset = s.$$

An obvious corollary is that, for all finite sets B and C,

$$(s\backslash B)\backslash C = s\backslash(B \cup C).$$

Next we show that these results can be extended to yield similar results about the hiding operation on processes. First we will need a lemma concerning the hiding operation on traces. We use a slight generalisation of the Lemma used in the continuity proof (where the case $B = \{ b \}$ was used).

Lemma A: If $\{ s_n \mid n \geq 0 \}$ is a sequence of traces and B is a finite set of events such that each of the traces $s_n\backslash B$ is uniformly bounded, i.e., there is a trace u such that

$$s_n\backslash B \leq u, \quad \text{for all } n,$$

then either infinitely many s_n are identical or there is an infinite sequence $t \in B^\infty$ and a trace s such that

$$\forall k.\exists n.\ st_k \leq s_n,$$

where for each k t_k is the prefix of t having length k.

Proof. See Appendix. ∎

If we hide first b then c, there will be a possibility of internal chatter if and only if the original process was able to perform arbitrarily long sequences of actions drawn from the set $\{b, c\}$. The traces will be obtained by removing all occurrences of b and c, and the process $(P/b)/c$ will be able to refuse a set X if P could have refused at the corresponding stage the set X along with the events to be hidden. These informal comments serve to explain the following result.

Theorem 2.8.7: Let b and c be events and P be a process.

$$(P/b)/c = \{((s\backslash b)\backslash c, X) \mid (s, X \cup \{b, c\}) \in P\} \\ \cup \{(((s\backslash b)\backslash c)w, X) \mid \forall n.\exists t \in \{b, c\}^n.\ st \in \text{traces}(P)\}.$$

Proof. By definition, a pair (u, X) is a failure of $(P/b)/c$ if and only if

$$\begin{array}{lll} \text{either} & (1) & \exists t.\ t\backslash c = u\ \&\ (t, X \cup \{c\}) \in P/b \\ \text{or} & (2) & \exists t.\ t\backslash c \leq u\ \&\ \forall n.(tc^n, \emptyset) \in P/b. \end{array}$$

In case (1) we have $(t, X \cup \{c\}) \in P/b$, or, equivalently:

$$\begin{array}{lll} \text{either} & (i) & \exists s.\ s\backslash b = t\ \&\ (s, X \cup \{b, c\}) \in P \\ \text{or} & (ii) & \exists s.\ s\backslash b \leq t\ \&\ \forall k.(sb^k, \emptyset) \in P. \end{array}$$

In case (2) we have, for each n,

$$\begin{array}{lll} \text{either} & (iii) & \exists s_n.\ s_n\backslash b = tc^n\ \&\ (s_n, \{b\}) \in P \\ \text{or} & (iv) & \exists s_n.\ s_n\backslash b \leq tc^n\ \&\ \forall m.(s_n b^m, \emptyset) \in P. \end{array}$$

One of these alternatives must hold for infinitely many n, and hence for all n. It cannot be the case that infinitely many of the s_n are equal traces, because (iii) guarantees that

$$\text{length}(s_n) \geq \text{length}(t) + n, \quad \text{for all } n.$$

By Lemma A (putting $B = \{b, c\}$) we know there is an infinite subsequence of the s_n consisting of longer and longer extensions of a common trace by events to be hidden. More precisely, we know that there is a subsequence s_{n_k} having prefixes of the form st_k, where each t_k is a sequence over $\{b, c\}$ and where the t_k form an increasing sequence:

$$t_k \in \{b, c\}^*\ \&\ t_k < t_{k+1}, \quad \text{for all } k.$$

We lose no generality if we replace the original sequence by the subsequence satisfying this property. It follows that the condition for (u, X) to be a failure of $(P/b)/c$ is equivalent to the following:

$$\begin{array}{lll} \text{either} & (i) & \exists s.\ (s\backslash b)\backslash c = u\ \&\ (s, X \cup \{b, c\}) \in P \\ \text{or} & (ii) & \exists s.\ (s\backslash b)\backslash c \leq u\ \&\ \forall k.(sb^k, \emptyset) \in P \\ \text{or} & (iii) & \exists s.(s\backslash b)\backslash c \leq u\ \&\ \forall k.\exists t_k \in B^k\ \&\ t_k < t_{k+1}\ \&\ (st_k, \emptyset) \in P. \end{array}$$

Since (ii) is clearly a special case of (iii), this is the required condition. ∎

Corollary 2.8.8:

$$\begin{array}{ll} \text{(i)} & (P/b)/c = (P/c)/b \\ \text{(ii)} & (P/b)/b = P/b. \end{array}$$

Now we can extend the notation used for traces; if B is any finite set of events $\{ b_1, \ldots, b_n \}$, we will write

$$P/B = (\ldots(P/b_1)\ldots/b_n).$$

In the special case where B is a singleton, we have $P/\{ b \} = P/b$, and when B is empty we take $P/\emptyset = P$.

The following properties are immediate.

Corollary 2.8.9:

$$\begin{aligned} P/B = &\{ (s\backslash B, X) \mid (s, X \cup B) \in P \} \\ &\cup \{ ((s\backslash B)w, X) \mid \forall n. \exists t \in B^n.\ st \in \text{traces}(P) \}. \end{aligned}$$

Corollary 2.8.10:

$$\begin{array}{ll} \text{(i)} & (P/B)/C = (P/C)/B = P/(B \cup C). \\ \text{(ii)} & (P/B)/B = P/B. \end{array}$$

Since hiding a single event is a monotone and finitely distributive operation, the generalised hiding operation $/B$ is also monotone and finitely distributive. Similarly, the generalised operation is continuous. Many of the identities already established for the single event hiding operation can be restated for $/B$.

The following result states a condition on trace sets under which hiding a set B preserves determinism.

Theorem 2.8.11: If *each* event in the set B is unavoidable in T, then

$$\det(T)/B = \det(T\backslash B).$$

Proof. By induction on the size of B. The base case is Theorem 2.7.10. For the inductive step, one merely shows that if all events in $B \cup \{ b \}$ are unavoidable in T, then b is unavoidable in the tree $T\backslash B$. Details are omitted. ∎

Examples.

Example 1. For any pair of processes P and Q,

$$R = (a \to P) \,\Box\, (b \to Q)$$

is a process which allows its environment to select either P or Q on the first step, by performing the appropriate initial event. If we hide first event a, we get

$$R/a = P/a \sqcap ((P/a) \,\Box\, (b \to Q/a)).$$

In this process, the environment cannot be sure that its attempt to perform b will succeed. Finally, if we hide b now, all control of the first step is removed, and we get:

$$R/\{a, b\} = P/\{a, b\} \sqcap Q/\{a, b\}.$$

Notice that here the events being hidden were *not* unavoidable, and nondeterminism was introduced on the first step.

9. Sequential composition

Now we introduce a sequencing operation. Suppose the distinguished event $\surd$ (pronounced "tick") represents successful termination of a process. We will say that a process has terminated when it has performed $\surd$. Similarly, if the set $\{\surd\}$ is a possible refusal of P we say that P can refuse to terminate. The sequential composition of two processes P and Q, which we denote $P;Q$, is intended to behave like P until P terminates, after which it should behave like Q. The termination of the first component process, however, is regarded as an internal event hidden from the environment. As in the definition of hiding, therefore, it is reasonable to allow the constructed process $P;Q$ to refuse a set X, at a stage when the first component process is still running, only when the first process can refuse X and refuse to terminate, i.e., when the first process can refuse the set $X \cup \{\surd\}$.

Definition 2.9.1: A trace s is tick-free if $\surd$ does not appear in s. For any tick-free trace s and any trace t we define:

$$(1) \qquad \langle\rangle;t = t$$
$$(2) \qquad s\surd u;t = st.$$

This defines the sequencing operation on all traces, and extends to sets of traces in the obvious way:

$$S;T = \{\, s;t \mid s \in S \,\&\, t \in t \,\}.$$

Some obvious properties of sequence composition of traces are summarised in the following result.

Corollary 2.9.2:

$$(1) \qquad \langle\rangle;t = \langle\rangle$$
$$(2) \qquad s\surd;t = st \quad (\text{if } s \text{ is tick-free})$$
$$(3) \qquad (s;t);u = s;(t;u).$$

Definition 2.9.3:

$$P;Q = \{\,(s,X) \mid (s, X \cup \{\surd\}) \in P \,\&\, s \text{ tick-free}\,\}$$
$$\cup\, \{\,(st,X) \mid s\surd \in \text{traces}(P) \,\&\, s \text{ tick-free} \,\&\, (t,X) \in Q\,\}.$$

Theorem 2.9.4: If P and Q are processes so is $P;Q$, and

$$\text{traces}(P;Q) = \text{traces}(P);\text{traces}(Q).$$

Proof. By definition, every trace of $P;Q$ is a sequential composition of traces of P and Q :

$$\text{traces}(P;Q) \subseteq \text{traces}(P);\text{traces}(Q).$$

Conversely, if $u \in \text{traces}(P);\text{traces}(Q)$, there are two cases to consider. Either u is a tick-free trace of P, or there are traces $s \in \text{traces}(P)$, and $t \in \text{traces}(Q)$, with $st = u$ and $s\surd \in \text{traces}(P)$. In the first case we have $(u, \emptyset) \in P$, and we want to deduce that $(u, \emptyset) \in P;Q$. This will follow from Definition 2.9.3 if $u\surd \notin \text{traces}(P)$, because we will then have $(u, \{\surd\}) \in P$. When $u\surd \in P$, it follows from the second clause of Definition 2.9.3, because $(\langle\rangle, \emptyset) \in Q$. The second case also follows immediately from the definition. Thus we have shown that

$$\text{traces}(P);\text{traces}(Q) = \text{traces}(P;Q).$$

To complete the proof, we must verify conditions (P0)–(P3). Of these, the first two are trivial. We give details only for (P3). Suppose $(u, X) \in P;Q$ and $u\langle c\rangle \notin \text{traces}(P;Q)$. We want to prove that $(u, X \cup \{c\}) \in P;Q$. As usual, there are two cases. The first possibility is that u is a tick-free trace of P, and $(u, X \cup \{\surd\}) \in P$. If $c = \surd$ there is nothing to prove. Otherwise, we know that

$$u\langle c\rangle \notin \text{traces}(P);\text{traces}(Q), \qquad u \in \text{traces}(P).$$

This means that $u\langle c\rangle$ cannot be a trace of P, because this would again be a tick-free trace and hence be a trace of $P;Q$. Since P is a process, it follows by (P3) that

$$(u, X \cup \{c, \surd\}) \in P,$$

and hence that

$$(u, X \cup \{c\}) \in P;Q,$$

as required. The second possibility is that $u = st$, for some tick-free trace s of P, and

$$s\surd \in \text{traces}(P) \ \&\ (t, X) \in Q.$$

In this case, $t\langle c\rangle$ cannot be a trace of Q, because this would imply that $st\langle c\rangle$ is a trace of $P;Q$, in contradiction to our assumption. By (P3) for the process Q, we get

$$(t, X \cup \{c\}) \in Q,$$

from which it follows that $(u, X \cup \{c\}) \in P;Q$. That completes the proof. ∎

Examples.

Example 1.

$$\text{SKIP} = (\surd \rightarrow \text{STOP})$$

This process terminates immediately, without performing any other action. It has the property that, for all processes Q,

$$\text{SKIP};Q = Q.$$

Example 2.

$$\text{STOP};Q = \text{STOP}$$

STOP never performs any action, and hence cannot terminate. More generally, $P;Q = P$, whenever all traces of P are tick-free.

Example 3.

$$(a \rightarrow \text{SKIP});(b \rightarrow \text{SKIP}) = (a \rightarrow (b \rightarrow \text{SKIP}))$$

Here the first process initially performs a and terminates, whereupon the second process performs b and terminates. The event signifying termination of the first process is concealed from the environment, and is used only internally.

Example 4. Let P and Q be the processes

$$P = (a \rightarrow \text{SKIP}) \,\Box\, \text{SKIP},$$
$$Q = (a \rightarrow \text{SKIP}) \sqcap \text{SKIP}.$$

They both have the same traces but only Q can initially refuse to terminate. Although P is deterministic, the sequential composition of P with another deterministic process need not be deterministic. This is because $\surd$ is not an avoidable event for P. For example, a simple calculation shows that

$$P;(b \rightarrow \text{SKIP}) = ((a \rightarrow (b \rightarrow \text{SKIP})) \,\Box\, (b \rightarrow \text{SKIP})) \sqcap (b \rightarrow \text{SKIP}).$$

The important fact here is that $P;(b \rightarrow \text{SKIP})$ can initially refuse $\{a\}$ but cannot refuse $\{b\}$. By way of contrast, $Q;(b \rightarrow \text{SKIP})$ can initially refuse either of these sets, and

$$Q;(b \rightarrow \text{SKIP}) = (a \rightarrow (b \rightarrow \text{SKIP})) \sqcap (b \rightarrow \text{SKIP}).$$

Properties of sequential composition.

It should be clear from the definition of sequential composition that, for any tick-free trace u of P, once $P;Q$ has performed u it is possible for the first process P to have progressed:

$$P;Q \,\underline{\text{after}}\, u \sqsubseteq (P \,\underline{\text{after}}\, u);Q.$$

Similarly, if $u = st$, and $s\surd \in \text{traces}(P), t \in \text{traces}(Q)$, then the first process may have terminated and the second may now be running:

$$P;Q \,\underline{\text{after}}\, u \sqsubseteq Q \,\underline{\text{after}}\, t.$$

In fact, a particular trace u of $P;Q$ may arise in more than one way from traces of P and Q, and in general $P;Q$ after u is the union of the several processes corresponding to the various cases. The following Theorem states this precisely.

Theorem 2.9.5: For all processes P and Q, and all traces u,

$$(P;Q) \text{ after } u = \bigcup\{ (P \text{ after } u);Q \mid u \in \text{traces}(P) \ \&\ u \text{ tick-free} \} \cup \bigcup\{ Q \text{ after } t \mid \exists s.\ s\surd \in \text{traces}(P) \ \&\ st = u \}.$$

Proof. Elementary. ∎

Although Example 4 showed that sequential composition is not generally a deterministic operation, it does preserve determinism in cases where termination is unavoidable; this is not surprising, since there is a close analogy with the hiding operation.

Theorem 2.9.6: If $\surd$ is unavoidable in S, then

$$\text{det}(S);\text{det}(T) = \text{det}(S;T).$$

Theorem 2.9.7: For any set B of events, not containing $\surd$,

$$\Box_{b \in B}(b \to P_b);Q = \Box_{b \in B}(b \to P_b;Q).$$

Corollary: If $a \neq \surd$, then for all processes P, Q

$$(a \to P);Q = (a \to P;Q).$$

The next results establish the important properties of monotonicity, distributivity and continuity. Since sequential composition is not a symmetric operation, it is most convenient to deal with both arguments at once. As usual, monotonicity follows from distributivity.

Theorem 2.9.8: Sequential composition is distributive.

Proof. Let $\mathcal{D}$ and $\mathcal{E}$ be non-empty sets of processes and let P and Q be their suprema. We need to show that

$$P;Q = \bigsqcup\{ P';Q' \mid P' \in \mathcal{D} \ \&\ Q' \in \mathcal{E} \}.$$

As usual we need to establish two inclusions. Let $(u, X) \in P;Q$. We want to show that there are processes $P' \in \mathcal{D}$ and $Q' \in \mathcal{E}$ such that $(u, X) \in P';Q'$. There are two cases. The first case is when u is a tick-free trace of P and $(u, X \cup \{\surd\}) \in P$. By definition of P this means there is a process $P' \in \mathcal{D}$ with $(u, X \cup \{\surd\}) \in P'$. Then it is easy to see that (u, X) is a failure of $P';Q'$,

for *all* processes $Q' \in \mathcal{E}$. The other case is when there are traces s and t with

$$u = st, \qquad s\surd \in \text{traces}(P), \qquad (t, X) \in Q.$$

But then there are processes $P' \in \mathcal{D}$ and $Q' \in \mathcal{E}$ such that

$$u = st, \qquad s\surd \in \text{traces}(P'), \qquad (t, X) \in Q',$$

and hence $(u, X) \in P';Q'$, as required.

The reverse inclusion is similar. ∎

Theorem 2.9.9: Sequential composition is continuous.

Proof. Let $\{ P_n \mid n \geq 0 \}$ and $\{ Q_n \mid n \geq 0 \}$ be chains with limits P and Q. We show that the processes $P_n;Q_n$ form a chain with limit $P;Q$. The continuity of sequential composition in both arguments follows if we replace one of these chains by a constant.

By monotonicity, the processes $\{ P_n;Q_n \mid n \geq 0 \}$ form a chain and

$$\bigsqcap_n (P_n;Q_n) \sqsubseteq P;Q.$$

For the converse, let (u, X) be a failure of the left-hand side; we must prove that $(u, X) \in P;Q$. By hypothesis, for each n we have

either (1) u is tick-free and $(u, X \cup \{\surd\}) \in P_n$,
or (2) $\exists s_n, t_n \quad s_n t_n = u \;\&\; s_n\surd \in \text{traces}(P_n) \;\&\; (t_n, X) \in Q_n$.

One of these alternatives must hold for infinitely many values of n. In the first case we get immediately $(u, X \cup \{\surd\}) \in P$, by the chain property; this gives $(u, X) \in P;Q$ as required. In the second case we use the fact that u has only finitely many prefixes to find a pair of traces s and t with

$$st = u, \quad s\surd \in \text{traces}(P_n), \quad (t, X) \in Q_n,$$

for infinitely many n. The result again follows by the chain property. That completes the proof. ∎

Theorem 2.9.10: Sequential composition is associative,

$$P;(Q;R) = (P;Q);R.$$

Proof. We already know that sequential composition of traces is associative. An elementary expansion of the definition shows that the failures (v, X) of $P;(Q;R)$ fall into three categories characterised as follows:

(i) v tick-free, $(v, X \cup \{\surd\}) \in P$
(ii) $v = st$, s and t tick-free, $s\surd \in \text{traces}(P)$ and $(t, X) \in Q$
(iii) $v = stu$, s and t tick-free, $s\surd \in \text{traces}(P), t\surd \in \text{traces}(Q)$ and $(u, X) \in R$.

The same is true of $(P;Q);R$. ∎

Examples.

Example 1. The recursively defined process

$$P = (a \rightarrow P;(a \rightarrow \text{SKIP})) \,\Box\, (b \rightarrow \text{SKIP})$$

is deterministic and can terminate successfully after any trace of the form $a^n b a^n$. This process can be thought of as "recognising" the formal language $\{ a^n b a^n \mid n \geq 0 \}$.

Example 2. The recursion

$$P = \text{SKIP} \Box (a \rightarrow \text{SKIP}) \Box (b \rightarrow \text{SKIP}) \Box (a \rightarrow P;(a \rightarrow \text{SKIP})) \Box (b \rightarrow P;(b \rightarrow \text{SKIP}))$$

defines a process which recognises palindromes in the letters a and b.

These examples and others in similar vein (see, for example, [H1,2]) show that unrestrained use of the sequential composition operation in recursive process definitions can produce processes with highly non-regular trace sets.

10. Iteration

There is a derived operation which corresponds informally to the idea of repeated sequential composition (or iteration) of a process with itself. Since sequential composition is continuous we know that, for any process P the function $\lambda Q.P;Q$ has a least fixed point. We denote this fixed point process by $\underline{*}P$. It is the limit of the sequence:

$$\begin{aligned} P_0 &= \text{CHAOS}, \\ P_{n+1} &= P;P_n, \end{aligned}$$

and the fixed point equation is

$$P;\underline{*}P = \underline{*}P.$$

Writing P^n for the n-fold iterate of P, i.e.,

$$\begin{aligned} P^1 &= P, \\ P^{n+1} &= P;P^n \quad (n \geq 1), \end{aligned}$$

it follows that $P_n = P^n;\text{CHAOS}$ for all $n \geq 1$. This notation will be used throughout the section.

Examples.

Example 1. The equation $P = (a \rightarrow P)$ has a unique solution $\text{det}(a^*)$. Using the fixed point equation for iteration, we have

$$\begin{aligned} \underline{*}(a \rightarrow \text{SKIP}) &= (a \rightarrow \text{SKIP});(\underline{*}(a \rightarrow \text{SKIP})) \\ &= (a \rightarrow \underline{*}(a \rightarrow \text{SKIP})), \end{aligned}$$

and hence $\underline{*}(a \rightarrow \text{SKIP}) = \text{det}(a^*)$.

Example 2.

$$\underline{*}\text{STOP} = \text{STOP}$$

More generally, whenever all traces of P are tick-free, $\underline{*}P = P$.

Example 3. Define two processes P and Q by:

$$\begin{aligned} P &= (a \rightarrow \text{SKIP}) \sqcap (b \rightarrow \text{SKIP}), \\ Q &= (a \rightarrow \text{SKIP}) \,\Box\, (b \rightarrow \text{SKIP}). \end{aligned}$$

Then, by analogy with Example 1, $\underline{*}Q = \det(\{a, b\}^*)$. However, the iterate of P is the limit of the chain:

$$\begin{aligned} P_0 &= \text{CHAOS}, \\ P_{n+1} &= ((a \rightarrow \text{SKIP}) \sqcap (b \rightarrow \text{SKIP}));P_n \\ &= (a \rightarrow P_n) \sqcap (b \rightarrow P_n). \end{aligned}$$

This is a chain which we have met before, so we have:

$$\begin{aligned} \text{traces}(\underline{*}P) &= \{a, b\}^* \\ \text{refusals}(\underline{*}P \text{ after } s) &= \{\emptyset, \{a\}, \{b\}\} \qquad (s \in \{a, b\}^*). \end{aligned}$$

Notice that iteration is not distributive, because

$$\begin{aligned} \underline{*}(a \rightarrow \text{SKIP}) &= \det(a^*), \\ \underline{*}(b \rightarrow \text{SKIP}) &= \det(b^*), \\ P &= (a \rightarrow \text{SKIP}) \sqcap (b \rightarrow \text{SKIP}), \\ \underline{*}P &\neq \det(a^*) \sqcap \det(b^*). \end{aligned}$$

The reason for this lack of distributivity is simple: in forming $\underline{*}P$ more than one copy of P is used, and if P is nondeterministic there is no reason to suppose that each copy will behave identically. In this example, P can decide between a and b, and $\underline{*}P$ can decide at each stage which of these events to allow next, regardless of past choices.

Example 4.

$$\underline{*}\text{SKIP} = \text{CHAOS}$$

The reason for this result is that $\underline{*}$SKIP is the limit of the chain:

$$\begin{aligned} P_0 &= \text{CHAOS}, \\ P_{n+1} &= \text{SKIP};P_n, \\ &= P_n, \end{aligned}$$

all of whose terms are equal to CHAOS. It can be argued that this does not really capture the intuitive behaviour of this process: surely $\underline{*}$SKIP is analogous to a non-terminating loop or a divergent process, since it is in effect engaging in an infinite sequence of invisible actions. In the extended model of processes of Chapter 5, where we take into account divergent behaviour explicitly, this identity will no longer hold.

Lemma 2.10.1: Iteration is monotone, i.e., $P \sqsubseteq Q \Rightarrow \underline{*}P \sqsubseteq \underline{*}Q$.

Proof. Using the obvious notation, $\underline{*}P = \bigcap_n P_n$, and $\underline{*}Q = \bigcap_n Q_n$, where

$$\begin{aligned} P_0 &= \text{CHAOS}, & Q_0 &= \text{CHAOS}, \\ P_{n+1} &= P;P_n, & Q_{n+1} &= Q;Q_n. \end{aligned}$$

Using monotonicity of sequential composition we can prove, by induction on n that, for all $n \geq 0$,

$$P \sqsubseteq Q \Rightarrow P_n \sqsubseteq Q_n.$$

It follows easily that $\underline{*}P \sqsubseteq \underline{*}Q$. ∎

Lemma 2.10.2: Iteration is continuous.

Proof. Let $\{ P_n \mid n \geq 0 \}$ be a chain with limit P. Define a doubly indexed sequence of processes P_{ij} for $i, j \geq 1$ by

$$P_{ij} = P_i{}^j;\text{CHAOS}.$$

Then for each i the set $\{ P_{ij} \mid j \geq 1 \}$ is a chain with limit

$$\bigcap_j P_{ij} = \underline{*}P_i.$$

Since sequential composition is monotone and the P_i form a chain, the set $\{ P_{ij} \mid i \geq 1 \}$ is also a chain for each j. Moreover, the limit of this chain is

$$\bigcap_i P_{ij} = \bigcap_i P_i{}^j;\text{CHAOS} = P^j;\text{CHAOS}.$$

But $\bigcap_j P^j = P$, by definition. Thus,

$$\begin{aligned} \underline{*}P &= \bigcap_j P^j;\text{CHAOS} \\ &= \bigcap_j \bigcap_i P_{ij} \\ &= \bigcap_i \bigcap_j P_{ij} \\ &= \bigcap_i \underline{*}P_i \end{aligned}$$

as required. ∎

11. Alphabet transformation

A total function $f : \Sigma \to \Sigma$ can be used to *rename* events. We will use the notation $f[P]$ for the process which can perform the event $f(c)$ whenever P could have performed c, and which can refuse an event b just in case P can refuse all events which are renamed to b. In other words, the traces of $f[P]$ are obtained by renaming the traces of P, and $f[P]$ can refuse a set X when P can refuse the inverse image $f^{-1}(X)$.

When f is injective $f[P]$ behaves exactly like a *copy* of P. However, in general f may *identify* one or more events which were previously distinguishable. In the case where events $c_1, \ldots, c_n$ are identified,

$$f(c_1) = \cdots = f(c_n) = b,$$

the behaviour of $f[P]$ after performing b could be that of any of the processes $f[P \underline{\text{after}}\, c_i]$, for $i = 1 \ldots n$. In other words, when $f[P]$ performs an event corresponding to several different original events, the environment cannot tell which of the original events actually occurred.

We will be interested in the case when f only makes finitely many identifications; this condition guarantees that the inverse image of any event in the range of f is a finite set. For our purposes, then, the following definition is in order.

Definition 2.11.1: An *alphabet transformation* f is a total function

$$f : \Sigma \to \Sigma$$

such that for any event b, the set $f^{-1}(b)$ is finite.

Notation: The following conventional notation will be used, extending the function f to sets and sequences of events in the usual way:

$$\begin{aligned} f^{-1}(X) &= \{\, y \in \Sigma \mid f(y) \in X \,\} \\ f(X) &= \{\, f(x) \mid x \in X \,\} \\ f(\langle c_1, \ldots, c_n \rangle) &= \langle f(c_1), \ldots, f(c_n) \rangle \\ f^{-1}(T) &= \{\, s \in \Sigma^* \mid f(s) \in T \,\} \\ f(T) &= \{\, f(s) \mid s \in T \,\}. \end{aligned}$$

Note that

(i) $f(\langle\,\rangle) = \langle\,\rangle$
(ii) $f(st) = f(s)f(t)$.

Definition 2.11.2:

$$f[P] = \{\, (f(s), X) \mid (s, f^{-1}(X)) \in P \,\}.$$

Theorem 2.11.3: If P is a process so is $f[P]$ and it is uniquely characterised by the following conditions:

$$\begin{array}{ll} \text{(i)} & \text{traces}(f[P]) = f(\text{traces}(P)) \\ \text{(ii)} & \text{refusals}(f[P]) = \{ X \mid f^{-1}(X) \in \text{refusals}(P) \} \\ \text{(iii)} & f[P] \underline{\text{after}}\, t = \bigcup\{ f[P \underline{\text{after}}\, s] \mid f(s) = t \}. \end{array}$$

Proof. By elementary properties of alphabet transformations. To verify that $f[P]$ is a process we check conditions (P0)–(P3). Property (P0) follows from the facts that $f(\langle\rangle) = \langle\rangle$ and that $s \leq u$ always implies $f(s) \leq f(u)$. For (P1) we use the fact that

$$Y \subseteq X \Rightarrow f^{-1}(Y) \subseteq f^{-1}(X).$$

Finally, for (P3), suppose (t, X) is a failure of $f[P]$, but that ta is not a trace. We wish to show that $(t, X \cup \{ a \})$ is a failure of $f[P]$. By assumption there is a trace s of P with

$$(s, f^{-1}(X)) \in P \ \&\ f(s) = t.$$

Let $A = f^{-1}(a)$. Then $f^{-1}(X \cup \{ a \}) = f^{-1}(X) \cup A$. If for any event $b \in A$ the trace sb was possible for P, it would also be possible for $f[P]$ to perform $f(sb) = ta$. By assumption, therefore, all events in the set A are impossible for P after s. Since f is an alphabet transformation it has the finite pre-image property, so A is finite. By finitely many applications of (P3) for the process P we get

$$(s, f^{-1}(X) \cup A) \in P$$

and hence

$$(t, X \cup \{ a \}) \in f[P]$$

as required. This argument still holds even if a is not in the range of f, when A is empty. ∎

Several properties of the renaming operation are immediate.

Theorem 2.11.4: The identity function $\iota : \Sigma \to \Sigma$ is a renaming, and the composition $f \circ g$ of two renamings f and g is itself a renaming. Moreover, for all processes P,

$$\begin{array}{ll} \text{(i)} & \iota[P] = P \\ \text{(ii)} & (f \circ g)[P] = f[g[P]]. \end{array}$$

Theorem 2.11.5:

$$\begin{array}{ll} \text{(i)} & f[\text{STOP}] = \text{STOP} \\ \text{(ii)} & f[c \to P] = (f(c) \to f[P]) \\ \text{(iii)} & f[\text{chaos}(T)] = \text{chaos}(f(T)) \\ \text{(iv)} & f[\text{det}(T)] \sqsubseteq \text{det}(f(T)), \quad \text{with equality when } f \text{ is injective.} \end{array}$$

Theorem 2.11.6: Renaming is a monotone, distributive and continuous operation:

$$\begin{array}{ll} \text{(i)} & P \sqsubseteq Q \Rightarrow f[P] \sqsubseteq f[Q] \\ \text{(ii)} & f[\bigcup \mathcal{D}] = \bigcup\{ f[P] \mid P \in \mathcal{D} \} \\ \text{(iii)} & f[\bigcap_n P_n] = \bigcap_n f[P_n] \end{array}$$

for all nonempty sets $\mathcal{D}$ and chains $\{ P_n \mid n \geq 0 \}$.

The next result shows that hiding an event *after* a renaming has the same effect as hiding all events which map to the hidden event *before* the renaming. The proof can be easily adapted for the second version of the hiding operator.

Theorem 2.11.7: Let b be an event in the range of an alphabet transformation f, and let $B = f^{-1}\{ b \}$. Then $f[P\backslash B] = f[P]\backslash b$.

Proof. An elementary calculation shows that (t, X) is a failure of $f[P\backslash B]$ iff one of the following conditions holds:

$$\begin{array}{ll} (1) & \exists s.t = f(s\backslash B) \,\&\, (s, f^{-1}(X) \cup B) \in P \\ (2) & \exists s.t = f(s\backslash B) \,\&\, \forall n \exists w \in B^n.sw \in \text{traces}(P). \end{array}$$

Similarly, (t, X) is a failure of $f[P]\backslash b$ iff either of the following hold:

$$\begin{array}{ll} (1') & \exists s.t = f(s)\backslash b \,\&\, (s, f^{-1}(X \cup \{ b \})) \in P \\ (2') & \exists s.t = f(s)\backslash b \,\&\, \forall n.f(s)b^n \in \text{traces}(P). \end{array}$$

But $B = f^{-1}\{ b \}$, so

$$f(s)\backslash b = f(s\backslash B),$$
$$f^{-1}(X \cup \{ b \}) = f^{-1}(X \cup B).$$

Hence we see that (1) $\Leftrightarrow$ (1'). Since $w \in B^n \Rightarrow f(sw) = f(s)b^n$, we also have (2) $\Leftrightarrow$ (2'). Finally, by asssumption that f only identifies finitely many events, for any trace u there can only be finitely many s such that $f(s) = u$. Hence, if $ub^n \in \text{traces}(P)$ for all n, there must be some trace s with $f(s) = u$ and $\forall n \exists w \in B^n.sw \in \text{traces}(P)$. This shows that (2') $\Rightarrow$ (2), completing the proof. ∎

Theorem 2.11.8:

$$\begin{array}{lll} \text{(i)} & f[P \,\Box\, Q] = f[P] \,\Box\, f[Q] & \\ \text{(ii)} & f[P \,|||\, Q] = f[P] \,|||\, f[Q] & \\ \text{(iii)} & f[P;Q] = f[P];f[Q] & \text{(provided } f^{-1}(\surd) = \{ \surd \}.) \\ \text{(iv)} & f[P \,\|\, Q] = f[P] \,\|\, f[Q] & \text{(provided } f \text{ is injective.)} \end{array}$$

Proof. We give details for (iv), the other cases being similar. Suppose f is injective. It is clear that the extended function $f : \Sigma^* \to \Sigma^*$ is also injective. Let $(t, X) \in f[P \,\|\, Q]$. There is a unique trace s of $P \,\|\, Q$ such that $f(s) = t$, and for this trace we know that

$$(s, f^{-1}(X)) \in P \,\|\, Q.$$

This means that there are sets Y' and Z' such that

$$(s, Y') \in P, \quad (s, Z') \in Q, \quad Y' \cup Z' = f^{-1}(X).$$

Let $Y = f(Y')$ and $Z = f(Z')$, so that

$$Y' = f^{-1}(Y), \quad Z' = f^{-1}(Z).$$

Then we have

$$\begin{aligned}(s, Y') \in P &\Rightarrow (f(s), Y) \in f[P] \\ (s, Z') \in Q &\Rightarrow (f(s), Z) \in f[Q] \\ &\Rightarrow (f(s), Y \cup Z) \in f[P] \,||\, f[Q].\end{aligned}$$

But $f(s) = t$ and

$$Y \cup Z = f(Y') \cup f(Z') = f(Y' \cup Z') = f(f^{-1}(X)) = X$$

(note that this holds by hypothesis that f is injective.) Hence,

$$(t, X) \in f[P] \,||\, f[Q].$$

This shows that every failure of $f[P \,||\, Q]$ is also a failure of $f[P] \,||\, f[Q]$; the argument can be reversed to establish the converse. ∎

Putting together the results of Theorems 2.11.5 and 2.11.8, we can deduce the effect of a renaming on a guarded process:

Corollary: For the process $P = \Box_{b \in B}(b \rightarrow P_b)$,

$$f[P] = \Box_{b \in B}(f(b) \rightarrow f[P_b]).$$

∎

Examples. Assuming throughout that a and b are distinct events, define the alphabet transformation f by:

$$\begin{aligned}f(a) &= b \\ f(b) &= b \\ f(x) &= x \quad (x \neq a, b).\end{aligned}$$

Example 1. Let P and Q be the processes

$$\begin{aligned}P &= (a \rightarrow (c \rightarrow \text{STOP})), \\ Q &= (b \rightarrow \text{STOP}),\end{aligned}$$

and let $R = P \,\Box\, Q$; then R is deterministic and has initials $\{a, b\}$. Applying the alphabet transformation f, we have

$$\begin{aligned}f[P] &= f[a \rightarrow (c \rightarrow \text{STOP})] \\ &= b \rightarrow f[c \rightarrow \text{STOP}] \\ &= b \rightarrow (c \rightarrow f[\text{STOP}]) \\ &= b \rightarrow (c \rightarrow \text{STOP}),\end{aligned}$$

and similarly

$$\begin{aligned} f[Q] &= f[b \to \text{STOP}] \\ &= b \to \text{STOP}. \end{aligned}$$

But, using Theorem 2.11.8,

$$\begin{aligned} f[R] &= f[P \,\Box\, Q] \\ &= f[P] \,\Box\, f[Q] \\ &= (b \to (c \to \text{STOP})) \,\Box\, (b \to \text{STOP}) \\ &= b \to ((c \to \text{STOP}) \sqcap \text{STOP}), \end{aligned}$$

so that, although R is deterministic, $f[R]$ is not. Indeed, once $f[R]$ has performed b either of the component processes may have progressed:

$$\begin{aligned} f[R] \,\underline{\text{after}}\, \langle b \rangle &= f[P \,\underline{\text{after}}\, \langle a \rangle] \sqcap f[Q \,\underline{\text{after}}\, \langle b \rangle] \\ &= (c \to \text{STOP}) \sqcap \text{STOP}. \end{aligned}$$

Finally, putting $B = f^{-1}(b) = \{ a, b \}$, we have

$$\begin{aligned} f[R] \backslash b &= (b \to (c \to \text{STOP})) \backslash b \sqcap \text{STOP} \backslash b \\ &= (c \to \text{STOP}) \sqcap \text{STOP} \\ R \backslash B &= (c \to \text{STOP}) \sqcap \text{STOP}. \end{aligned}$$

This illustrates the identity of Theorem 2.11.7.

Example 2. Let R be the recursively defined process

$$R = (a \to R) \sqcap (b \to R).$$

Then for the alphabet transformation f defined above, we have

$$\begin{aligned} f[R] &= f[a \to R] \sqcap f[b \to R] \\ &= (b \to f[R]) \sqcap (b \to f[R]) \\ &= (b \to f[R]). \end{aligned}$$

This equation for $f[R]$ has a unique solution, the deterministic process $\text{det}(b^*)$. Although R could decide autonomously between executing a or b at each stage, the renaming identifies these two events; $f[R]$ cannot refuse to perform the renamed event.

12. Inverse image

For an alphabet transformation f, the process $f[P]$ can be thought of as a *direct* image of P. Now we introduce a kind of *inverse image,* which we will denote $f^{-1}[P]$. This is to be a process which can perform event c whenever P could have performed $f(c)$. In addition, $f^{-1}(P)$ can refuse an event c only if P could have refused the image $f(c)$ of c : in general $f^{-1}[P]$ will be able to refuse a set X just in case P can refuse the image set $f(X)$.

Definition 2.12.1:

$$f^{-1}[P] = \{\, (s, X) \mid (f(s), f(X)) \in P \,\}.$$

Theorem 2.12.2: If P is a process so is $f^{-1}[P]$, and it is uniquely characterised by the conditions:

(i) $\quad \text{traces}(f^{-1}[P]) = f^{-1}(\text{traces}(P))$
(ii) $\quad \text{refusals}(f^{-1}[P]) = \{\, X \mid f(X) \in \text{refusals}(P) \,\}$
(iii) $\quad f^{-1}[P] \underline{\text{after}}\ s = f^{-1}[P \underline{\text{after}}\ f(s)].$

Proof. Straightforward. ∎

Theorem 2.12.3: Let ι be the identity transformation and f, g be any alphabet transformations. Then for all processes P,

(i) $\quad \iota^{-1}[P] = P$
(ii) $\quad (f \circ g)^{-1}[P] = g^{-1}[f^{-1}[P]].$

Theorem 2.12.4:

(i) $\quad f^{-1}[c \to P] = \Box\{\, (b \to f^{-1}[P]) \mid f(b) = c \,\}$
(ii) $\quad f^{-1}[\text{det}(T)] = \text{det}(f^{-1}(T))$
(iii) $\quad f^{-1}[\text{chaos}(T)] = \text{chaos}(f^{-1}(T))$
(iv) $\quad f^{-1}[\text{STOP}] = \text{STOP}.$

Theorem 2.12.5: The inverse image operation is monotone, distributive and continuous:

(i) $\quad P \sqsubseteq Q \Rightarrow f^{-1}[P] \sqsubseteq f^{-1}[Q]$
(ii) $\quad f^{-1}[\bigcup \mathcal{D}] = \bigcup\{\, f^{-1}[P] \mid P \in \mathcal{D} \,\}$
(iii) $\quad f^{-1}[\bigcap_n P_n] = \bigcap_n f^{-1}[P_n],$

for all nonempty sets $\mathcal{D}$ and chains $\{\, P_n \mid n \geq 0 \,\}$.

Theorem 2.12.6: Let f be an alphabet transformation and let $b \in \text{range}(f)$. Let $B = f^{-1}(b)$. Then for all processes P,

$$f^{-1}[P \backslash b] = f^{-1}[P] \backslash B.$$

Proof. As in the proof of Theorem 2.11.7, we see that (s, X) is failure of $f^{-1}[P]\backslash B$ iff one of the following conditions holds:

$$(1) \quad \exists u.s = u\backslash B \;\&\; (u, X \cup B) \in f^{-1}[P]$$
$$(2) \quad \exists u.s = u\backslash B \;\&\; \forall n \exists w \in B^n.uw \in \text{traces}(f^{-1}[P]).$$

Similarly, (s, X) is a failure of $f^{-1}[P\backslash b]$ iff

$$\text{either}(1') \quad \exists t.t\backslash b = f(s) \;\&\; (t, f(X) \cup B) \in P$$
$$\text{or} \quad (2') \quad \exists t.t\backslash b = f(s) \;\&\; \forall n\, tb^n \in \text{traces}(P).$$

But $f(X \cup B) = f(X) \cup \{b\}$, since b is in the range of f. Since $w \in B^n \Rightarrow f(uw) = f(u)b^n$, it follows that (1) $\Leftrightarrow$ (1′) and (2) $\Leftrightarrow$ (2′). That completes the proof. ∎

The proof of this result on hiding can again be adapted to the alternate hiding operator.

Theorem 2.12.7:

$$\text{(i)} \quad f^{-1}[P \,\Box\, Q] = f^{-1}[P] \,\Box\, f^{-1}[Q]$$
$$\text{(ii)} \quad f^{-1}[P \,|||\, Q] = f^{-1}[P] \,|||\, f^{-1}[Q]$$
$$\text{(iii)} \quad f^{-1}[P;Q] = f^{-1}[P];f^{-1}[Q] \quad (\text{if } f^{-1}(\surd) = \{\surd\}).$$
$$\text{(iv)} \quad f^{-1}[P \,\|\, Q] = f^{-1}[P] \,\|\, f^{-1}[Q].$$

Again, as with direct alphabet transformations, we deduce the effect of an inverse transformation on a guarded process:

Corollary: For a process $\Box_{b \in B}(b \to P_b)$

$$f^{-1}[P] = \Box_{c \in f^{-1}(B)}(c \to f^{-1}[P_{f(c)}])$$
$$= \Box\{\, (c \to f^{-1}[P_b]) \mid b \in B \;\&\; f(c) = b \,\}.$$

Proof. By Theorem 2.12.7(i) and 2.12.4(i). ∎

Examples. Revisit the examples of the previous section. Let f be the alphabet transformation used there:

$$f(a) = b$$
$$f(b) = b$$
$$f(x) = x \quad (x \neq a, b).$$

Example 1. The process P performs first a then c.

$$P = a \to (c \to \text{STOP})$$
$$f^{-1}[P] = \text{STOP},$$

because $f^{-1}(a) = \emptyset$. The inverse image of P is unable to perform any event, because no event initially possible for P is in the range of f. Notice that this example shows that the condition on the hidden event in Theorem 2.12.6 is necessary: here a is not in the range of f, and we have

$$\begin{aligned} f^{-1}[P\backslash a] &= f^{-1}[c \rightarrow \text{STOP}] \\ &= (c \rightarrow \text{STOP}) \\ &\neq f^{-1}[P]\backslash f^{-1}(a) \\ &= f^{-1}[P] \\ &= \text{STOP}. \end{aligned}$$

The process Q can initially perform an event which is associated to two previously distinct events by f :

$$\begin{aligned} Q &= b \rightarrow \text{STOP} \\ f^{-1}[Q] &= (a \rightarrow \text{STOP}) \,\Box\, (b \rightarrow \text{STOP}), \end{aligned}$$

because $f^{-1}(b) = \{ a, b \}$. For combinations of P and Q we get:

$$\begin{aligned} f^{-1}[P \,\Box\, Q] &= f^{-1}[P] \,\Box\, f^{-1}[Q] \\ &= (a \rightarrow \text{STOP}) \,\Box\, (b \rightarrow \text{STOP}), \\ f^{-1}[P \sqcap Q] &= f^{-1}[P] \sqcap f^{-1}[Q] \\ &= ((a \rightarrow \text{STOP}) \,\Box\, (b \rightarrow \text{STOP})) \sqcap \text{STOP}. \end{aligned}$$

Unlike $P \,\Box\, Q$, the process $P \sqcap Q$ can refuse the set $\{ b \} = f(\{ a, b \})$. Hence, $f^{-1}[P \sqcap Q]$ can refuse $\{ a, b \}$ whereas $f^{-1}[P \,\Box\, Q]$ cannot.

Notice also that we have

$$\begin{aligned} f[f^{-1}[P]] &= f[\text{STOP}] = \text{STOP}, \\ f^{-1}[f[Q]] &= f^{-1}[Q] = (a \rightarrow \text{STOP}) \,\Box\, (b \rightarrow \text{STOP}). \end{aligned}$$

Thus in general P and $f[f^{-1}[P]]$ are incomparable and so are Q and $f^{-1}[f[Q]]$.

Inverse and direct renamings.

Finally in this section let us consider the relationship between direct and inverse image operations. For any function $f : \Sigma \rightarrow \Sigma$, and any set X, we know that

$$\begin{aligned} &\text{(i)} && X \subseteq f^{-1}(f(X)) \\ &\text{(ii)} && f(f^{-1}(X) \subseteq X. \end{aligned}$$

Moreover, these inclusions are identities when the function is, respectively, injective and surjective. Indeed, when f is a bijection, the inverse function

$f^{-1} : \Sigma \to \Sigma$ will also be an alphabet transformation. The next result states that, for a bijective alphabet transformation f the inverse image of P under f is the same process as the direct image under f^{-1}. It also gives the analogous results to (i) and (ii) above.

Theorem 2.12.8:

(i) When f is injective, $f^{-1}[f[P]] = P$.
(ii) When f is surjective, $f[f^{-1}[P]] = P$.
(iii) When f is bijective and $g : \Sigma \to \Sigma$ is its inverse function, $g[P] = f^{-1}[P]$.

Proof. Elementary properties of functions and their inverses. ∎

Chapter 3

Implementations

0. Introduction

So far we have presented a domain of processes, equipped with a partial order which corresponds in a precise way to a measure of nondeterminism. Processes were specified as sets of *failures,* in terms of the actions they could perform and refuse to perform. Each failure of a process represents part of a possible behaviour of that process: a sequence of actions which the process *may* engage in, and a set of events which the process *may* refuse to perform on the next step. We saw that it was possible to define a wide variety of processes and functions on processes. In each case we explained the behaviour of a constructed process in terms of the behaviours of its constituents.

This chapter considers an alternative, but equivalent, formulation of the notion of process. This formulation has strong similarities with the work of Kennaway [K1,2], which will be explored more fully in Chapter 6. We introduce a simple abstract notion of *implementation.* Our attitude is that, in order to *implement* a process P, one must *resolve* the nondeterministic decisions of the process. This abstract view of implementation amounts to saying that an implementation of a process is just a possible deterministic behaviour of that process. Each process will be characterised by its set of possible implementations. It turns out that implementation sets have some elegant properties. The idea extends to process operations: an implementation of a function on processes should be a function which maps implementations of argument processes to implementations of results.

1. Implementing processes

The task of *implementing* a process P is understood to involve making a set of decisions to resolve the nondeterminism inherent in P. Implementations of a process will therefore represent possible *deterministic* behaviours of that process. Since a deterministic process is uniquely represented by its trace set, and this set always forms a tree, we will identify an implementation with the corresponding tree. A particular tree T is a valid implementation of P if and only if it is possible for P or its implementor to decide to allow all traces in T. This situation is possible if and only if P is at least as nondeterministic as the deterministic process with trace set T, *i.e.*, , if $P \sqsubseteq \det(T)$. Informally we may think of implementing a process by making a sequence of choices of refusal set, at each stage remaining consistent with the failures of the process. It will be convenient to introduce a simple notation for trees. A tree can be described by giving its initial arcs and the subtrees rooted at these arcs. We write

$$(x : B \to T(x))$$

for the tree with initial arcs labelled by the events in B and with subtrees $T(c)$ hanging from the arc labelled c, for each $c \in B$. In this notation, x is a bound variable; we are free to use any other bound variable without loss of generality. We also use the abbreviation S^0 for initials(S). Thus the general tree T can be expressed in the form

$$T = (x : T^0 \to T \underline{\text{after}} \langle x \rangle).$$

We will understand a term of the form $(x : \emptyset \to T(x))$ to represent the trivial tree NIL, as this tree has empty initial set.

The map

$$\text{imp} : \text{PROC} \to \mathcal{P}(\text{TREE})$$

maps a process to its implementation set. The map

$$\text{proc} : \mathcal{P}(\text{TREE}) \to \text{PROC}$$

produces the most nondeterministic process having a given implementation set.

Definition 3.1.1: Let P be a process and $\mathcal{T}$ be a set of trees.

(i) $\text{imp}(P) = \{ T \mid P \sqsubseteq \det(T) \}$

(ii) $\text{proc}(\mathcal{T}) = \bigcup \{ \det(T) \mid T \in \mathcal{T} \}$.

Notice that the tree $T = (x : B \to T(x))$ is a possible implementation of P just in case P can refuse every finite subset of the complement $\overline{B}$ and for each $c \in B$ the tree $T(c)$ implements $P \underline{\text{after}} \langle c \rangle$.

Lemma 3.1.2:

$$\begin{array}{ll} \text{(i)} & \text{traces}(P) \in \text{imp}(P) \\ \text{(ii)} & \text{imp}(\text{det}(T)) = \{T\} \\ \text{(iii)} & \text{proc}(\text{imp}(P)) = P \\ \text{(iv)} & \text{imp}(\text{proc}(\mathcal{T})) \supseteq \mathcal{T}. \end{array}$$

Proof. These are restatements of earlier results in the new notation. ∎

Corollary 3.1.3:

$$P \sqsubseteq Q \quad \Leftrightarrow \quad \text{imp}(P) \supseteq \text{imp}(Q).$$

The last result shows that no information is lost when identifying a process with its implementation set. Indeed, the superset ordering on implementation sets coincides with the ordering on processes. The notion of a process as a set of implementations could therefore have been used as an alternative basis for construction of the domain of processes. It is clear that the sets of trees which serve as implementation sets of processes satisfy a simple condition. If $\mathcal{T} = \text{imp}(P)$, then we have

$$\begin{array}{rl} & \text{proc}(\mathcal{T}) = P \\ \Rightarrow & \text{imp}(\text{proc}(\mathcal{T})) = \text{imp}(P) \\ \Rightarrow & \text{imp}(\text{proc}(\mathcal{T})) = \mathcal{T}. \end{array}$$

This implies that any implementation set $\mathcal{T}$ of trees satisfies the condition $\text{imp}(\text{proc}(\mathcal{T})) = \mathcal{T}$. It is obvious that any set of trees satisfying this condition is an implementation set. Let us call such sets of processes *closed.*

Definition 3.1.4: A set $\mathcal{T}$ of trees is *closed* if and only if

$$\mathcal{T} = \text{imp}(\text{proc}(\mathcal{T})).$$

We will refer to $\text{imp}(\text{proc}(\mathcal{T}))$ as the *closure* of $\mathcal{T}$. Arguing from the definitions above, we see that a tree U belongs to the closure $\text{imp}(\text{proc}(\mathcal{T}))$ just in case

$$\text{proc}(\mathcal{T}) \sqsubseteq \text{det}(U),$$

and this holds if and only if for each $u \in U$ and each finite set of events X such that $(u, X) \in \text{det}(U)$ there is a tree T in $\mathcal{T}$ such that $(u, X) \in \text{det}(T)$. Equivalently, using the definition of $\text{det}(\cdot)$, $U \in \text{imp}(\text{proc}(\mathcal{T}))$ if and only if for all $u \in U$ and $X \in p\Sigma$,

$$uX \cap U = \emptyset \Rightarrow \exists T \in \mathcal{T}.\, u \in T \;\&\; uX \cap T = \emptyset.$$

This can again be rewritten: $U \in \text{imp}(\text{proc}(\mathcal{T}))$ iff for all $u \in U$ there does not exist a finite set X such that

$$uX \cap U = \emptyset \;\&\; \forall T \in \mathcal{T}.\, (u \in T \Rightarrow uX \cap T \neq \emptyset).$$

Finally, if we define a *cross–section* of a collection of sets to be a set which has a nontrivial intersection with every member of the collection, we see that the above condition on U is equivalent to requiring that for each $u \in U$ the collection of sets

$$\{ (T \underline{\text{after}}\, u)^0 \mid u \in T \ \&\ T \in \mathcal{T} \}$$

has no finite cross–section drawn from the set $\overline{(U \underline{\text{after}}\, u)^0}$. Notice that if the set $\{ T^0 \mid T \in \mathcal{T} \}$ has itself no finite cross-sections then the trivial tree NIL must be included in the closure of $\mathcal{T}$. A similar result was proved by Kennaway [K], except that in his formulation of processes as sets of trees any set $\mathcal{T}$ containing NIL becomes identified with all other such sets: whenever $\mathcal{T}$ contains NIL Kennaway takes the closure of $\mathcal{T}$ to be the entire set TREE of all trees. This fact will reappear in Chapter 6, where a more detailed discussion of Kennaway's work and its connections with this work will be given.

Recall that the alphabet of a process is the smallest set of events A such that the events in all traces of P are drawn from A. By analogy, we define the alphabet of a set of trees $\mathcal{T}$ to be the alphabet of the process $\text{proc}(\mathcal{T})$. The following result gives an alternative characterization of closed sets of trees in the case when their alphabet is finite. First some definitions.

Definition 3.1.5: The *convex hull* $\text{conv}(\mathcal{T})$ of a set $\mathcal{T}$ of trees is the set of all trees T such that, for all t,

$$t \in T \Rightarrow \exists S \in \mathcal{T} . t \in S \ \&\ \text{initials}(S \underline{\text{after}}\, t) \subseteq \text{initials}(T \underline{\text{after}}\, t).$$

We say that $\mathcal{T}$ is *convex* if $\mathcal{T} = \text{conv}(\mathcal{T})$.

By definition, $\mathcal{T} \subseteq \text{conv}(\mathcal{T})$, and $\text{conv}(\mathcal{T})$ is closed under arbitrary non-empty unions. In particular, the tree $\bigcup \mathcal{T}$ is the largest member of $\text{conv}(\mathcal{T})$. This explains our use of the term *convex hull* where it might have seemed more natural from the given definition to use *upward* closure. Putting U for the union of the trees in $\mathcal{T}$, we see that for all $T \in \text{conv}(\mathcal{T})$ and all $t \in T$

$$\exists S \in \mathcal{T} . t \in S \cap U \ \&\ \text{initials}(S \underline{\text{after}}\, t) \subseteq \text{initials}(T \underline{\text{after}}\, t) \subseteq \text{initials}(U \underline{\text{after}}\, t).$$

This version corresponds more closely to the normal formulation of convex closure.

Lemma 3.1.6:

(i) $\text{conv}(\text{conv}(\mathcal{T})) = \text{conv}(\mathcal{T})$

(ii) $\text{conv}(\mathcal{T}) \subseteq \text{imp}(\text{proc}(\mathcal{T}))$.

Proof. Elementary. ∎

Lemma 3.1.7: If the alphabet of $\mathcal{T}$ is finite ,

$$\text{imp}(\text{proc}(\mathcal{T})) \subseteq \text{conv}(\mathcal{T}).$$

Proof. Elementary. ∎

Corollary: If the alphabet of $\mathcal{T}$ is finite, then

$$\mathcal{T} \text{ is convex} \Leftrightarrow \mathcal{T} = \text{imp}(\text{proc}(\mathcal{T})).$$

Examples.

Example 1. STOP has a single implementation, the trivial tree:

$$\text{imp}(\text{STOP}) = \{\,\text{NIL}\,\}$$
$$\text{proc}(\{\,\text{NIL}\,\}) = \text{STOP}.$$

Example 2. CHAOS can be implemented by any tree whatsoever:

$$\text{imp}(\text{CHAOS}) = \text{TREE}(\Sigma)$$
$$\text{proc}(\text{TREE}(\Sigma)) = \text{CHAOS}.$$

Example 3. Let $P = (a \to \text{STOP})$ and $Q = (b \to \text{STOP})$.

$$\text{imp}(P) = \{\,a\text{NIL}\,\}$$
$$\text{imp}(Q) = \{\,b\text{NIL}\,\}$$
$$\text{imp}(P \sqcap Q) = \{\,a\text{NIL}, b\text{NIL}, a\text{NIL} + b\text{NIL}\,\}.$$

Putting $\mathcal{T} = \{\,a\text{NIL}, b\text{NIL}\,\}$ we have $\text{proc}(\mathcal{T}) = P \sqcap Q$. In this example, $\mathcal{T}$ is not convex and $\text{imp}(\text{proc}(\mathcal{T})) = \text{conv}(\mathcal{T}) \neq \mathcal{T}$.

Example 4.. Let $P = (a \to \text{STOP}) \,\Box\, (b \to \text{STOP})$ and $Q = \text{STOP}$. Let $\mathcal{T} = \{\,\text{NIL}, a\text{NIL} + b\text{NIL}\,\}$, so that $\text{proc}(\mathcal{T}) = P \sqcap Q$. By a simple calculation we have

$$\text{conv}(\mathcal{T}) = \{\,\text{NIL}, a\text{NIL}, b\text{NIL}, a\text{NIL} + b\text{NIL}\,\},$$
$$\text{imp}(P \sqcap Q) = \{\,\text{NIL}, a\text{NIL}, b\text{NIL}, a\text{NIL} + b\text{NIL}\,\}.$$

Indeed, using distributivity of $\Box$ over $\sqcap$ and vice versa, we have the identity

$$P \sqcap Q = \text{STOP} \sqcap (a \to \text{STOP}) \sqcap (b \to \text{STOP}) \sqcap (x : \{\,a, b\,\} \to \text{STOP}).$$

In this example again $\mathcal{T}$ is not closed.

Example 5. Assume that each natural number n denotes an event, so that the alphabet Σ is infinite. Let P be the process

$$P = \sqcap\{(n \rightarrow \text{STOP}) \mid n \geq 0\}.$$

Let $\mathcal{T} = \{ n\text{NIL} \mid n \geq 0 \}$, so that $P = \text{proc}(\mathcal{T})$.

Notice that $\mathcal{T}$ has no finite cross–section. The convex hull of $\mathcal{T}$ consists of those trees which can initially accept any of a non-empty set of natural numbers and then deadlock. This means that the tree NIL, which cannot perform any event, does not belong to $\text{conv}(\mathcal{T})$. However, P can refuse any finite set of events, by choosing to behave like $(n \rightarrow \text{STOP})$ for a sufficiently large integer n. Thus the process STOP is a possible implementation of P. In fact, we have:

$$\text{conv}(\mathcal{T}) \neq \text{imp}(\text{proc}(\mathcal{T})) = \text{conv}(\mathcal{T}) \cup \{ \text{NIL} \}.$$

This example illustrates the characterization of closure discussed above when the alphabet is not finite.

2. Implementing functions

A function $F : \mathrm{PROC}^n \to \mathrm{PROC}$ represents a method of constructing processes. Given argument processes $P_1, \ldots, P_n$, applying the function produces the result process $F(P_1, \ldots, P_n)$. Extending the ideas of the previous section, it seems natural to define an *implementation* of F to be an operation f on *implementations* of processes which is consistent with F: when applied to implementations $T_1, \ldots, T_n$ of $P_1, \ldots P_n$, f should always produce an implementation of $F(P_1, \ldots, P_n)$. Thus, an implementation of F will be a function $f : \mathrm{TREE}^n \to \mathrm{TREE}$ such that

$$f(T_1, \ldots, T_n) \in \mathrm{imp}(F(P_1, \ldots, P_n)),$$

whenever $T_i \in \mathrm{imp}(P_i)$ for $i = 1 \ldots n$. Equivalently, f is an implementation of F iff, for all trees $T_1, \ldots, T_n$,

$$f(T_1, \ldots, T_n) \in \mathrm{imp}(F(\det(T_1), \ldots, \det(T_n))).$$

This means that the various implementations of F are determined by the effects of F on deterministic processes. Just as any set of trees $\mathcal{T}$ determines a process $\mathrm{proc}(\mathcal{T})$, any set $\mathcal{F}$ of functions on trees determines a function $\mathrm{fun}(\mathcal{F})$ on processes. For simplicity we consider the case when the functions are unary; the generalisation to functions of several variables should be clear.

Definition 3.2.1: Let F be a function on PROC and let $\mathcal{F}$ be a non-empty set of functions on trees.

(i) $\mathrm{imp}(F) = \{ f \mid \forall T. f(T) \in \mathrm{imp}(F(\det(T))) \}$

(ii) $\mathrm{fun}(\mathcal{F})(P) = \bigcup \{ \det(f(T)) \mid f \in \mathcal{F} \;\&\; T \in \mathrm{imp}(P) \}$.

Recall that the traces of any process form a tree, and that this tree is a possible implementation of that process. Correspondingly, a generally applicable method of implementing a process operation F is to take traces: for any process operation F, there is an implementation $\hat{F}$ defined by

$$\hat{F}(T) = \mathrm{traces}(F(\det(T))).$$

When F is a deterministic operation, $\hat{F}$ will be the *only* possible implementation; for this reason we will call $\hat{F}$ the *deterministic* implementation of F.

Lemma 3.2.2: Let F be a process operation.

(i) $\hat{F} \in \mathrm{imp}(F)$

(ii) $\mathrm{imp}(F) = \{ \hat{F} \}$, if F is deterministic.

Proof. By Lemma 3.1.2. ∎

By definition, the process operations F and fun(imp(F)) agree on all deterministic arguments. This agreement will extend to all arguments only if F is determined uniquely by its effect on deterministic processes alone in the following way:

$$F(P) = \bigsqcup\{ F(\mathrm{det}(T)) \mid T \in \mathrm{imp}(P) \}.$$

In view of the identity

$$P = \bigsqcup\{ \mathrm{det}(T) \mid T \in \mathrm{imp}(P) \},$$

the above condition amounts to a weak form of distributivity. Not every process operation satisfies this condition. We will call functions which are uniquely determined by their action on deterministic processes *implementable.*

Definition 3.2.3: A process operation F is *implementable* iff for all processes P,

$$F(P) = \bigsqcup\{ F(\mathrm{det}(T)) \mid T \in \mathrm{imp}(P) \}.$$

Lemma 3.2.4: The following three conditions on a process operation F are equivalent:

(i) F is implementable
(ii) $F = \mathrm{fun}(\mathrm{imp}(F))$
(iii) $F(P) = \mathrm{proc}(\{ f(T) \mid f \in \mathrm{imp}(F) \,\&\, T \in \mathrm{imp}(P) \})$.

Proof. Straightforward. ∎

Examples.

Example 1. The function F : PROC → PROC defined by:

$$F(P) = P, \text{ if P is deterministic,}$$
$$= \text{CHAOS, otherwise,}$$

is not implementable. The only possible implementation of F is the identity function on trees, and fun(imp(F)) is the identity function on processes, and not equal to F.

Example 2. Any fully distributive operation is implementable, but not every implementable operation is fully distributive.

3. Implementing CSP operations

In this section we show how all of the CSP operations defined earlier could have been given in terms of implementations. The first fact to note is that all these operations are indeed implementable; this is hardly surprising, since most of them were shown to be fully distributive. In the only non-distributive case, that of hiding, we state the fact which renders the hiding operations implementable. Although neither hiding operation is fully distributive they *do* distribute through implementation sets in the required way. To facilitate comparison with the earlier definitions, we consider the CSP operations in the same order as before.

0. Inaction.

STOP is a deterministic process. It has a single implementation NIL, its trace set.

1. Prefixing.

Prefixing is a deterministic operation. If we want to construct an implementation for $(c \to P)$ from an implementation T of P, we simply prefix the event c to T, producing the tree $(c \to T)$. The earlier definition of prefixing was:

$$(c \to P) = \{\, (\langle\rangle, X) \mid c \notin X \,\} \cup \{\, (\langle c\rangle s, X) \mid (s, X) \in P \,\}.$$

The new definition is simple.

$$(c \to T) = \{\, \langle\rangle \,\} \cup \{\, \langle c\rangle t \mid t \in T \,\}$$
$$\mathrm{imp}(c \to P) = \{\, (c \to T) \mid T \in \mathrm{imp}(P) \,\}.$$

It is easy to see that the two definitions agree.

2. Nondeterministic choice.

The process $P \sqcap Q$ can behave either like P or like Q. It can be implemented by selecting either an implementation of P or an implementation of Q. In other words, the two projection functions Π_1, Π_2 from TREE^2 to TREE are implementations of the nondeterministic choice operation. The earlier definition was simply:

$$P \sqcap Q = P \cup Q.$$

In terms of implementations, the required definition is:

$$P \sqcap Q = \mathrm{proc}(\mathrm{imp}(P) \cup \mathrm{imp}(Q))$$

or, equivalently,

$$P \sqcap Q = \text{fun}\{\,\Pi_1, \Pi_2\,\}(P, Q).$$

Again it is left to the reader to verify that the definitions agree. The deterministic implementation of this operation is just union of trees, because

$$\text{traces}(P \sqcap Q) = \text{traces}(P) \cup \text{traces}(Q).$$

3. Conditional choice.

Let S and T be implementations of P and Q respectively. We wish to build an implementation of $P \,\Box\, Q$. If there is no event initially possible for both S and T, we may simply choose the union $S \cup T$ to implement $P \Box Q$; this allows the choice between behaving like P and behaving like Q to be made on the first step. When $S^0 \cap T^0 = \emptyset$ this choice can be made unambiguously because any potential first event will be possible for at most one of the two trees. In the general case, when the initials of S and T have some event in common, we are permitted to implement $P \,\Box\, Q$ by giving priority to one tree rather than the other on the common initial events. This amounts to forming a *biased* union of the two trees. When the trees have disjoint initials, of course, the biased union is simply the usual union. We use the notation $S \,\underline{\text{else}}\, T$ for the union biased in favour of S. Firstly we restate the earlier definition of $\Box$, for comparison with the following implementation version.

$$P \,\Box\, Q = \{\,(\langle\rangle, X) \mid (\langle\rangle, X) \in P \cap Q\,\} \cup \{\,(s, X) \mid s \neq \langle\rangle \,\&\, (s, X) \in P \cup Q\,\}$$
$$S \,\underline{\text{else}}\, T = S \cup (x : T^0 - S^0 \to T(x))$$
$$P \,\Box\, Q = \text{proc}\{\,S \,\underline{\text{else}}\, T, T \,\underline{\text{else}}\, S \mid S \in \text{imp}(P) \,\&\, T \in \text{imp}(Q)\,\}.$$

The most elegant way to establish this fact from the earlier definition of $\Box$ is to begin from the easy identity

$$\det(S) \,\Box\, \det(T) = \det(S \,\underline{\text{else}}\, T) \sqcap \det(T \,\underline{\text{else}}\, S),$$

which is easily deducible from Lemma 2.3.8.

4. Parallel composition.

This is a deterministic operation. The only implementation of a parallel composition of two deterministic processes $\det(S) \,||\, \det(T)$ is $S \cap T$, obtained by intersecting the implementation of $\det(S)$ with the implementation of $\det(T)$. The earlier definition was:

$$P \,||\, Q = \{\,(s, X \cup Y) \mid (s, X) \in P \,\&\, (s, Y) \in Q\,\}.$$

In terms of implementations we merely require:

$$P \,||\, Q = \text{proc}\{\,S \cap T \mid S \in \text{imp}(P) \,\&\, T \in \text{imp}(Q)\,\}.$$

This identity is easily verified.

5. Interleaving.

$P|||Q$ is a process whose traces are obtained from those of P and Q by interleaving. Given two implementations S and T of P and Q, the deterministic implementation of $P|||Q$ would be the tree $S|||T$. However, in this form of interleaving a nondeterministic choice is allowed whenever an event is possible for both constituent processes. Indeed, when the processes being interleaved are

$$P = (x : B \to P(x))$$
$$Q = (x : C \to Q(x))$$

we have

$$P|||Q = (x : B \to (P(x)|||Q)) \,\Box\, (x : C \to (P|||Q(x))).$$

It is easy to see that this identity suffices to define interleaving on deterministic processes, by recursion using the definition of $\Box$. This suggests a recursive definition of the set of possible implementations of $\mathrm{det}(S)|||\mathrm{det}(T)$. The following set of trees is what is required:

Definition 3.3.5.1: For any trees S and T the set $\mathrm{ndmerge}(S,T)$ of all *nondeterministic merges* is defined by

$$\begin{array}{rrl} U \in \mathrm{ndmerge}(S,T) \Leftrightarrow & \text{either} & U = (x : S^0 \to U(x)) \,\underline{\text{else}}\, (x : T^0 \to V(x)) \\ & \text{or} & U = (x : T^0 \to V(x) \,\underline{\text{else}}\, (x : S^0 \to U(x)) \\ & \text{where} & \forall x \in S^0.\quad U(x) \in \mathrm{ndmerge}(S(x), T) \\ & \text{and} & \forall x \in T^0.\quad V(x) \in \mathrm{ndmerge}(S, T(x)). \end{array}$$

This definition is symmetric in S and T. In the special case when S is trivial, we interpret the definition as reducing to $\{T\}$:

$$\mathrm{ndmerge}(\mathrm{NIL}, T) = \{T\}.$$

In order to check whether or not a finite tree belongs to $\mathrm{ndmerge}(S,T)$ this recursive definition provides us with a terminating algorithm. For infinite trees we are required to check that the initial set at every node of the tree is consistent with the definition; the nodes at each level in the tree can be checked off successively, since the definition of ndmerge uses recursion on lower levels of trees. This informal idea could be made rigorous in the obvious way.

The following results are easy to prove, giving a characterisation of the interleaving operation in terms of implementations.

$$\mathrm{det}(S)|||\mathrm{det}(T) = \mathrm{proc}(\mathrm{ndmerge}(S,T))$$
$$P|||Q = \mathrm{proc}(\bigcup\{\, \mathrm{ndmerge}(S,T) \mid S \in \mathrm{imp}(P) \,\&\, T \in \mathrm{imp}(Q)\,\}).$$

6. Hiding.

Suppose we begin with an implementation S of P and we wish to construct an implementation of $P\backslash b$. The effect of hiding the event b is to allow occurrences of b to take place invisibly, without the knowledge or participation of the environment. At each stage where the old process could have performed either a visible action or the hidden action, a nondeterministic choice is introduced. In implementing $P\backslash b$ we must take these decisions into account. Suppose first that S cannot initially perform a hidden action, because $b \notin S^0$. In this case the initials of our implementation must be the same as those of S; and once an event c has taken place we must continue by implementing $\det(S \underline{\text{after}} \langle c \rangle)\backslash b$. Next suppose that S can initially perform b, and let $S^0 = C \cup \{ b \}$. If C is empty, this hidden action is unavoidable and we must implement $\det(S \underline{\text{after}} \langle b \rangle)\backslash b$. Otherwise, when C is nonempty, there is initially a possibility of both hidden and visible action. We may implement $\det(S)\backslash b$ in such a case either by *forcing* the initial hidden action or by *avoiding* it. The following two identities capture precisely these ideas:

(i) $(x : C \to P(x))\backslash b = (x : C \to P(x)\backslash b)$

(ii) $(x : C \cup \{ b \} \to P(x))\backslash b = P(b)\backslash b \sqcap (P(b)\backslash b \,\Box\, (x : C \to P(x)\backslash b))$ when $b \notin C$.

In the special case when C is empty, when the initial occurrence of b is unavoidable, the second identity reduces to

(iii) $(b \to P(b))\backslash b = P(b)\backslash b.$

Since we already know how to implement prefixing, nondeterministic choice and conditional choice, we can define a recursive algorithm for generating implementations of $P\backslash b$ from implementations of P.

Definition 3.3.6.1: For any tree S let $\text{hide}(b)(S)$ be the set of trees given by:

(i) $\text{hide}(b)(S) = \{ (x : S^0 \to T(x)) \mid \forall x \in S^0 . T(x) \in \text{hide}(b)(S(x)) \}$ if $b \notin S^0$,

(ii) $\text{hide}(b)(S) = \{ U, U \underline{\text{else}} V, V \underline{\text{else}} U \mid U \in \text{hide}(b)(S(b)) \;\&\; V \in \text{hide}(b)(\text{vis}(S)) \}$ if $b \in S^0$, where $\text{vis}(S) = (x : S^0 - \{ b \} \to S(x))$.

Here we have used $\text{vis}(S)$ to represent the tree obtained from S by removing any initial arc and its corresponding subtree in which the label on the arc is to be hidden.

It should be clear that one can use this definition of $\text{hide}(b)(S)$ to generate the successive layers of all implementations of $\det(S)\backslash b$, at least unless the

possibility of internal chatter arises; this will be the case when $b^* \subseteq S$, or more generally when there is a trace s such that $sb^* \subseteq S$. In fact, the above "definition" gives no information at all about implementations of the process $\det(b^*)\backslash b$, since it reduces to the vacuous:

$$\text{hide}(b)(b^*) = \{ U \mid U \in \text{hide}(b)(b^*) \}.$$

In this situation, there are two alternatives. If we assume that we are to use the algorithm to establish that a tree is an implementation by considering its successive finite depth cross-sections, and that we cannot assume that a particular tree is a valid implementation unless each of its cross-sections can be shown to be consistent with the above definition, then only the trivial tree NIL can safely be assumed to belong to $\text{hide}(b)(b^*)$. On the other hand, if we regard the algorithm as a method of eliminating a tree from the reckoning whenever one of its finite depth cross-sections fails to match the definition, we can never rule out any particular tree as a potential member of $\text{hide}(b)(b^*)$. The first approach amounts to identifying internal chatter with STOP, while the second identifies it with CHAOS. We must modify the above definition of $\text{hide}(b)(S)$ with one of the following extra clauses. The first corresponds to our first definition of hiding, and the second to our later version.

$$\begin{array}{lll} \text{(iii)} & \text{NIL} \in \text{hide}(b)(S) & \text{if } b^* \subseteq S, \\ \text{(iii')} & \text{hide}(b)(S) = \text{TREE} & \text{if } b^* \subseteq S. \end{array}$$

Using the notation of the previous chapter for the two versions of hiding, we can summarise as follows. We write $\text{hide}(b)$ and $\text{hide}(b)'$ for the two versions of Definition 3.3.6.1 obtained by using respectively clauses (i),(ii),(iii) and clauses (i),(ii),(iii'). The following identities show that these alternatives correspond to our two forms of hiding: one identifies infinite chatter with deadlock, the other with CHAOS.

$$\begin{aligned} \det(S)\backslash b &= \text{proc}(\text{hide}(b)(S)) \\ P\backslash b &= \text{proc}(\bigcup\{ \text{hide}(b)(S) \mid S \in \text{imp}(P) \}). \end{aligned}$$

$$\begin{aligned} \det(S)/b &= \text{proc}(\text{hide}(b)'(S)) \\ P/b &= \text{proc}(\bigcup\{ \text{hide}(b)'(S) \mid S \in \text{imp}(P) \}). \end{aligned}$$

In order to prove these identities, which is tantamount to proving that both versions of hiding are *implementable,* one needs to show that

$$\begin{aligned} P\backslash b &= \bigcup\{ \det(S)\backslash b \mid S \in \text{imp}(P) \}, \\ P/b &= \bigcup\{ \det(S)/b \mid S \in \text{imp}(P) \}. \end{aligned}$$

This is straightforward, once one has shown that P can indulge in infinite internal chatter if and only if one of its implementations also has this possibility.

7. Sequential composition.

Recall that we use the distinguished event $\surd$ to denote successful termination. When we implement the sequential composition $P;Q$ we want to run an implementation S of P until it terminates, and then run an implementation T of Q. However, termination of the first process is allowed to take place invisibly, without the participation of the environment (or implementor). If at some stage during the activity of P it is possible for the next step to be either termination or another event, the environment or implementor may be allowed a conditional choice between starting up the second process and continuing the first. These ideas are captured precisely in the following identities.

$$\text{(i)} \qquad (x : B \to P(x));Q = (x : B \to P(x);Q)$$

$$\text{(ii)} \quad (x : B \cup \{\surd\} \to P(x));Q = Q \sqcap (Q \,\Box\, (x : B \to P(x);Q)) \quad \text{if } \surd \notin B.$$

Again these identities enable us to define the effects of sequential composition on deterministic processes, as follows. We will write $\text{seq}(S,T)$ for the set of implementations of $\text{det}(S);\text{det}(T)$.

Definition 3.3.7.1: Let S and T be trees.

$$\begin{aligned}\text{seq}(S,T) &= \{\,(x : S^0 \to U(x)) \mid \forall x \in S^0.\, U(x) \in \text{seq}(S(x),T)\,\} \\ &\qquad \text{when } \surd \notin S^0, \\ &= \{\,T, T \,\underline{\text{else}}\, U, U \,\underline{\text{else}}\, T \mid U \in \text{seq}(S',T)\,\} \\ &\qquad \text{when } \surd \in S^0 \text{ and } S' = (x : S^0 - \{\surd\} \to S(x)).\end{aligned}$$

Theorem 3.3.7.2:

$$\text{(i)} \quad \text{det}(S);\text{det}(T) = \text{proc}(\text{seq}(S,T))$$

$$\text{(ii)} \quad P;Q = \text{proc}(\textstyle\bigcup\{\,\text{seq}(S,T) \mid S \in \text{imp}(P)\ \&\ T \in \text{imp}(Q)\,\}).$$

When termination is an unavoidable event in S, the definition of $\text{seq}(S,T)$ reduces to

$$\text{seq}(S,T) = \{\,S;T\,\}.$$

Hence, when $\surd$ is unavoidable for P, we get

$$P;Q = \text{proc}\{\,S;T \mid S \in \text{imp}(P)\ \&\ T \in \text{imp}(Q)\,\}.$$

8. Inverse image.

For any alphabet transformation f the inverse image operation is deterministic. Since we have the identity

$$f^{-1}[\det(T)] = \det(f^{-1}(T)),$$

it follows that the implementations of $f^{-1}[P]$ are simply the inverse images of implementations of P.

$$f^{-1}[P] = \mathrm{proc}\{\, f^{-1}(T) \mid T \in \mathrm{imp}(P) \,\}.$$

9. Direct image.

Unless the alphabet transformation f is injective, the direct image operation is not deterministic. Intuitively speaking, whenever $f[P]$ performs an event c we can choose to implement this action as any event $b \in f^{-1}(c)$. Of course, when f is injective there is no choice here. We can give a recursive characterisation of the implementations of $f[\det(T)]$ as follows. The initials of any implementation S must be the set $f(T^0)$. For every event c in this set, there must be an event $b \in f^{-1}(c)$ such that $S(c)$ implements the process $f[\det(T)\ \underline{\text{after}}\ \langle b \rangle]$. If we let $\mathrm{apply}(f)(T)$ be the set of trees satisying this condition, we can characterise this set as follows:

$$\mathrm{apply}(f)(T) = \{\, S \mid S^0 = f(T^0) \& \forall c \in S^0 . \exists b \in f^{-1}(c) . S(c) \in \mathrm{apply}(f)(T(b)) \,\}.$$

When f is injective the definition of $\mathrm{apply}(f)(T)$ reduces to $\{\, f(T) \,\}$.

Summary.

This chapter shows that an equivalent formulation of the domain of processes can be given in terms of sets of trees, or sets of deterministic processes. The functions defined on processes in Chapter 2 are all *implementable* as functions on sets of trees.

Chapter 4

Relationship with Milner's CCS

0. Introduction

In this chapter we consider the relationship between the failures model of CSP and Milner's synchronization tree model for his language CCS. Milner bases his model on a simple event-labelled tree structure, in which branches represent possible sequences of actions and subtrees represent possible future behaviours. He introduces a notion of *observation equivalence* between trees, to capture precisely the conditions under which two trees can be said to represent the same behaviour. We will show that there is a natural way to use synchronization trees to represent CSP processes, and that there is an equivalence relation on trees which faithfully mirrors the failure set semantics. We will be able to define tree operations which correspond to the CSP process operations. The relationship between Milner's equivalence relation and ours will be investigated. This will reveal interesting differences between the two systems, reflecting the fact that the underlying philosophies of the two models of concurrency differ.

1. Milner's synchronization trees

We now present a slightly simplified version of Milner's model of concurrency. The notation has been adapted to aid the comparison.

Synchronisation trees.

Milner's model is again based on the concept of an event as an indivisible atomic action performed by a process. Again behaviour is characterised in a step-by-step manner. In [M] Milner motivates his model by considering nondeterministic machines accepting strings over an alphabet. We may paraphrase his description as follows. A nondeterministic acceptor over alphabet Σ is a black box whose behaviour can be tested by asking it to accept symbols one at a time. The box has a button for each event in the alphabet. If one attempts to press the button labelled b there are two possible outcomes:

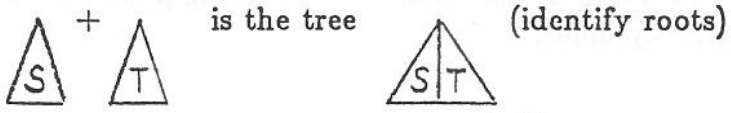
(i) Failure – the button is locked;
(ii) Success – the button goes down (and a state transition occurs).

An *experiment* consists of trying to press a sequence of buttons and it *succeeds* if the buttons go down in the correct order. Of course, if the black box exhibits nondeterministic behaviour then its behaviour will not necessarily be discernable from observing its successful experiments alone. Indeed, under the conventional definition, a nondeterministic acceptor may have some of its transitions labelled by the empty string. An empty transition indicates the possibility that the machine can, without being observed, change its internal state. Such a state change will not be detectable by the experimenter, since it is not accompanied by any visible action. Milner uses the symbol τ to stand for an empty transition, treating it in effect as an extra event. A *synchronisation tree* (ST) is then a rooted, unordered, finitely branching tree each of whose arcs is labelled by a member of $\Sigma \cup \{\tau\}$. A *rigid* ST is just an ST with no arcs labelled by τ; it represents the behaviour of a machine which cannot make silent (empty) transitions.

Milner uses an elementary algebra over STs, whose operations are:

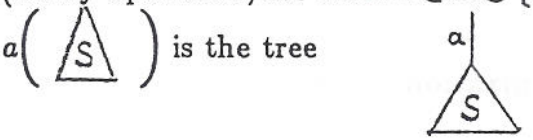
NIL (nullary operation)
NIL is the tree •

\+ (binary operation)



(identify roots)

a (unary operation, for each $a \in \Sigma \cup \{\tau\}$)



Thus, the general synchronisation tree can be expressed in the form

$$S = \sum_{i=1}^{n} a_i S_i + \sum_{j=1}^{m} \tau S_j'.$$

In the case $S = \text{NIL}$ we have $n = m = 0$.

Using variables S, T, U to range over STs, the following laws hold:

(A1)	Associativity	$S + (T + U) = (S + T) + U$
(A2)	Commutativity	$S + T = T + S$
(A3)	Nullity	$S + \text{NIL} = S$
(A4)	Idempotence	$S + S = S.$

Observation equivalence.

Next we introduce binary relations $\xrightarrow{c}$ on trees ($c \in \Sigma \cup \{\tau\}$). We will use b, c to range over $\Sigma \cup \{\tau\}$, and a to range over Σ.

$S \xrightarrow{a} S'$ means "S can transform to S' by an a-action"
$S \xrightarrow{\tau} S'$ means "S can transform to S' by an invisible action"

For instance, the transitions of the tree $S = \sum_{i=1}^{n} c_i S_i.$ are:

$$S \xrightarrow{c_i} S_i \qquad (i = 1, \ldots, n).$$

We will write $S \xrightarrow{s} S'$, for $s = \langle c_1, \ldots, c_n \rangle$, to mean that for some $S_0, \ldots, S_n$

$$S = S_0 \xrightarrow{c_1} S_1 \cdots \xrightarrow{c_n} S_n = S'.$$

Thus, $S \xrightarrow{s} S'$ if and only if S has a path from its root on which the sequence of labels is s and the subtree remaining is S'.

The result of performing any sequence $\langle a_1, \ldots, a_n \rangle$ of visible actions on S may be any S' for which there is a sequence t with

$$S \xrightarrow{t} S'$$

and t has the form

$$t = \tau^{k_0} a_1 \tau^{k_1} a_2 \ldots a_n \tau^{k_n}.$$

That is, any number of invisible transitions may occur, before, among or after the a_i. Recall that $t \backslash \tau$ denotes the sequence obtained from t by omitting all occurrences of τ. The above condition on t is clearly equivalent to requiring that $t \backslash \tau = s$. Following Milner, we introduce the relations $\stackrel{s}{\Longrightarrow}$ on trees for $s \in \Sigma^*$.

$S \stackrel{s}{\Longrightarrow} S'$ means "S can transform to S' by an s-experiment."
$S \stackrel{s}{\Longrightarrow} S'$ iff, for some t such that $t \backslash \tau = s$, $S \xrightarrow{t} S'$.

Milner's definition of observation equivalence is in terms of a sequence $\{\approx_n | \ n \geq 0\}$, of finer and finer equivalence relations:

Definition 4.1.1: (Observation equivalence)

(1) $S \approx_0 T$ is always true;

(2) $S \approx_{k+1} T \quad \Leftrightarrow \quad \forall s \in \Sigma^*.$
(i) $S \stackrel{s}{\Longrightarrow} S'$ implies $\exists T'.T \stackrel{s}{\Longrightarrow} T' \ \& \ S' \approx_k T'$;
(ii) $T \stackrel{s}{\Longrightarrow} T'$ implies $\exists S'.S \stackrel{s}{\Longrightarrow} S' \ \& \ S' \approx_k T'$.

(3) $S \approx T \quad \Leftrightarrow \quad \forall k \geq 0.\ S \approx_k T.$

It is easy to see that $\approx_0$ makes no distinctions between trees, and that two trees are related by $\approx_1$ iff they have the same possible sequences of visible actions. The successive relations make more and more distinctions. The observation equivalence relation $\approx$ is thus obtained as a limit. If we define an s-derivative of S to be any S' such that $S \stackrel{s}{\Longrightarrow} S'$, we can rephrase the definition of $\approx_{k+1}$ in terms of $\approx_k$ as follows:

$S \approx_{k+1} T \quad \Leftrightarrow \quad \forall s \in \Sigma^*,$
S and T have the same s-derivatives, up to $\approx_k$ equivalence.

For finite trees, it turns out that the observation equivalence relation "satisfies" its own definition, in that the following recursive formulation defines the same relation:

Theorem 4.1.2: For finite trees,

$S \approx T \quad \Leftrightarrow \quad \forall s \in \Sigma^*.$
(i) $S \stackrel{s}{\Longrightarrow} S' \Rightarrow \exists T'.T \stackrel{s}{\Longrightarrow} T' \ \& \ S' \approx T'$
(ii) $T \stackrel{s}{\Longrightarrow} T' \Rightarrow \exists S'.S \stackrel{s}{\Longrightarrow} S' \ \& \ S' \approx T'.$

Thus, two finite trees S and T are observation equivalent iff for every possible sequence of visible actions they have observationally equivalent derivatives.

Milner also shows that his observation equivalence satisfies a simple set of axioms. The following three axioms, together with the earlier axioms (A1)–(A4) and the following inference rule, are all valid.

Theorem 4.1.3: Observation equivalence satisfies the following system of axioms and rules:

(M1) $\tau S \approx S$
(M2) $S + \tau S + T \approx \tau S + T$
(M3) $aS + a(\tau S + T) + U \approx a(\tau S + T) + U$

$$\text{(R)} \qquad \frac{S \approx S'}{aS + T \approx aS' + T}$$

Proof. We must establish that the axioms and inference rule are sound; a simple inductive argument serves to show that for each n the left-hand and right-hand side of each axiom are n-equivalent. Similarly we use induction to show that if the premise in an instance of rule (R) is valid then the consequent is also valid. The details are omitted; see [M1]. ∎

Milner's tree operations.

CCS, a Calculus for Communicating Systems, described in full in [M1], is based on a set of operations including the ones already introduced (NIL, $+$ and $a\,\cdot$) and a binary *composition* $|$, a *relabelling* operation $[a \setminus b]$ and a *restriction* operation which Milner denotes $\backslash a$ (not to be confused with our hiding operation). We will write instead $S-a$ as restriction is anyway very similar in intent to our derived operation of the same name.

We assume, as does Milner, that the alphabet Σ is composed of two disjoint sets Δ and $\overline{\Delta}$, and that there is a bijection between these two sets such that for all $a \in \Sigma$ we have $\overline{\overline{a}} = a$. In Milner's terminology, events a and $\overline{a}$ are *complementary*. The special event τ has no complement.

Following Milner, we define "expansion rules" for these new operations. For the synchronisation trees S and T of form

$$S = \sum_{i=1}^{n} a_i S_i + \sum_{i=1}^{N} \tau S_i',$$
$$T = \sum_{j=1}^{m} b_j T_j + \sum_{j=1}^{M} \tau T_j',$$

we have

$$S|T = \sum_{i=1}^{n} a_i(S_i|T) + \sum_{i=1}^{N} \tau(S_i'|T) + \sum_{a_i=\overline{b_j}} \tau(S_i|T_j) + \sum_{j=1}^{m} b_j(S|T_j) + \sum_{j=1}^{M} \tau(S|T_j').$$

$$S[a \setminus b] = \sum_{i=1}^{n} \hat{a_i} S_i[a \setminus b] + \sum_{i=1}^{N} \tau S_i'[a \setminus b].$$

$$S - a = \sum_{a_i \neq a, \overline{a}} a_i(S_i - a) + \sum_{i=1}^{N} \tau(S_i' - a).$$

Here we have used the notation $\hat{c}$, for the alphabet transformation with the effect:

$$\hat{c} = a, \overline{a} \quad \text{if } c = b, \overline{b} \text{ respectively}$$
$$= c \qquad \text{otherwise.}$$

This alphabet transformation respects complementary events.

Observation congruence.

Milner is primarily concerned with the *congruence* generated by observation equivalence. This is characterized as the largest relation contained in the observation equivalence relation but which also respects all of the CCS operations. Milner shows that all CCS operations except + already preserve observation equivalence, but that + does not: for example, although the law $S \approx \tau S$ holds generally, it is not the case that $S + T \approx \tau S + T$ for all S and T. Milner proves that the congruence relation, which he denotes $\approx^c$, ia axiomatizable (at least, when restricted to finite trees). For later reference, we state his definition and result.

Definition 4.1.4: The equivalence relation $\approx^c$ on trees is:

$$S \approx^c T \Leftrightarrow \forall U.\, S + U \approx T + U.$$

This relation is a full congruence with respect to all CCS contexts. See [Milner] for details. The following set of laws, together with (A1)–(A4) and the expansion rules, are complete for finite trees. We have already seen that they are *valid,* by Theorem 4.1.3 and the fact that observation congruence implies observation equivalence.

Theorem 4.1.5: (τ laws)

(T1) $$a\tau S \approx^c aS$$

(T2) $$S + \tau S + T \approx^c \tau S + T$$

(T3) $$aS + a(\tau S + T) + U \approx^c a(\tau S + T) + U$$

(R) $$\frac{S \approx^c S'}{aS + T \approx^c aS' + T}$$

The proof system containing axioms (A1)–(A4), (T1)–(T3) and rule (R) is complete for observation congruence of finite trees.

Proof. See [M1]. ∎

2. Failure equivalence

A synchronisation tree T without any τ arcs, i.e., a rigid synchronisation tree, clearly represents a deterministic process. Since τ arcs correspond to nondeterministic choices, in general a synchronisation tree represents a nondeterministic process. We will see that it is possible to make this relationship precise. First we define the *failure set* of a synchronisation tree. Then we exhibit a mapping from CSP notation to synchronisation trees which behaves properly with respect to failure sets. We also define an equivalence relation on trees that identifies two trees if and only if they have the same failure sets. This equivalence relation turns out to be axiomatizable in the same way as Milner's observation equivalence, and the axiom system differs from Milner's in several interesting ways. Finally, we define synchronisation tree operations corresponding to the process operations.

Failure sets.

The *initials* of a synchronisation tree are those visible events appearing first on the branches of the tree. The *traces* of a tree are the sequences of visible actions appearing on the branches. Equivalently, the traces of T are the sequences s for which T has at least one s-derivative. It is natural to say that a tree can *refuse* a set of events X if it can make a silent transition to a subtree none of whose initials is a member of X. Finally, the *failures* of a tree are the pairs (s, X) such that the tree has an s-derivative which can refuse X. The following formal definition makes this precise.

Definition 4.2.1: Let T be a synchronisation tree.

(i) $\text{initials}(T) = \{ a \in \Sigma \mid \exists T'.T \stackrel{\langle a \rangle}{\Longrightarrow} T' \}$

(ii) $\text{traces}(T) = \{ s \mid \exists T'.T \stackrel{s}{\Longrightarrow} T' \}$

(iii) $\text{refusals}(T) = \{ X \mid \exists T'.T \stackrel{\langle\rangle}{\Longrightarrow} T' \;\&\; X \cap \text{initials}(T') = \emptyset \}$

(iv) $\text{failures}(T) = \{ (s, X) \mid \exists T'.T \stackrel{s}{\Longrightarrow} T' \;\&\; X \cap \text{initials}(T') = \emptyset \}.$

Definition 4.2.2: The *failure equivalence* relation $\equiv$ on synchronisation trees is defined as follows:

$$S \equiv T \quad \Leftrightarrow \quad \text{failures}(S) = \text{failures}(T).$$

We can now make a simple connection between Milner's equivalence and the failure equivalence. For two trees are identified by Milner's $\approx_1$ relation if and only if they have the same trace set, and equality of failure sets clearly implies equality of trace sets. Moreover, Milner's second equivalence relation $\approx_2$ identifies two trees when their respective derivatives can be paired up to have the same trace sets, while the failure equivalence only looks at that the initials of the derivatives. This means that $\approx_2$ makes fewer identifications than failure equivalence. Hence the following result:

Theorem 4.2.3: The failure equivalence relation lies between Milner's first and second relations, in the sense that, for all synchronisation trees S and T,

$$S \approx_2 T \quad \Rightarrow \quad S \equiv T \quad \Rightarrow \quad S \approx_1 T.$$

Moreover, these implications are strict; as relations we have the following strict inclusions:

$$\approx_1 \quad \supset \quad \equiv \quad \supset \quad \approx_2 .$$

It is important to notice that $\equiv$ *cannot* be defined in an analogous way to the above alternative definitions of Milner's first two relations, simply by replacing the condition that the *traces* of the derivatives be the same by the condition that their *initials* be the same. That is to say, the relation $\simeq$ defined by

$$\begin{aligned} S \simeq T \Leftrightarrow \quad & \forall s \in \Sigma^*. \\ \text{(i)} \quad & S \stackrel{s}{\Longrightarrow} S' \Rightarrow \exists T'.T \stackrel{s}{\Longrightarrow} T' \ \&\ \text{initials}(T') = \text{initials}(S') \\ \text{(ii)} \quad & T \stackrel{s}{\Longrightarrow} T' \Rightarrow \exists S'.S \stackrel{s}{\Longrightarrow} S' \ \&\ \text{initials}(T') = \text{initials}(S'), \end{aligned}$$

is not the same as failure equivalence. Indeed, there is a strict inclusion:

$$S \equiv T \quad \supset \quad S \simeq T.$$

Failure equivalence can be axiomatized in much the same way that we had axioms for observation equivalence. Indeed, since observation equivalence implies failure equivalence, all of the axioms for observation equivalence are still valid for the failure relation. However, the two relations differ; the differences are made apparent by the choice of axioms to characterise failure equivalence.

Theorem 4.2.4: Failure equivalence is the relation on trees characterised by the following axioms and inference rule (together with the basic axioms (A1)–(A4)):

$$\text{(B1)} \quad \tau S \equiv S$$

$$\text{(B2)} \quad S + \tau T + U \equiv \tau(S + T) + \tau T + U$$

$$\text{(B3)} \quad aS + aT + U \equiv a(\tau S + \tau T) + U$$

$$\text{(B4)} \quad \tau(aS + T) + \tau(aS' + T') \equiv \tau(aS + aS' + T) + \tau(aS + aS' + T')$$

$$\text{(R)} \quad \frac{S \equiv S'}{aS + T \equiv aS' + T}.$$

The proof of this theorem relies on some lemmas. Firstly we should show that the axioms are valid in the failures model, by checking that in each axiom the failures of the left-hand and right-hand side coincide, and we should show that the inference rule is sound. This is straightforward, and the details are left to the reader. Thus we know that every equivalence derivable from the proof system is valid; to complete the proof we must show that the system is *complete:* for finite trees S and T, if the failures are identical then the equivalence $S \equiv T$ is provable. The proof is based on reduction of a tree to *normal form.* We show that any tree can be proved equivalent to a unique tree in normal form, and that equivalent normal forms are identical (and hence provably equivalent.)

Lemma 4.2.5: The following laws are derivable from axioms (B1)–(B4) and rule (R), for all $n \geq 1$:

$$\text{(D1)} \quad a(\sum_{i=1}^{n} \tau S_i) + T \equiv \sum_{i=1}^{n} aS_i + T$$

$$\text{(D2)} \quad S + \sum_{j=1}^{m} \tau T_j \equiv \sum_{j=1}^{n} \tau(S + T_j) + \sum_{j=1}^{n} \tau T_j.$$

$$\text{(D3)} \quad \sum_{i=1}^{n} \tau S_i + U \equiv \tau \sum_{i=1}^{n} \tau S_i + U$$

$$\text{(D4)} \quad \sum_{i=1}^{n} \tau(aS_i + T_i) \equiv \sum_{i=1}^{n} \tau(aS + T_i),$$

$$\text{where } S = \sum_{j=1}^{n} \tau S_j.$$

Proof. By induction on the number of terms in the sums. The base case is an instance of an axiom, in each case. We omit details. ∎

Lemma 4.2.6: The following laws are derivable:

$$\text{(C1)} \quad \tau S + \tau T \equiv \tau S + \tau T + \tau(S + T)$$

$$\text{(C2)} \quad \tau S + \tau(S + T + U) \equiv \tau S + \tau(S + T) + \tau(S + T + U).$$

Proof. For (C1):

$$\begin{aligned} \tau S + \tau T &\equiv \tau(\tau S + T) + \tau T && \text{by (B2)} \\ &\equiv \tau(\tau S + \tau(S+T)) + \tau T && \text{by (B2) and (R)} \\ &\equiv \tau S + \tau(S+T) + \tau T && \text{by (B3).} \end{aligned}$$

For (C2):

$$\begin{aligned} \tau S + \tau(S+T+U) &\equiv \tau(\tau S + S + T + U) + \tau(S+T+U) && \text{by (B2)} \\ &\equiv \tau(\tau S + T + U) + \tau(S+T+U) && \text{by (B2), (A4) and (R)} \\ &\equiv \tau(\tau S + \tau(S+T) + U) + \tau(S+T+U) && \text{by (B2) and (R)} \\ &\equiv \tau(\tau S + \tau(S+T) + \tau(S+T+U)) + \tau(S+T+U) && \\ & && \text{by (B2) and (R)} \\ &\equiv \tau S + \tau(S+T) + \tau(S+T+U) + \tau(S+T+U) && \text{by (D3)} \\ &\equiv \tau S + \tau(S+T) + \tau(S+T+U) && \text{by (A4)} \end{aligned}$$

∎

Lemma 4.2.7: The following rule is a derived rule in the above system:

$$\text{(T)} \qquad \frac{\sum_{i=1}^{n} \tau S_i \equiv \sum_{j=1}^{m} \tau T_j}{\sum_{i=1}^{n} \tau S_i + U \equiv \sum_{j=1}^{m} \tau T_j + U}.$$

Proof. By (D3) and rule (R). ∎

Definition 4.2.8: Let $\mathcal{B}$ be a set of sets of events. Say $\mathcal{B}$ is *convex* if and only if it is non-empty, and for all A and C in $\mathcal{B}$,

$$\begin{aligned} &\text{(i)} && A \cup C \in \mathcal{B}, \\ &\text{(ii)} && A \subseteq B \subseteq C \Rightarrow B \in \mathcal{B}. \end{aligned}$$

There is a connection between this notion of convexity and the definition of convex closure in Chapter 3. Quite simply, a set $\mathcal{T}$ of trees over Σ is convex closed if and only if for all traces $t \in \bigcup \mathcal{T}$, the sets of events

$$\{\, \text{initials}(T \underline{\text{after}}\, t) \mid t \in T \in \mathcal{T} \,\}$$

form a convex set. The proof is elementary, and is only mentioned in passing; this fact will not be used in this chapter.

Definition 4.2.9: The tree T is a *normal form* iff it has the structure:

$$T = \sum_{B \in \mathcal{B}} \tau \sum_{b \in B} bT_b,$$

for some convex set $\mathcal{B}$, and where each T_b is also a normal form.

Note that when a tree is in normal form there is a unique tree corresponding to each trace of the tree; a particular trace may correspond to more than one path in the tree, but in each case the subtree rooted at the end of the path is the same.

Theorem 4.2.10:

Any tree T is provably equivalent to some tree T^* in normal form.

Proof. We outline a method for reducing an arbitrary (finite) tree to normal form. Derived laws (D1)–(D4) and the derived rule (T) can be used to induce *uniformity:* at each node of the tree either all outgoing arcs are labelled τ or none of them are. Then multiple occurrences of a visible label at a node can be combined into a single arc leading to a uniform τ node, using (D3). Convexity can be achieved by using (C1) and (C2). Finally, the uniqueness property is obtained by applying (D4). ∎

Theorem 4.2.11: If two normal forms S and T are failure equivalent, then $S \equiv T$ is deducible from the axiom system.

Proof. Let the two normal forms be

$$S = \sum_{B \in \mathcal{B}} \tau \sum_{b \in B} bS_b,$$
$$T = \sum_{C \in \mathcal{C}} \tau \sum_{c \in C} cT_c.$$

We are assuming that failures(S) = failures(T). We want to show that the formula $S \equiv T$ is derivable. The proof uses induction on the depth of the trees. The base case, when both normal forms are NIL, is trivial. For the inductive step, first we show that

(i) $\mathcal{B} = \mathcal{C}$
(ii) failures(S_b) = failures(T_b), for all b.

For (i) it is enough to establish an inclusion $\mathcal{B} \subseteq \mathcal{C}$, because we may then interchange S and T and repeat the argument for the converse. If $\mathcal{B} \not\subseteq \mathcal{C}$, let B be a set in $\mathcal{B}$ not in $\mathcal{C}$. Let X be the complement of B in initials(T),

$$X = \text{initials}(T) - B.$$

Then we know that $(\langle\rangle, X)$ is a failure of S, because

$$S \stackrel{\langle\rangle}{\Longrightarrow} \sum_{b \in B} bS_b.$$

By hypothesis, $(\langle\rangle, X)$ is also a failure of T, so there must be a set $C \in \mathcal{C}$ such that

$$T \stackrel{\langle\rangle}{\Longrightarrow} \sum_{c \in C} cT_c \ \& \ C \cap X = \emptyset.$$

But this can hold only if C is a subset of B. We also have assumed that $\mathcal{C}$ is a subset of initials(S); since we are supposing that S and T have the same failures, and hence the same traces, $\mathcal{C}$ is also a subset of initials(T). But this set is just $\bigcup\mathcal{C}$, so we have

$$C \subseteq B \subseteq \bigcup\mathcal{C}.$$

This implies that $B \in \mathcal{C}$, because both C and $\bigcup\mathcal{C}$ belong to the convex set $\mathcal{C}$. Thus we have reached a contradiction, so our assumption that $\mathcal{B} \not\subseteq \mathcal{C}$ must have been wrong. It follows that $\mathcal{B} \subseteq \mathcal{C}$; the converse is similar, and we have shown (i).

For (ii), merely observe that for all $a \in$ initials(S), a pair (at, X) is a failure of S if and only if (t, X) is a failure of S_a, because S has a unique a-derivative. The same holds for T.

To complete the proof we use induction. Since each S_b and T_b has smaller depth than S and T and are also in normal form, we may use the inductive hypothesis: for each b, the formula $S_b \equiv T_b$ is derivable. Hence, using rule (R) we can derive the equivalences

$$\sum_{b \in B} bS_b \equiv \sum_{b \in B} bT_b,$$

for all $B \in \mathcal{B} = \mathcal{C}$. The result follows by another application of (R) and from (A1)–(A4). ∎

Corollary: Two normal forms are equivalent iff they are identical up to order of terms. The proof of equivalence uses only the associativity and symmetry of + and can be derived using axioms (A1)–(A4) and rule (R).

Because of this last result, it is clear that any equivalence on trees which is at least strong enough to satisfy the laws of symmetry and associativity will agree with failure equivalence on normal forms; in particular, Milner's observation equivalence and our failure equivalence are identical relations when restricted to normal forms.

The failures preorder.

The failure set model of processes was based on a partial order which related two processes $P \sqsubseteq Q$ iff the failures of Q are also possible failures of P. Since we have also defined a notion of failures on trees, it is not surprising that there is a corresponding order on trees; strictly speaking, there is a *preorder,* as defined below.

Definition 4.2.12: The failures preorder $\sqsubseteq$ on trees is defined

$$S \sqsubseteq T \quad \Leftrightarrow \quad \text{failures}(S) \supseteq \text{failures}(T).$$

Corollary: For all trees S and T

$$S \equiv T \quad \Leftrightarrow \quad S \sqsubseteq T \ \& \ T \sqsubseteq S.$$

There is a simple relationship between $\sqsubseteq$ and $\equiv$, which will allow us to use the above complete axiom system for equivalence to construct a complete system for the preorder. The relevant result is:

Lemma 4.2.13: For all trees S and T

$$S \sqsubseteq T \quad \Leftrightarrow \quad \tau S + \tau T \equiv S.$$

Proof. From the definition, it is easy to see that

$$\text{failures}(\tau S + \tau T) = \text{failures}(S) \cup \text{failures}(T).$$

The result follows. ∎

As a corollary, we obtain a complete axiom system for $\sqsubseteq$ on finite trees by adding the following axioms and inference rules to the above system:

$$(O1) \qquad \frac{S \equiv T}{S \sqsubseteq T \ \& \ T \sqsubseteq S}$$

$$(O2) \qquad \frac{S \sqsubseteq T \ \& \ T \sqsubseteq S}{S \equiv T}$$

$$(O3) \qquad \frac{\tau S + \tau T \equiv S}{S \sqsubseteq T}$$

$$(O4) \qquad \frac{S \sqsubseteq T \ \& \ T \sqsubseteq U}{S \sqsubseteq U}.$$

3. Mapping processes to synchronisation trees

Now we define a mapping $[\![\cdot]\!]$ from processes to trees which may be thought of as a *representation map*. It will be the case that, for all processes P, the tree $[\![P]\!]$ has the same failure set as P. The representation map is defined by structural induction on the CSP syntax.

Definition 4.3.1: The mapping $[\![\cdot]\!] : \mathrm{PROC} \to \mathrm{TREE}$ is given by:

$$\begin{aligned}
[\![\mathrm{STOP}]\!] &= \mathrm{NIL} \\
[\![(a \to P)]\!] &= a[\![P]\!] \\
[\![P \sqcap Q]\!] &= \tau[\![P]\!] + \tau[\![Q]\!] \\
[\![P \,\Box\, Q]\!] &= [\![P]\!] \,\Box\, [\![Q]\!] \\
[\![P \,\|\, Q]\!] &= [\![P]\!] \,\|\, [\![Q]\!] \\
[\![P \,|||\, Q]\!] &= [\![P]\!] \,|||\, [\![Q]\!] \\
[\![P \backslash b]\!] &= [\![P]\!][\tau \setminus b] \\
[\![P;Q]\!] &= [\![P]\!];[\![Q]\!] \\
[\![f[P]]\!] &= f([\![P]\!]) \\
[\![f^{-1}[P]]\!] &= f^{-1}([\![P]\!])
\end{aligned}$$

where the tree operations $\Box, \|, |||, ;, f(\cdot), f^{-1}(\cdot)$ are given by

$$S = \sum_{i=1}^{n} a_i S_i + \sum_{i=1}^{m} \tau S_i'$$

$$T = \sum_{j=1}^{N} b_j T_j + \sum_{j=1}^{M} \tau T_j'$$

$$S \,\Box\, T = \sum_{i=1}^{n} a_i S_i + \sum_{j=1}^{N} b_j T_j + \sum_{i=1}^{m} \tau(S_i' \,\Box\, T) + \sum_{j=1}^{M} \tau(S \,\Box\, T_j')$$

$$S \,\|\, T = \sum_{a_i = b_j} a_i(S_i \,\|\, T_j) + \sum_{i=1}^{m} \tau(S_i' \,\|\, T) + \sum_{j=1}^{M} \tau(S \,\|\, T_j')$$

$$S \,|||\, T = \sum_{i=1}^{n} a_i(S_i \,|||\, T) + \sum_{j=1}^{m} b_j(S \,|||\, T_j) + \sum_{i=1}^{N} \tau(S_i' \,|||\, T) + \sum_{j=1}^{M} \tau(S \,|||\, T_j')$$

$$S;T = \sum_{a_i \neq \surd} a_i(S_i;T) + \sum_{i=1}^{m} \tau S_i';T + \sum_{a_i = \surd} \tau T$$

$$f(S) = \sum_{i=1}^{n} f(a_i) f(S_i) + \sum_{i=1}^{m} \tau f(S_i')$$

$$f^{-1}(S) = \sum_{i=1}^{n} \sum_{c \in f^{-1}(a_i)} c f^{-1}(S_i) + \sum_{i=1}^{m} \tau f^{-1}(S_i')$$

and where $S[\tau \setminus b]$ denotes the tree obtained by renaming the b-arcs of S to τ.

Note that nondeterministic choice is represented by τ-branching, hiding by renaming the hidden event to τ, and alphabet transformations and their inverses are again represented by relabelling events in the appropriate way. The definition of $S;T$ ensures that termination of S is hidden from the environment.

Theorem 4.3.2: For all processes P, $\text{failures}([\![P]\!]) = P$.

Proof. By structural induction. The result holds clearly for the case $P =$ STOP. We treat in detail only two cases.

(i) *Nondeterministic choice.* When $P = P_1 \sqcap P_2$, we may use the inductive hypothesis to assume that

$$\text{failures}([\![P_i]\!]) = P_i \qquad (i = 1, 2).$$

By definition,

$$[\![P]\!] = \tau[\![P_1]\!] + \tau[\![P_2]\!],$$

so that every derivative of $[\![P]\!]$ is also a derivative either of $[\![P_1]\!]$ or of $[\![P_2]\!]$, with the exception of the derivation

$$[\![P]\!] \stackrel{\langle\rangle}{\Longrightarrow} [\![P]\!].$$

However, it is clear that the initials of $[\![P]\!]$ are initials either of $[\![P_1]\!]$ or of $[\![P_2]\!]$, and hence that any set X satisfying

$$X \cap \text{initials}([\![P]\!]) = \emptyset$$

also satisfies the condition

$$X \cap \text{initials}([\![P_i]\!]) = \emptyset \qquad (i = 1, 2).$$

This means that this extra derivation does not add to the failures of $[\![P]\!]$. Since all other derivations are also derivations of $[\![P_1]\!]$ or $[\![P_2]\!]$, we have:

$$\begin{aligned} \text{failures}([\![P]\!]) &= \text{failures}([\![P_1]\!]) \cup \text{failures}([\![P_2]\!]) \\ &= P_1 \cup P_2 \qquad \text{(by hypothesis)} \\ &= P_1 \sqcap P_2 \qquad \text{(by definition)} \\ &= P. \end{aligned}$$

(ii) *Interleaving.* When $P = P_1 \,|||\, P_2$, we have to show that

$$\begin{aligned} \text{failures}([\![P_1 \,|||\, P_2]\!]) &= P_1 \,|||\, P_2 \\ &= \{\, (u, X) \mid \exists s, t.u \in \text{merge}(s, t) \,\&\, (s, X) \in P_1 \,\&\, (t, X) \in P_2 \,\} \end{aligned}$$

We use induction on the length of traces. The base case is for a failure of the form $(\langle\,\rangle, X)$. Such a pair belongs to the failure set of $[\![P]\!]$ iff there is a tree U such that

$$[\![P]\!] \stackrel{\langle\rangle}{\Longrightarrow} U \;\&\; X \cap \text{initials}(U) = \emptyset.$$

Let $S = [\![P_1]\!]$ and $T = [\![P_2]\!]$, and suppose they have the form

$$S = \sum_{i=1}^{n} a_i S_i + \sum_{i=1}^{N} \tau S_i' \qquad (a_i \neq \tau)$$

$$T = \sum_{j=1}^{m} b_j T_j + \sum_{j=1}^{M} \tau T_j' \qquad (b_j \neq \tau).$$

Then the tree $[\![P]\!]$ has the form

$$S|||T = \sum_{i=1}^{n} a_i(S_i|||T) + \sum_{j=1}^{m} b_j(S|||T_j) + \sum_{i=1}^{N} \tau(S_i'|||T) + \sum_{j=1}^{M} \tau(S|||T_j').$$

Thus we see that

$$\begin{aligned} S|||T \stackrel{\langle\rangle}{\Longrightarrow} U \Leftrightarrow & \\ \text{either} \quad & \exists S'.S \stackrel{\langle\rangle}{\Longrightarrow} S' \ \&\ U = S'|||T \\ \text{or} \quad & \exists T'.T \stackrel{\langle\rangle}{\Longrightarrow} T' \ \&\ U = S|||T'. \end{aligned}$$

Since it is clear that $\text{initials}(S|||T) = \text{initials}(S) \cup \text{initials}(T)$, this gives

$$\begin{aligned} \text{refusals}(S|||T) &= \{X \mid \exists U.S|||T \stackrel{\langle\rangle}{\Longrightarrow} U \ \&\ X \cap \text{initials}(U) = \emptyset\} \\ &= \{X \mid \exists S'.S \stackrel{\langle\rangle}{\Longrightarrow} S' \ \&\ X \cap \text{initials}(S'|||T) = \emptyset\} \\ &\quad \cup \{X \mid \exists T'.T \stackrel{\langle\rangle}{\Longrightarrow} T' \ \&\ X \cap \text{initials}(S|||T') = \emptyset\} \\ &= \{X \mid \exists S', T'.S \stackrel{\langle\rangle}{\Longrightarrow} S' \ \&\ T \stackrel{\langle\rangle}{\Longrightarrow} T' \ \&\ X \cap \text{initials}(S'|||T') = \emptyset\} \\ &= \{X \mid \exists S', T'.S \stackrel{\langle\rangle}{\Longrightarrow} S' \ \&\ T \stackrel{\langle\rangle}{\Longrightarrow} T' \ \& \\ &\qquad X \cap \text{initials}(S') = X \cap \text{initials}(T') = \emptyset\} \\ &= \{X \mid X \in \text{refusals}(S) \ \&\ X \in \text{refusals}(T)\} \\ &= \text{refusals}(S) \cap \text{refusals}(T). \end{aligned}$$

Since any traces which merge to the empty trace are also empty, this establishes the result for the case $u = \langle\rangle$.

Now suppose that $u = \langle c\rangle w$ and that $(u, X) \in \text{failures}(S|||T)$. Let U be a tree such that

$$S|||T \stackrel{u}{\Longrightarrow} U \ \&\ X \cap \text{initials}(U) = \emptyset.$$

Then it is easy to see that one of the following two conditions holds:

$$\begin{aligned} \text{either} \quad & \exists S', T'.S \stackrel{\langle c\rangle}{\Longrightarrow} S' \ \&\ T \stackrel{\langle\rangle}{\Longrightarrow} T' \ \&\ S'|||T' \stackrel{w}{\Longrightarrow} U \\ \text{or} \quad & \exists S', T'.S \stackrel{\langle\rangle}{\Longrightarrow} S' \ \&\ T \stackrel{\langle c\rangle}{\Longrightarrow} T' \ \&\ S'|||T' \stackrel{w}{\Longrightarrow} U. \end{aligned}$$

In the first case, since w has shorter length than u, we may assume that there are traces s and t, and trees S'' and T'' such that

$$w \in \text{merge}(s, t) \ \&\ S' \stackrel{s}{\Longrightarrow} S'' \ \&\ T' \stackrel{t}{\Longrightarrow} T'' \ \&\ U = S''|||T''.$$

But then we get

$$S \stackrel{\langle c \rangle}{\Longrightarrow} S' \stackrel{s}{\Longrightarrow} S''$$
$$T \stackrel{\langle\rangle}{\Longrightarrow} T' \stackrel{t}{\Longrightarrow} T''$$
$$X \cap \text{initials}(S'') = \emptyset$$
$$X \cap \text{initials}(T'') = \emptyset$$

and hence, since $\langle c \rangle s$ and t merge to u, this gives $(u, X) \in \text{failures}(S|||T)$, as required. The converse is similar. ∎

Corollary 4.3.3: For all processes P and Q,

$$P = Q \quad \Leftrightarrow \quad [\![P]\!] \equiv [\![Q]\!].$$

Theorem 4.3.4: Each of the tree operations in Definition 4.3.1 respects failure equivalence: whenever $S \equiv S', T \equiv T'$, we also have

$$aS \equiv aS'$$
$$\tau S + \tau T \equiv \tau S' + \tau T'$$
$$S \,\Box\, T \equiv S' \,\Box\, T'$$
$$S \,\|\, T \equiv S' \,\|\, T'$$
$$S \,|||\, T \equiv S' \,|||\, T'$$
$$S[\tau \setminus b] \equiv S'[\tau \setminus b]$$
$$S;T \equiv S';T'$$
$$f(S) \equiv f(S')$$
$$f^{-1}(S) \equiv f^{-1}(S').$$

Proof. Similar to Theorem 4.3.2. ∎

This result shows that, with respect to this set of tree operations, failure equivalence is a *congruence*. It also shows that $+$ as a tree operation is not definable in terms of the set of CSP operations, since $+$ does not respect failure equivalence.

4. Failures and CCS operations

We have seen already that Milner's prefixing, renaming and restriction all preserve failure equivalence, and that $+$ does not. Now let us consider the behaviour of Milner's *composition* operation $|$ with respect to failures. Recall that the alphabet Σ is assumed to be partitioned into disjoint sets Δ and $\overline{\Delta}$, and the function $a \mapsto \overline{a}$ is a bijection between these sets so that $\overline{\overline{a}} = a$ for all

$a \in \Delta$. The composition $S|T$ of two trees

$$S = \sum_{i=1}^{n} a_i S_i$$

$$T = \sum_{j=1}^{m} b_j T_j \quad (a_i, b_j \in \Sigma \cup \{\tau\})$$

is given by

$$S|T = \sum_{i=1}^{n} a_i(S_i|T) + \sum_{j=1}^{m} b_j(S|T_j) + \sum_{a_i=\overline{b_j}} \tau(S_i|T_j),$$

where we do not include any terms in the last summation with $a_i = \tau$ or $b_j = \tau$, because τ is assumed to have no complementary event.

The traces of $S|T$ are obtained by *composing* traces of S and T, in the following sense:

Definition 4.3.5: The set of all *compositions* of two traces s and t is defined by induction on the length of the traces:

(i) $\quad \mathrm{comp}(\langle\,\rangle, t) = \mathrm{comp}(t, \langle\,\rangle) = \{\, t \,\}$

(ii) $\quad \mathrm{comp}(as, bt) = \{\, au \mid u \in \mathrm{comp}(s, bt) \,\} \cup \{\, bu \mid u \in \mathrm{comp}(as, t) \,\}$
$\qquad\qquad \cup \{\, u \in \mathrm{comp}(s, t) \mid a = \overline{b} \,\}.$

Lemma 4.3.6: The traces of a composition $S|T$ are obtained by composition:

$$\mathrm{traces}(S|T) = \bigcup\{\, \mathrm{comp}(s, t) \mid s \in \mathrm{traces}(S) \;\&\; t \in \mathrm{traces}(T) \,\}.$$

Thus we see that the initials of a composition $S|T$ are given by the formula

$$\mathrm{initials}(S|T) = \bigcup\{\, \mathrm{initials}(S'), \mathrm{initials}(T') \mid \exists s.S \stackrel{s}{\Longrightarrow} S' \;\&\; T \stackrel{\overline{s}}{\Longrightarrow} T' \,\},$$

since an event c is initial for $S|T$ if and only if there is a trace s such that sc is a trace of S and $\overline{s}c$ is a trace of T. Similarly, the refusals of $S|T$ are:

$$X \in \mathrm{refusals}(S|T) \Leftrightarrow \exists s.(s, X) \in \mathrm{failures}(S) \;\&\; (\overline{s}, X) \in \mathrm{failures}(T),$$

and in general (u, X) is a failure of $S|T$ iff there are traces s and t such that

$$(s, X) \in \mathrm{failures}(S) \;\&\; (t, X) \in \mathrm{failures}(T) \;\&\; u \in \mathrm{comp}(s, t).$$

This formula gives a way of determining the failures of $S|T$ from the failures of S and T. Since the failures of a composite tree are therefore uniquely determined by the failures of the component trees, it follows that Milner's composition respects failure equivalence:

$$S \equiv S',\ T \equiv T' \;\Rightarrow\; S|T \equiv S'|T'.$$

Chapter 5

A proof system for CSP

1. Introduction

In this chapter we introduce a proof system which is sound and complete for failure equivalence of processes. First we consider a simple sub-language, whose terms are built from STOP, prefixing, conditional and unconditional choice. Obviously terms in this language will denote processes with only a finite number of possible traces. The logical language will contain assertions of the form $P \sqsubseteq Q$ or $P \equiv Q$. We give a set of axioms and inference rules for proving such assertions, and show that the system is both sound and complete; under the obvious interpretation of assertions, every provable assertion is true and every true assertion is provable.

Next we make the transition to a more general language, by allowing recursive terms. Now terms may denote infinite processes. We show how to modify the previous proof system to obtain a new system complete and sound for the larger language. Essentially, we use the well known ideas of *syntactic approximation* of terms, and use the fact that the failure semantics of an infinite process is uniquely determined by its finite syntactic approximants. A new inference rule is added which states this fact and basically allows us to reason about infinite terms by manipulating their finite approximations. Crucial to this work is the fact that all of the process operations in the language are continuous with respect to the nondeterminism ordering. Similar techniques were used by Hennessy and de Nicola [HN] to give a proof system for CCS.

Bearing in mind our earlier problems with the notion of *divergence,* we have made the proof system sufficiently general to cope with divergent processes, by adding to the language a term $\perp$ (representing divergence) and augmenting the failure set model with a divergence set component. A similar

augmentation of the failures model was suggested by Roscoe [R]. The special case of well-behaved (divergence–free) processes turns out to correspond to the sublanguage of all terms in which no recursive subterm has an unguarded occurrence of its bound variable, and in which there is no sub-term $\perp$. As a corollary, the proof system is also complete for the old failures ordering on well-behaved processes.

Throughout this chapter we will use P, Q, R to stand for terms in the variant of CSP currently under consideration. We are not necessarily assuming that the universal alphabet Σ is finite, but every term will only use a finite set of events. As usual, refusal sets are finite. In order to distinguish clearly between a term and the failure set it denotes, we will define a semantic function on terms.

2. A simple subset of CSP

Let FCSP (Finite CSP) be the language generated by the following syntax:

$$P ::= \text{STOP} \mid (a \to P) \mid P \,\Box\, P \mid P \sqcap P,$$

where a ranges over Σ. The semantic function $\mathcal{F}$ maps terms to failure sets, and is defined by structural induction as usual:

$$\mathcal{F}[\![\text{STOP}]\!] = \{(\langle\rangle, X) \mid X \subseteq \Sigma\}$$

$$\mathcal{F}[\![a \to P]\!] = \{(\langle\rangle, X) \mid a \notin X\} \cup \{(as, X) \mid (s, X) \in \mathcal{F}[\![P]\!]\}$$

$$\begin{aligned}\mathcal{F}[\![P \,\Box\, Q]\!] = &\{(\langle\rangle, X) \mid (\langle\rangle, X) \in \mathcal{F}[\![P]\!] \cap \mathcal{F}[\![Q]\!]\} \\ &\cup \{(s, X) \mid s \neq \langle\rangle \ \&\ (s, X) \in \mathcal{F}[\![P]\!] \cup \mathcal{F}[\![Q]\!]\}\end{aligned}$$

$$\mathcal{F}[\![P \sqcap Q]\!] = \mathcal{F}[\![P]\!] \cup \mathcal{F}[\![Q]\!].$$

We will use Φ to stand for a failure set. Recall that a failure set is a subset of $\Sigma^* \times p\Sigma$ such that

(i) $\quad \text{dom}(\Phi)$ is non-empty and prefix-closed,
(ii) $\quad (s, X) \in \Phi, Y \subseteq X \Rightarrow (s, Y) \in \Phi,$
(iii) $\quad (s, X) \in \Phi, (sa, \emptyset) \notin \Phi \Rightarrow (s, X \cup \{a\}) \in \Phi.$

The order on failure sets is

$$\Phi_1 \sqsubseteq \Phi_2 \Leftrightarrow \Phi_1 \supseteq \Phi_2.$$

We will write $P \sqsubseteq Q$ to mean $\mathcal{F}[\![P]\!] \sqsubseteq \mathcal{F}[\![Q]\!]$, where no confusion can arise; this convention merely corresponds to our earlier usage, when we were not concerned to keep terms and their denotations separate. So far we have merely recast the definitions of earlier sections. Now we introduce the proof system.

The logical language is built from FCSP terms and two binary relation symbols $\sqsubseteq$ and $\equiv$. Each formula in the language has the form $P \sqsubseteq Q$ or $P \equiv Q$. We include axioms on idempotence, symmetry, associativity of $\sqcap$ and $\Box$, distribution of these two operators over each other, and some interactions with prefixing. The inference rules assert monotonicity of the operators with respect to $\sqsubseteq$, and state that $\sqsubseteq$ is a partial order and $\equiv$ the associated equivalence. The following table lists the axioms:

(A1) $$P \sqcap P \equiv P$$

(A2) $$P \,\Box\, P \equiv P$$

(A3) $$P \sqcap Q \equiv Q \sqcap P$$

(A4) $$P \,\Box\, Q \equiv Q \,\Box\, P$$

(A5) $$P \sqcap (Q \sqcap R) \equiv (P \sqcap Q) \sqcap R$$

(A6) $$P \,\Box\, (Q \,\Box\, R) \equiv (P \,\Box\, Q) \,\Box\, R$$

(A7) $$P \sqcap (Q \,\Box\, R) \equiv (P \sqcap Q) \,\Box\, (P \sqcap R)$$

(A8) $$P \,\Box\, (Q \sqcap R) \equiv (P \,\Box\, Q) \sqcap (P \,\Box\, R)$$

(A9) $$P \,\Box\, \mathrm{STOP} \equiv P$$

(A10) $$P \sqcap Q \sqsubseteq P$$

(A11) $$(a \to P) \sqcap (a \to Q) \equiv (a \to P \sqcap Q)$$

(A12) $$(a \to P) \,\Box\, (a \to Q) \equiv (a \to P \sqcap Q)$$

(O1) $$\frac{P \sqsubseteq Q \sqsubseteq P}{P \equiv Q}$$

(O2) $$\frac{P \equiv Q}{P \sqsubseteq Q \sqsubseteq P}$$

(O3) $$\frac{P \sqsubseteq Q \sqsubseteq R}{P \sqsubseteq R}$$

(M1) $$\frac{P \sqsubseteq Q}{(a \to P) \sqsubseteq (a \to Q)}$$

(M2) $$\frac{P_1 \sqsubseteq Q_1 \;\&\; P_2 \sqsubseteq Q_2}{P_1 \sqcap P_2 \sqsubseteq Q_1 \sqcap Q_2}$$

(M3) $$\frac{P_1 \sqsubseteq Q_1 \;\&\; P_2 \sqsubseteq Q_2}{P_1 \,\Box\, P_2 \sqsubseteq Q_1 \,\Box\, Q_2}$$

Soundness.

Each axiom has already appeared in a previous chapter, where proofs of validity were given; similarly all of the operators were shown to be monotonic, so the inference rules (M1)–(M3) are valid. This means that every provable formula is true. We write $\vdash P \sqsubseteq Q$ when the formula $P \sqsubseteq Q$ is provable. The following theorem states that the proof system is *sound.*

Theorem 5.2.1: For all terms P,Q

$$\vdash P \sqsubseteq Q \Rightarrow \mathcal{F}[\![P]\!] \supseteq \mathcal{F}[\![Q]\!].$$

Derived laws.

The following laws are derivable, and hence valid. They will be useful in establishing completeness. The first states the connection between nondeterministic choice and the ordering. The second says that nondeterministic choice allows more failures in general than conditional choice, in accordance with our intuition and earlier results on these operators. The third law will be heavily used in establishing the existence of normal forms; it can be thought of as a *convexity law.*

Lemma 5.2.2:

The following formulae can be derived in the above proof system:

$$\begin{array}{ll} \text{(D1)} & P \sqcap Q \equiv P \Leftrightarrow P \sqsubseteq Q \\ \text{(D2)} & P \sqcap Q \sqsubseteq P \,\Box\, Q \\ \text{(D3)} & P \sqcap (Q \,\Box\, R) \sqsubseteq P \,\Box\, Q. \end{array}$$

Proof. For (D1) we have

$$\vdash P \sqcap Q \sqsubseteq P$$

by (A10). And if we assume $P \sqsubseteq Q$ is provable, we have:

$$P \sqsubseteq Q \vdash P \sqcap P \sqsubseteq P \sqcap Q,$$

by (M2). The result follows by (A1) and (O1).

For (D2):

$$\begin{array}{lll} (P \sqcap Q) \sqcap (P \,\Box\, Q) \equiv ((P \sqcap Q) \sqcap P) \,\Box\, ((P \sqcap Q) \sqcap Q) & & \text{by (A7)} \\ \equiv (P \sqcap Q) \,\Box\, (P \sqcap Q) & & \text{by (A1)} \\ \equiv P \sqcap Q & & \text{by (A2)} \end{array}$$

The result follows from (D1).

For (D3) we have

$$\begin{aligned} P \sqcap (Q \,\Box\, R) &\equiv (P \sqcap Q) \,\Box\, (P \sqcap R) && \text{by (A7)} \\ &\sqsubseteq (P \,\Box\, Q) \,\Box\, P && \text{by (M3), (A10) and (D2)} \\ &\equiv (P \,\Box\, P) \,\Box\, Q && \text{by (A6)} \\ &\equiv P \,\Box\, Q && \text{by (A2)} \end{aligned}$$

That completes the proof. ∎

Completeness.

We will show that whenever the failures of P include the failures of Q the formula $P \sqsubseteq Q$ is provable. For the proof we use a *normal form* theorem. We define a class of normal forms and show that every term is provably equivalent to a unique term in normal form. Moreover, we show that whenever the failures of one normal form include the failures of another, the corresponding formula is provable.

Essentially, a normal form will be a term with a *uniform* structure, rather like a nondeterministic composition of a collection of *guarded* terms. In order to get uniqueness of normal forms we will require certain closure conditions on the sets of guards appearing at each position in the term; these conditions amount to a *convexity* requirement. In addition, we will require that in a normal form every subterm guarded by a particular event be identical and also in normal form. This means that every normal form is itself built up from normal forms in a simple way that facilitates proofs. Formally, these constraints are defined as follows.

Normal forms.

Definition 5.2.3: A subset $\mathcal{B}$ is *convex* iff it is non-empty and

(i) $A, B \in \mathcal{B} \Rightarrow A \cup B \in \mathcal{B}$,

(ii) $A, C \in \mathcal{B} \;\&\; A \subseteq B \subseteq C \Rightarrow B \in \mathcal{B}$.

We will write $\mathrm{con}(\mathcal{B})$ for the smallest convex set containing $\mathcal{B}$, and refer to the *convex closure* of $\mathcal{B}$. There will be clear connections between this form of convexity on sets of sets of events and the convexity of Chapter 3 (on sets of trees).

Example 1. The set $\mathcal{A} = \{\emptyset, \{a, b\}\}$ is not convex, because

$$\emptyset \subseteq \{a\} \subseteq \{a, b\}$$

but $\{a\}$ is not in $\mathcal{A}$.

Example 2. The set $\mathcal{B} = \{\emptyset, \{a\}, \{b\}\}$ is not convex, because it does not contain $\{a, b\}$.

Example 3. The smallest convex set containing $\mathcal{A}$ and $\mathcal{B}$ is the set

$$\mathcal{C} = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}.$$

We have $\mathrm{con}(\mathcal{A}) = \mathrm{con}(\mathcal{B}) = \mathcal{C}$.

Example 4. For any set $B \subseteq \Sigma$ the powerset of B is convex.

Now we can define normal form:

Definition 5.2.4: A term P is in *normal form* iff it has the structure

$$\text{either } P = \mathrm{STOP}$$
$$\text{or } \quad P = \sqcap_{B \in \mathcal{B}} \Box_{b \in B}(b \to P_b)$$

for some convex set $\mathcal{B}$, and each P_b is also in normal form.

Note that although a normal form P may have "disjuncts " P_B and P_C with some initials in common, say

$$P_B = \Box_{b \in B}(b \to P_b)$$
$$P_C = \Box_{c \in C}(c \to P_c)$$

the definition forces these two processes to have identical derivatives P_a for all $a \in B \cap C$. Some examples will help.

Example 1. $P = \mathrm{STOP} \sqcap (a \to \mathrm{STOP})$ is in normal form: here $\mathcal{B}$ is the convex set $\{\emptyset, \{a\}\}$ and $P_a = \mathrm{STOP}$.

Example 2. $P = (a \to (b \to \mathrm{STOP})) \sqcap ((a \to \mathrm{STOP}) \Box (b \to \mathrm{STOP}))$ is not in normal form, because the two subterms guarded by a are distinct.

The next result is the basis of our completeness theorem.

Lemma 5.2.5: Any term P can be transformed using the proof system into a normal form.

Proof. By induction on the length of the term.

The base case, when $P = \mathrm{STOP}$, is trivial; the case when $P = (a \to Q)$ is also straightforward. In the remaining two cases, we must show that if P and Q are normal forms then $P \sqcap Q$ and $P \Box Q$ can be put into normal form.

To this end, suppose the two normal forms are:

$$P = \sqcap_{B\in\mathcal{B}}\Box_{b\in B}(b \to P_b)$$
$$Q = \sqcap_{C\in\mathcal{C}}\Box_{c\in C}(c \to Q_c).$$

Write $P_B = \Box_{b\in B}(b \to P_b)$ and $Q_C = \Box_{c\in C}(c \to Q_c)$, so that

$$P = \sqcap_{B\in\mathcal{B}} P_B$$
$$Q = \sqcap_{C\in\mathcal{C}} Q_C.$$

Then it is easily provable that

$$P \sqcap Q \equiv \sqcap_{B\in\mathcal{B}}\sqcap_{C\in\mathcal{C}} P_B \sqcap Q_C.$$

Let $\mathcal{A} = \mathcal{B} \cup \mathcal{C}$, and define the terms R_a for $a \in \mathcal{A}$ by:

$$\begin{aligned} R_a &= P_a && \text{if } a \in B\text{–}C, \\ &= Q_a && \text{if } a \in C\text{–}B, \\ &= P_a \sqcap Q_a && \text{if } a \in B \cap C. \end{aligned}$$

Using the obvious notation, it is clear that the statements

$$P_B \sqcap Q_C = R_B \sqcap R_C$$

are provable, and hence that

$$\vdash P \sqcap Q \equiv \sqcap_{A\in\mathcal{A}} R_A.$$

To complete the proof in this case we use the convexity laws to replace $\mathcal{A}$ by its convex closure.

Finally we must reduce $P \,\Box\, Q$ to normal form. Again it is easy to show that

$$\vdash P \,\Box\, Q \equiv \sqcap_{B\in\mathcal{B}}\sqcap_{C\in\mathcal{C}} P_B \,\Box\, Q_C,$$

and (using the same notation as above) that

$$\vdash P_B \,\Box\, Q_C \equiv R_{B\cup C}.$$

It follows that

$$\vdash P \,\Box\, Q \equiv \sqcap_{A\in\mathcal{A}}\Box_{a\in A}(a \to R_a),$$

where $\mathcal{A} = \mathrm{con}(\{\, B \cup C \mid B \in \mathcal{B}, C \in \mathcal{C} \,\})$.

The following result states that every true statement about normal forms is provable.

Lemma 5.2.6: Given two normal forms P^* and Q^*,

$$\mathcal{F}[\![P^*]\!] \supseteq \mathcal{F}[\![Q^*]\!] \Rightarrow \vdash P^* \sqsubseteq Q^*.$$

Proof. Let the two normal forms be

$$P^* = \sqcap_{B\in\mathcal{B}}\Box_{b\in B}(b \to P_b)$$
$$Q^* = \sqcap_{C\in\mathcal{C}}\Box_{c\in C}(c \to Q_c).$$

Write P_B and Q_C for the subterms:

$$P_B = \Box_{b \in B}(b \to P_b),$$
$$Q_C = \Box_{c \in C}(c \to Q_c).$$

Then $P^* = \sqcap_{B \in \mathcal{B}} P_B$ and $Q^* = \sqcap_{C \in \mathcal{C}} Q_C$.

By definition of normal form, the sets $\mathcal{B}$ and $\mathcal{C}$ are convex, and each P_b and Q_c is also in normal form. We will use an induction on the length of the normal forms. The base case, when both P and Q have zero length, is trivial; both terms are STOP. For the inductive step, we argue as follows. First we show that

$$\mathcal{F}[\![P^*]\!] \supseteq \mathcal{F}[\![Q^*]\!] \Rightarrow C \subseteq \mathcal{B} \ \& \ \forall c \in C \in \mathcal{C}.\ \mathcal{F}[\![P_c]\!] \supseteq \mathcal{F}[\![Q_c]\!] \quad (1).$$

To this end, assume that $\mathcal{F}[\![P^*]\!] \supseteq \mathcal{F}[\![Q^*]\!]$. Let $B_0 = \bigcup \mathcal{B}$ and $C_0 = \bigcup \mathcal{C}$ be the initials of P^* and Q^*. Then we know that

$$B_0 \supseteq C_0.$$

Since P^* and Q^* have unique c-derivatives P_c and Q_c respectively, for all $c \in B_0 \cap C_0$, we must have

$$\mathcal{F}[\![P_c]\!] \supseteq \mathcal{F}[\![Q_c]\!]$$

for all such c. All we need to show now is that $C \subseteq \mathcal{B}$. If this does not hold, let $X = B_0 - C$. Then $(\langle\rangle, X)$ must be a failure of P^*. By hypothesis, this is also a failure of Q^*. But this happens only if there is a $B \in \mathcal{B}$ with

$$B \cap X = B \cap (B_0 - C) = \emptyset.$$

Equivalently, $B \subseteq C$. But $C \subseteq C_0 \subseteq B_0$, and the sets B and B_0 belong to $\mathcal{B}$ (B by assumption and B_0 by convexity). Thus we find that $C \in \mathcal{B}$, contradicting our assumption. It must therefore be the case that $C \subseteq \mathcal{B}$, as required. The truth of (1) has now been established.

Now the inductive hypothesis applied to the terms P_c and Q_c gives

$$\vdash P_c \sqsubseteq Q_c,$$

for all $c \in \bigcup C$. This implies that, for each $C \in \mathcal{C}$,

$$\vdash P_C \sqsubseteq Q_C.$$

Then, since $C \subseteq \mathcal{B}$, we may use (A10) and (M2) to show

$$\vdash P^* \sqsubseteq Q^*,$$

as required. That completes the proof. ∎

Corollary 5.2.7: For all terms P and Q,

$$\mathcal{F}[\![P]\!] \supseteq \mathcal{F}[\![Q]\!] \Rightarrow \vdash P \sqsubseteq Q.$$

Proof. By Lemmas 5.2.5 and 5.2.6. ∎

3. Extending to infinite processes

In this section we modify the language FCSP, adding process variables, recursion and a new constant $\perp$, which is intended to denote a process whose only capability is to diverge. Such a pathological process will turn out to correspond precisely to the terms in which a badly constructed recursion appears. We will be mainly interested in terms without free process variables, so-called *closed terms*. The semantics we use for this language is based on failure sets but has an extra component called a *divergence set* in order to allow us to distinguish between deadlock and divergence.

Let RCSP (Recursive CSP) be the language generated by the following syntax:

$$P ::= \text{STOP} \mid (a \to P) \mid P \,\Box\, P \mid P \sqcap P \mid \perp \mid x \mid \mu x.P$$

where $a \in \Sigma$ and x ranges over a set of process variables or *identifiers*.

Let $\mathbf{F}$ be the domain of failure sets, ordered by $\supseteq$. Now we introduce $\mathbf{D}$, the domain of *divergence sets,* which is just $p(\Sigma^*)$, ordered also by $\supseteq$. The semantics of terms in RCSP will be given via two semantic functions, one for failures and one for divergence. Since terms may contain occurrences of identifiers we will use an *environment* in the semantics, which binds each identifier to the failure set and divergence set it is intended to denote. Let Ide be the set of identifiers. Then the domain of environments is

$$U = \text{Ide} \to (\mathbf{F} \times \mathbf{D}).$$

For an environment u which maps identifiers to pairs, we will use the conventional notation $(u[\![x]\!])_1$ and $(u[\![x]\!])_2$ to refer to the components of pairs.

The semantic function $\mathcal{D}$ maps terms to divergence sets, relative to an environment. It is defined in the usual way, by structural induction:

Definition 5.3.1: The divergence semantic function is:

$$\begin{aligned}
\mathcal{D} &: \text{RCSP} \to U \to \mathbf{D} \\
\mathcal{D}[\![\text{STOP}]\!]u &= \emptyset \\
\mathcal{D}[\![a \to P]\!]u &= \{\, as \mid s \in \mathcal{D}[\![P]\!]u \,\} \\
\mathcal{D}[\![P \,\Box\, Q]\!]u &= \mathcal{D}[\![P]\!]u \cup \mathcal{D}[\![Q]\!]u \\
\mathcal{D}[\![P \sqcap Q]\!]u &= \mathcal{D}[\![P]\!]u \cup \mathcal{D}[\![Q]\!]u \\
\mathcal{D}[\![\perp]\!]u &= \Sigma^* \\
\mathcal{D}[\![x]\!]u &= (u[\![x]\!])_2 \\
\mathcal{D}[\![\mu x.P]\!]u &= \text{fix}(\lambda\delta.\mathcal{D}[\![P]\!](u + [x \mapsto \delta]))
\end{aligned}$$

It is easy to see that all of the operations induced on divergence sets by the above definitions are continuous with respect to the superset ordering and hence that the fixed point used in the semantics of recursion will always exists. The usual fixpoint characterisation as a limit is expressed in:

$$\begin{aligned} \mathcal{D}[\![\mu x.P]\!]u &= \bigcap_{n=0}^{\infty} \delta_n, \\ \text{where} \quad \delta_0 &= \Sigma^* = \mathcal{D}[\![\bot]\!]u, \\ \text{and} \quad \delta_{n+1} &= \mathcal{D}[\![P]\!](u + [x \mapsto \delta_n]) \quad \text{for } n \geq 0. \end{aligned}$$

Notice also that the only terms with a non-trivial divergence set are those with a subterm $\bot$ or with an *unguarded* recursion, i.e., a subterm of the form $\mu x.P$ in which there is an occurrence of x appearing in P without a guard. This fact could be proved by a structural induction, once we have defined rigorously the notion of well-guardedness. Finally, our definition guarantees that whenever a particular trace s belongs to a divergence set then all extensions of that trace are also included:

$$s \in \mathcal{D}[\![P]\!]u \Rightarrow st \in \mathcal{D}[\![P]\!]u, \quad \text{for all } t.$$

Example 1. The recursion $\mu x.(a \rightarrow x)$ is guarded. Applying the previous definition, we have

$$\begin{aligned} \mathcal{D}[\![\mu x.(a \rightarrow x)]\!]u &= \cap_{n=0}^{\infty} \delta_n, \\ \text{where} \quad \delta_0 &= \Sigma^*, \\ \text{and, for each } n, \quad \delta_{n+1} &= \{\, as \mid s \in \delta_n \,\}. \end{aligned}$$

Thus $\delta_n = a^n\Sigma^*$, for each n, and the intersection of these sets is empty: $\mathcal{D}[\![\mu x.(a \rightarrow x)]\!]u$ is the empty set.

Example 2. The recursion $\mu x.x$ is obviously not well-guarded.

$$\mathcal{D}[\![\mu x.x]\!]u = \Sigma^*.$$

Example 3. The term $\mu x.((a \rightarrow x) \,\Box\, (\mu y.y))$ is not well-guarded, because of the subterm $\mu y.y$.

The semantic function for failures is also given by structural induction, and it makes use of $\mathcal{D}$. For the most part, the definition is along the lines of the previous chapters, but extended to make it consistent with the notion that divergence is catastrophic: when a process is diverging we can guarantee no

aspect of its behaviour; thus we make the operations *strict*, so that a process constructed from divergent components can diverge too.

Definition 5.3.2: The failures semantic function is:

$$\mathcal{F} : \mathrm{RCSP} \to U \to \mathbf{F}$$

$$\mathcal{F}[\![\mathrm{STOP}]\!]u = \{(\langle\rangle, X) \mid X \subseteq \Sigma\}$$

$$\mathcal{F}[\![\bot]\!]u = \Sigma^* \times p\Sigma$$

$$\mathcal{F}[\![a \to P]\!]u = \{(\langle\rangle, X) \mid a \notin X\} \cup \{(as, X) \mid (s, X) \in \mathcal{F}[\![P]\!]u\}$$

$$\begin{aligned}
\mathcal{F}[\![P \,\Box\, Q]\!]u &= \Sigma^* \times p\Sigma, \qquad \text{if } \langle\rangle \in \mathcal{D}[\![P \,\Box\, Q]\!]u, \\
\mathcal{F}[\![P \,\Box\, Q]\!]u &= \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \mathcal{F}[\![P]\!]u \cap \mathcal{F}[\![Q]\!]u\} \\
&\qquad \cup \{(s, X) \mid s \neq \langle\rangle \ \&\ (s, X) \in \mathcal{F}[\![P]\!]u \cup \mathcal{F}[\![Q]\!]u\} \\
&\qquad\qquad \text{otherwise}
\end{aligned}$$

$$\mathcal{F}[\![P \sqcap Q]\!]u = \mathcal{F}[\![P]\!]u \cup \mathcal{F}[\![Q]\!]u$$

$$\mathcal{F}[\![x]\!]u = (u[\![x]\!])_1$$

$$\mathcal{F}[\![\mu x.P]\!]u = \mathrm{fix}(\lambda\phi.\mathcal{F}[\![P]\!](u + [x \mapsto \phi]))$$

Again we can prove that all operations on failure sets used here are continuous, and therefore the least fixed point of any construction exists and is given by the limit:

$$\begin{aligned}
\mathcal{F}[\![\mu x.P]\!]u &= \bigcap_{n=0}^{\infty} \Phi_n, \\
\text{where} \quad \Phi_0 &= \Sigma^* \times p\Sigma, \\
\text{and} \quad \Phi_{n+1} &= \mathcal{F}[\![P]\!](u + [x \mapsto \Phi_n]).
\end{aligned}$$

We say that P may diverge on s if s is a trace in the divergence set of P. Notice that we have defined the semantics of terms in such a way that the following conditions hold:

$$\begin{aligned}
&\text{(i)} \quad s \in \mathcal{D}[\![P]\!]u \Rightarrow \forall t, X.(st, X) \in \mathcal{F}[\![P]\!]u. \\
&\text{(ii)} \quad s \in \mathcal{D}[\![P]\!]u \Rightarrow \forall t.st \in \mathcal{D}[\![P]\!]u.
\end{aligned}$$

Intuitively, (i) says that a divergent process is totally unpredictable: we cannot be sure that it will or will not ever stop diverging and allow some sequence of actions. Condition (ii) says that once a process starts to diverge it cannot "recover" by performing a visible action: divergence continues forever. Thus, a pair (Φ, δ) is a reasonable model for a process iff the following conditions hold:

$$
\begin{array}{lrl}
(1) & & \mathrm{dom}(\Phi) \text{ is non-empty and prefix-closed} \\
(2) & (s,X) \in \Phi,\ Y \subseteq X & \Rightarrow (s,Y) \in \Phi \\
(3) & (s,X) \in \Phi,\ (sa,\emptyset) \notin \Phi & \Rightarrow (s, X \cup \{a\}) \in \Phi \\
(4) & s \in \delta & \Rightarrow st \in \delta, \quad \text{for all } t \\
(5) & s \in \delta & \Rightarrow (st, X) \in \Phi, \quad \text{for all } t, X.
\end{array}
$$

This more general model of processes is thus seen to be derived from the old failures model by adding divergence sets and requiring a kind of *consistency* between failures and divergences. Indeed, the processes with empty divergence sets form a space isomorphic to the failures model. Notice that the limit of a directed set of pairs (Φ_i, δ_i) is the intersection and the greatest lower bound of a non-empty set of pairs is again the union. As with the set of failures, the new model forms a complete semi-lattice with respect to the (pairwise) superset ordering. We will write $P \sqsubseteq Q$ to mean that the failures of P contain those of Q and the divergence set of P contains the divergence set of Q. All of the operations considered in this section are continuous with respect to this ordering. This fact justifies our use of fixpoints in the semantics of recursively defined processes.

Syntactic approximation.

Before we introduce a complete axiom system for the new model, we will need some important results which allow us to reason about a (possibly) infinite process in terms of its (finite) approximations. Beginning with the standard definition of *syntactic approximation* on terms, we define the set of finite approximants of an arbitrary term and show that the semantics of any term is uniquely determined from the semantics of its finite approximants.

The notion of *syntactic approximation* on terms is well known (see, for example, [Gu]). The following presentation is typical of the general style.

Definition 5.3.3: The relation $\prec$ on terms is the smallest relation satisfying:

(i) $\bot \prec P$

(ii) $P \prec P$

(iii) $P \prec Q \prec R \Rightarrow P \prec R$

(iv) $P \prec Q \Rightarrow (a \rightarrow P) \prec (a \rightarrow Q)$

(v) $P_1 \prec Q_1,\ P_2 \prec Q_2 \Rightarrow P_1 \,\Box\, P_2 \prec Q_1 \,\Box\, Q_2$

(vi) $P_1 \prec Q_1,\ P_2 \prec Q_2 \Rightarrow P_1 \sqcap Q_1 \prec P_2 \sqcap Q_2$

(vii) $P[(\mu x.P) \setminus x] \prec \mu x.P$

We have used the notation $P[Q \setminus x]$ to denote the result of replacing every free occurrence of x in P by Q, taking care to avoid name clashes.

If $P \prec Q$ we say that P *approximates* Q. An easy structural induction shows that for all P and Q syntactic approximation implies semantic approximation:

$$P \prec Q \;\Rightarrow\; P \sqsubseteq Q.$$

A term P is *finite* iff it does not contain any subterm of the form $\mu x.Q$. For any term P, the set of finite approximants is

$$\mathrm{FIN}(P) = \{\, Q \mid Q \prec P \;\&\; Q \text{ is finite} \,\}.$$

It should be noted that $\mathrm{FIN}(P)$ is *directed* with respect to $\prec$.

The common notion of *unrolling* or *unwinding* a recursive term is intimately connected with finite approximation. The result of unrolling the term P n times will be denoted $P^{(n)}$. The formal definition is:

(i) $P^{(0)} = \bot$

(ii) $\mathrm{STOP}^{(n+1)} = \mathrm{STOP}$

(iii) $(a \rightarrow P)^{(n+1)} = (a \rightarrow P^{(n+1)})$

(iv) $(P \,\Box\, Q)^{(n+1)} = P^{(n+1)} \,\Box\, Q^{(n+1)}$

(v) $(P \sqcap Q)^{(n+1)} = P^{(n+1)} \sqcap Q^{(n+1)}$

(vi) $x^{(n+1)} = x$

(vii) $(\mu x.P)^{(n+1)} = P[(\mu x.P)^{(n)} \setminus x].$

Every finite approximation to a term P is also a finite approximation to some unrolling of that term:

Lemma 5.3.4: If $Q \in \mathrm{FIN}(P)$ then there is an n such that $Q \prec P^{(n)}$.

Proof. See [Gu]. ∎

Corollary: For all P, $\mathrm{FIN}(P) = \bigcup_{n=0}^{\infty} \mathrm{FIN}(P^{(n)})$.

Lemma 5.3.5:

(i) $\mathrm{FIN}(\bot) = \{\bot\}$
(ii) $\mathrm{FIN}(\mathrm{STOP}) = \{\bot, \mathrm{STOP}\}$
(iii) $\mathrm{FIN}(P \,\Box\, Q) = \{\bot\} \cup \{P' \,\Box\, Q' \mid P' \in \mathrm{FIN}(P) \;\&\; Q' \in \mathrm{FIN}(Q)\}$
(iv) $\mathrm{FIN}(P \sqcap Q) = \{\bot\} \cup \{P' \sqcap Q' \mid P' \in \mathrm{FIN}(P) \;\&\; Q' \in \mathrm{FIN}(Q)\}$
(v) $\mathrm{FIN}(a \to P) = \{\bot\} \cup \{(a \to P') \mid P' \in \mathrm{FIN}(P)\}$
(vi) $\mathrm{FIN}(x) = \{\bot, x\}$.

Proof. Elementary. ∎

In a sense, a term P is the "syntactic limit" of its finite approximation set $\mathrm{FIN}(P)$. Recall that this set is directed with respect to the syntactic relation $\prec$, and therefore the semantic images of the finite approximations to P form a directed set with respect to the semantic order $\sqsubseteq$. The following results show that the semantics of a term is uniquely determined by the semantics of its finite approximations, and allow us to deduce that the semantics of a term P is in fact the limit of the semantics of its finite approximations. We omit proofs, as they follow standard lines. The reader might remember that in Chapter 1 the failure set of an arbitrary process was shown to be derivable as a limit of failure sets of finite processes. Here we are obtaining analogous results for the extended model.

Lemma 5.3.6: If P is finite and $P \sqsubseteq Q$, then there is a finite approximation R of Q such that $P \sqsubseteq R$.

Theorem 5.3.7: For all P and u,

$$\mathcal{D}[\![P]\!]u = \bigcap\{\mathcal{D}[\![Q]\!]u \mid Q \in \mathrm{FIN}(P)\}.$$

Proof. By structural induction on P. ∎

Theorem 5.3.8: For all P and u,

$$\mathcal{F}[\![P]\!]u = \bigcap\{\mathcal{F}[\![Q]\!]u \mid Q \in \mathrm{FIN}(P)\}.$$

Proof. By structural induction. ∎

Proof system.

Let $\mathcal{L}$ be the proof system containing all axioms and rules of the earlier

system together with the following additions:

$$\begin{array}{ll} \text{(B1)} & P \,\Box\, \bot = \bot \\ \text{(B2)} & P \sqcap \bot = \bot \\ \text{(B3)} & \bot \sqsubseteq P \\ \text{(B4)} & P[(\mu x.P) \setminus x] \sqsubseteq \mu x.P \end{array}$$

$$\text{(R)} \qquad \frac{\forall Q \in \mathrm{FIN}(P).\, Q \sqsubseteq R}{P \sqsubseteq R}$$

The new axioms state that the two conditional combinators are *strict,* and that $\bot$ is the bottom element with respect to $\sqsubseteq$. The new inference rule essentially says that any property of a term is deducible from the properties of its finite approximations. This is an *infinitary rule,* because a term may have an infinite set of syntactic approximants; in such a case one would need an infinite number of premises in order to use rule (R). It seems unlikely that a finitary proof system could be found which was still complete, although some interesting sublanguages (in which use of recursion is constrained) will presumably have decidable proof systems. This remains a topic for future work. Now we are concerned with the soundness and completeness of our enlarged proof system.

Soundness.

Under the interpretation that for closed terms P and Q, $P \sqsubseteq Q$ means

$$\mathcal{F}[\![P]\!]u \supseteq \mathcal{F}[\![Q]\!]u \;\&\; \mathcal{D}[\![P]\!]u \supseteq \mathcal{D}[\![Q]\!]u$$

for all environments u, the proof system $\mathcal{L}$ is sound. We need merely to check that the axioms are valid and the proof rules sound. Since the semantics was defined to make the conditional operators strict, the new axioms are clearly valid. Soundness of rule (R) follows from Theorems 5.3.7 and 5.3.8. It is easy to check validity of the old axioms and rules. Thus we have:

Theorem 5.3.9: For all closed terms P and Q, and all u,

$$\vdash_{\mathcal{L}} P \sqsubseteq Q \Rightarrow P \sqsubseteq Q.$$

Completeness.

In order to establish that the new proof system is complete, we must first modify the definition of normal form. Essentially, we just allow $\bot$ as well as STOP in building up normal forms.

Definition 5.3.10: A term P in RCSP is in normal form iff it has the structure:

$$\begin{array}{ll} \text{either} & P = \text{STOP}, \\ \text{or} & P = \bot, \\ \text{or} & P = \sqcap_{B \in \mathcal{B}} \Box_{b \in B} (b \rightarrow P_b) \end{array}$$

where $\mathcal{B}$ is convex and each P_b is in normal form.

It is easy to modify the proof of Lemmas 5.2.5 and 5.2.6 to show that any *finite* term can be reduced to normal form using the axioms and rules, and that whenever P and Q are normal forms

$$P \sqsubseteq Q \Rightarrow \vdash_{\mathcal{L}} P \sqsubseteq Q.$$

The completeness theorem relies on Lemma 5.3.6, which states that whenever P is finite and $P \sqsubseteq Q$ there is a finite term $R \in \text{FIN}(Q)$ such that $P \sqsubseteq R$.

Theorem 5.3.11: For all terms P and Q,

$$P \sqsubseteq Q \Rightarrow \vdash_{\mathcal{L}} P \sqsubseteq Q.$$

Proof. Let P' be a finite approximation to P and suppose $P \sqsubseteq Q$. Then

$$P' \sqsubseteq P \sqsubseteq Q.$$

By Lemma 5.3.6 there is a finite approximation Q' to Q such that $P' \sqsubseteq Q'$. But then

$$\vdash P' \sqsubseteq Q'.$$

Since for every $Q' \in \text{FIN}(Q)$ the formula $Q' \sqsubseteq Q$ is provable, we have

$$\vdash P' \sqsubseteq Q.$$

The result follows by an application of rule (R). ∎

4. Adding more CSP operations

We may extend the proof system to encompass other CSP operations provided we add enough axioms and rules to allow normal form reductions.

To stay within the framework of this chapter, we must introduce failure sets and divergence sets for the new forms of processes, by extending the definition of $\mathcal{D}$ and $\mathcal{F}$ accordingly. We must also add axioms and inference rules corresponding to these definitions, in such a way that Theorems 5.3.7 and 5.3.8 still hold. This will keep the proof system complete and consistent.

In keeping with the notion that a divergent process is totally unpredictable, and that divergence of a component process should also give rise to divergence of the compound process (so that a process built from divergent components diverges) we stipulate that all operations should be *strict,* in that they map $\perp$ to $\perp$. The only exception to this rule is the prefixing operation, for obvious reasons. Now we consider extending the proof system and the semantic definitions to include parallel composition, interleaving, and hiding. It should be clear how to include the other operations of Chapter 2, with these examples as illustration of the general method. Essentially, we make each operation strict, and include axioms for strictness and for distribution over $\sqcap$ and guarded terms.

Parallel composition.

For the parallel composition $P \parallel Q$, for example, we require divergence when either P or Q diverges, and thus we specify:

(i) $\mathcal{D}[\![P \parallel Q]\!]u = \{\, st \mid s \in (\mathcal{D}[\![P]\!]u \cup \mathcal{D}[\![Q]\!]u) \cap (\mathrm{traces}(P) \cap \mathrm{traces}(Q)) \,\}$

(ii) $\mathcal{F}[\![P \parallel Q]\!]u = \{\, (s, X \cup Y) \mid (s,X) \in \mathcal{F}[\![P]\!]u \ \&\ (s,Y) \in \mathcal{F}[\![Q]\!]u \,\}$
$\cup \{\, (st, X) \mid s \in \mathcal{D}[\![P \parallel Q]\!]u \,\}.$

Extending the syntactic approximation relation in the obvious way, we add the clause

$$P_1 \prec P_2,\ Q_1 \prec Q_2 \Rightarrow P_1 \parallel Q_1 \prec P_2 \parallel Q_2$$

to Definition 5.3.3. Then the finite approximations of $P \parallel Q$ are built up as parallel compositions of finite approximations of P and Q :

$$\mathrm{FIN}(P \parallel Q) = \{\, \perp \,\} \cup \{\, P' \parallel Q' \mid P' \in \mathrm{FIN}(P) \ \&\ Q' \in \mathrm{FIN}(Q) \,\}.$$

It is clear from this that Theorems 5.3.7 and 5.3.8 still hold.

We add axioms for strictness and manipulation of normal forms. In each case the axiom is either a restatement of an earlier result which clearly still holds in the extended model, or is self-evident. In the axioms we adopt the convention that a term P_B stands for

$$\Box_{b \in B}(b \to P_b).$$

(PAR 0) $$P \| \bot \equiv \bot$$

(PAR 1) $$P \| (Q \sqcap R) \equiv (P \| Q) \sqcap (P \| R)$$

(PAR 2) $$P_B \| Q_C \equiv \Box_{a \in B \cap C}(a \to P_a \| Q_a).$$

In the special case when B is empty the last of these axioms should be interpreted $\mathrm{STOP} \| Q_C = \mathrm{STOP}$. It is easy to check that these axioms enable any parallel composition of normal forms to be reduced to normal form.

Interleaving. For the interleaving operation $P|||Q$ we add:

(i) $$\mathcal{D}[\![P|||Q]\!]u = \mathrm{merge}(\mathcal{D}[\![P]\!]u, \mathcal{D}[\![Q]\!]u)$$

(ii) $$\mathcal{F}[\![P|||Q]\!]u = \{(u, X) \mid \exists s, t.\, (s, X) \in \mathcal{F}[\![P]\!]u \ \&\ (t, X) \in \mathcal{F}[\![Q]\!]u \ \&\ u \in \mathrm{merge}(s, t)\} \cup \{(st, X) \mid s \in \mathcal{D}[\![P|||Q]\!]u\}.$$

For syntactic approximation we add to Definition 5.3.3:

$$P_1 \prec P_2,\ Q_1 \prec Q_2 \Rightarrow P_1|||Q_1 \prec P_2|||Q_2.$$

Again the finite approximations of an interleaved process are formed by interleaving finite approximations to the components:

$$\mathrm{FIN}(P|||Q) = \{\bot\} \cup \{P'|||Q' \mid P' \in \mathrm{FIN}(P)\ \&\ Q' \in \mathrm{FIN}(Q)\}.$$

Again Theorems 5.3.7 and 5.3.8 are still true.

Our definition yields a strict operation, since $\mathcal{D}[\![\bot|||Q]\!]u = \Sigma^*$. Otherwise it has similar properties to our earlier interleaving operation. We add axioms:

(INT 0) $$P|||\bot \equiv \bot$$

(INT 1) $$P|||\mathrm{STOP} \equiv P$$

(INT 2) $$P|||(Q \sqcap R) \equiv (P \sqcap Q)|||(P \sqcap R)$$

(INT 3) $$P_B|||Q_C \equiv (\Box_{b \in B}(b \to (P_b|||Q_C))) \,\Box\, (\Box_{c \in C}(c \to (P_B|||Q_c))).$$

Again it is easy to check the validity of these axioms, and to verify that an interleaving of two normal forms can be reduced to normal form.

Hiding. For the hiding operator, we will take the view that hiding a potentially infinite sequence of actions produces divergence: we will identify the phenomenon of infinite internal chatter with divergence. This version of hiding is closely related to the second form of hiding introduced in Chapter 2, where infinite chatter was identified with CHAOS; here a process which is chattering has the same failure set as CHAOS, but (unlike CHAOS) can also diverge. It is simple to alter the proofs given earlier for the chaotic version of hiding, to show that this form enjoys similar properties, such as continuity.

(i) $\mathcal{D}[\![P/b]\!]u = \{(s\backslash b)t \mid s \in \mathcal{D}[\![P]\!]u\} \cup \{(s\backslash b)t \mid \forall n.\ sb^n \in \text{traces}(P)\}.$

(ii) $\mathcal{F}[\![P/b]\!]u = \{(s\backslash b, X) \mid (s, X \cup \{b\}) \in \mathcal{F}[\![P]\!]u\} \cup \{(st, X) \mid s \in \mathcal{D}[\![P/b]\!]u\}.$

For finite approximations, we again add to Definition 5.3.3:

$$P' \prec P \Rightarrow P'/b \prec P/b.$$

The finite approximations to a process formed by hiding are again formed by hiding:

$$\text{FIN}(P/b) = \{\bot\} \cup \{P'/b \mid P' \in \text{FIN}(P)\}.$$

Our new hiding operator is strict. We add axioms:

(HIDE 0) $\quad \bot/b \equiv \bot$

(HIDE 1) $\quad (P \sqcap Q)/b \equiv (P/b) \sqcap (Q/b)$

(HIDE 2) $\quad (b \to P)/c \equiv (b \to P/c) \quad \text{if } b \neq c,$
$\qquad\qquad\qquad\quad \equiv P/c \quad \text{if } b = c.$

(HIDE 3) $\quad P_B/c \equiv \Box_{b \in B}((b \to P_b)/c), \quad \text{if } c \notin B,$
$\qquad\qquad\qquad\quad \equiv (P_c/c) \sqcap \Box_{b \in B}((b \to P_b)/c) \quad \text{if } b \in C.$

Again the validity of these axioms is easy to check, and one can use the axioms to produce normal forms.

Examples.

Example 1. The term $P = \mu x.(a \to x)$ has finite approximations

$$P_n = (a^n \to \bot), \qquad \text{for all } n,$$

using the obvious abbreviations. Thus, the term P/a has finite approximations

$$\perp, \text{ and } P_n/a = (a^n \rightarrow \perp)/a,$$

for all n. Using (HIDE 2) we see that, for each n,

$$\vdash (a^n \rightarrow \perp)/a \equiv \perp/a,$$

and so, by (HIDE 1) every finite approximation to P/a is provably equivalent to $\perp$. By rule (R), it follows that P/a is equivalent to $\perp$, as expected because P/a diverges.

Chapter 6

Testing processes

0. Introduction

This chapter relates the failures model of processes with other work, notably by Hennessy and de Nicola [HN] and by Kennaway [K1,2]. These authors have suggested various notions of *testing* a process. The basic idea is that a program or process can be investigated by applying a series of tests. The general situation can be expressed as follows. There is a set of processes and a set of relevant tests. Two processes are equivalent if they pass exactly the same tests. Different choices of the appropriate set of tests, and what it means for a process to satisfy or pass a test, lead to different notions of process equivalence. We will describe the work of Hennessy and de Nicola, and then show how their ideas can be related to ours. It turns out that there is a very close connection; this is especially interesting, since their motivation for constructing tests and their notions of passing tests were independent of ours, and might not appear related at first glance. Similar results are then given relating our work to the work of Kennaway.

It will simplify the comparison considerably if we work entirely in CCS. In view of the results of Chapter 4, this can be done without loss of generality. We will, therefore, identify processes with synchronisation trees throughout this chapter. Initially we will only consider *divergence-free* trees, since this makes it easier to see the relationship between failure equivalence and the various equivalences based on tests. The notion of divergence first arose in Chapter 2, when we defined two versions of a hiding operator. We saw that hiding an action which can be performed arbitrarily many times gives rise to an undesirable phenomenon in which the process may not be able to respond to its environment because it is engaging in so-called *infinite internal chatter*. In our synchronisation tree representation of processes this situation corresponds to the presence of an infinite path, all of whose labels are τ. A synchronisation tree will be divergence-free iff none of its subtrees has a divergent path.

1. Divergence-free terms

We begin by defining the divergence-free terms of the version of CCS to be used in this chapter. As usual, we use S, T, and U as meta-variables to range over terms. Recall that the CCS terms are given by the grammar:

$$S ::= \text{NIL} \mid aS \mid S_1 + S_2 \mid x \mid \mu x.S$$

where x ranges over a set of identifiers and a ranges over the extended alphabet $\Sigma \cup \{\tau\}$. A term is *closed* if it has no free identifier occurrences. A recursive term $\mu x.S$ is *well-guarded* iff every free occurrence of x in the body S is guarded by (at least) one visible label. Intuitively, a term *diverges* if it has an unguarded recursion. We make this notion precise by defining a syntactic condition of *well-formedness*. A closed term is well-formed iff all of its recursive subterms are well-guarded. In formalising these notions, we begin with guarding.

Definition 6.1.1: The identifier x is guarded by s in term T if and only if one of the following conditions holds:

(i)	$s = \langle\rangle$	and	$T = x$
(ii)	$s = at, T = aS$	and	x is guarded by t in S
(iii)	$T = S_1 + S_2$	and	x is guarded by s in S_1 or S_2
(iv)	$T = \mu y.S,\ x \neq y$	and	x is guarded by s in S

For example, x occurs guarded by $\langle\rangle$ in $(a\text{NIL} + x)$ and by a and τ in $(ax + \tau x)$.

Say an occurrence of x in T is *properly guarded* iff all of its guards contain at least one visible action. Thus there is an improperly guarded occurrence of x in the term $(ax + \tau x)$.

Definition 6.1.2:

(1) A term T is *well-guarded for* x iff every free occurrences of x in T is properly guarded.

(2) A term is *well-formed* iff it is closed and in every subterm of the form $\mu x.S$ S is well-guarded for x.

For example, the term $\mu x.ax + bx$ is well-formed but $\mu x.ax + x$ and $\mu x.(ax + \tau x)$ are not.

The synchronisation tree formed by unrolling a well-formed term will be finitely branching and have no infinite paths of τ-arcs; proof of this is left to

the reader. From now until further notice, we will be dealing with well-formed terms and the trees they denote. We may refer to such trees and terms as *divergence-free.*

We refer the reader to Chapter 4 for definition of the traces, failures and derivations of a synchronisation tree. The notation introduced there will be used heavily throughout this chapter.

2. Testing processes

Hennessy and de Nicola set up a general framework for discussing processes and tests upon them. Beginning with a predefined set of *states,* a *computation* is defined to be a non-empty sequence of states. Given a set of processes and a set of tests, they formalise the effect of performing a test on a process by associating a set of computations with every process and test: the result of testing a process will then be one of these computations. To indicate that a process *passes* a test a subset of *successful* states is distinguished. A computation is *successful* if and only if it contains a successful state; otherwise, the computation is *unsuccessful.* To cope with partially defined processes they allow partial states, states whose properties are, in a sense, incompletely specified. They assume the existence of a unary predicate $\uparrow$ on states, which distinguishes the partial states from the complete states. They then define *divergence,* a unary predicate on computations, denoted by $\Uparrow$: a computation diverges if either it is unsuccessful or it contains a partial state which is not preceded by a successful state. The converse predicate, *convergence* will be denoted $\Downarrow$. Our divergence-free terms never give rise to divergent computations, in their definition.

Hennessy and de Nicola choose their tests in a natural way; an observer may test a process by attempting to communicate with it. Since CCS was designed to express communication, it seems reasonable to use the same language in describing tests. Moreover, an observer can only be expected to be capable of a finite number of communication attempts in any finite amount of time. This is tantamount to requiring a test to be of finite depth and finitely branching. In addition, however, one needs some way of indicating the success or failure of a test; their method, which we adopt, is to use a special, distinguished action ω which is interpreted as "reporting success". Thus, an *observer* or *test* will be a term described by the following simple grammar. We will use the meta-variable O to range over observers.

$$O ::= \text{NIL} \mid aO \mid O_1 + O_2$$

where a ranges over $\Sigma \cup \{\omega, \tau\}$. For convenience, we will assume that the event ω has no complement. *Example* . The term $\overline{a}\overline{b}\omega$NIL is an observer for testing whether or not a process can perform an a-action followed by a b-action.

Intuitively, a process S passes a test O if when S and O are composed in parallel success can eventually be reported. In this model the *states* are simply CCS terms obtained by the parallel composition of a process and a test. A change of state comes about when a τ-transition occurs, and a *computation* is a sequence of terms T_n (possibly infinite) such that for each pair of successive terms T_n and T_{n+1} we have $T_n \xrightarrow{\tau} T_{n+1}$, and such that if T_{n+1} is not defined then T_n has no τ-transitions. A successful state is one in which an ω-action is possible.

A reasonable formal definition of passing a test is the following; this was Hennessy and de Nicola's original attempt.

Definition 6.2.1:

(i) $S \underline{\text{may}}\ O \quad \Leftrightarrow \exists U.\, (S|O) \stackrel{\tau}{\Longrightarrow} U \ \&\ (U \xrightarrow{\omega})$.

(ii) $S \underline{\text{must}}\ O \quad \Leftrightarrow \forall U.\, (S|O) \stackrel{\tau}{\Longrightarrow} U \ \&\ \neg(U \xrightarrow{\tau}) \ \Rightarrow (U \xrightarrow{\omega})$
$\&\ (S|O)\Downarrow$.

In the absence of divergence the clause $(S|O)\Downarrow$ makes no contribution. The finiteness of tests and our insistence that processes be divergence-free means that the possibility of divergent computations never arises when testing a process.

According to this definition, a process *may* pass a test if and only if at least one computation arising from the test succeeds; and a process *must* pass a test if and only if every computation arising from the test terminates in a successful state. Note the requirement that all computations must *terminate*. As we will see later, there are variants of these definitions which allow more freedom in testing; in particular, it might be considered appropriate to allow an observer to detect success without requiring that the process halt.

These definitions can be conveniently reformulated in terms of failures. We omit the proof, which is obvious.

Lemma 6.2.2:

(i) $S \underline{\text{may}}\ O \quad \Leftrightarrow \omega \in \text{traces}(S|O)$
$\Leftrightarrow (\langle\omega\rangle, \emptyset) \in \text{failures}(S|O)$

(ii) $S \underline{\text{must}}\ O \quad \Leftrightarrow \{\omega\} \notin \text{refusals}(S|O)$
$\Leftrightarrow (\langle\,\rangle, \{\omega\}) \notin \text{failures}(S|O)$.

Thus, a process S *may* pass a test O iff a possible trace of $S|O$ records success; and S *must* pass O iff $S|O$ cannot refuse ω. No matter how the process makes its nondeterministic choices of refusals, it cannot avoid success.

Following Hennessy and de Nicola, we define three preorders on terms:

Definition 6.2.3: The preorders $\sqsubseteq_i$ on terms, $(i = 1, 2, 3)$ are defined by:

$$\text{(a)} \quad S \sqsubseteq_3 T \;\Leftrightarrow\; \forall O.\, S \underline{\text{may}}\ O \Rightarrow T \underline{\text{may}}\ O$$

$$\text{(b)} \quad S \sqsubseteq_2 T \;\Leftrightarrow\; \forall O.\, S \underline{\text{must}}\ O \Rightarrow T \underline{\text{must}}\ O$$

$$\text{(c)} \quad S \sqsubseteq_1 T \;\Leftrightarrow\; S \sqsubseteq_2 T \;\&\; S \sqsubseteq_3 T.$$

They also single out for investigation the preorders corresponding to restriction of the tests to some particular class of observer. If θ is a set of observers, then the preorders *generated* by θ are defined to be:

$$\text{(a)} \quad S \sqsubseteq_3^{\theta} T \;\Leftrightarrow\; \forall O \in \theta.\, S \underline{\text{may}}\ O \Rightarrow T \underline{\text{may}}\ O$$

$$\text{(b)} \quad S \sqsubseteq_2^{\theta} T \;\Leftrightarrow\; \forall O \in \theta.\, S \underline{\text{must}}\ O \Rightarrow T \underline{\text{must}}\ O$$

$$\text{(c)} \quad S \sqsubseteq_1^{\theta} T \;\Leftrightarrow\; S \sqsubseteq_2^{\theta} T \;\&\; S \sqsubseteq_3^{\theta} T.$$

Putting Θ for the set of all observers, as defined by the above grammar, we have

$$S \sqsubseteq_i T \;\Leftrightarrow\; S \sqsubseteq_i^{\Theta} T, \quad \text{for } i = 1, 2, 3.$$

Several interesting sets of observers are natural candidates for special consideration. We will use $\mathcal{D}$ for the set of *deterministic* observers, in which no τ labels appear and the arcs at each node have distinct labels; $\mathcal{S}$ will denote the class of *sequential* observers, deterministic trees with a unique branch; and $\mathcal{K}$ will denote the set of all deterministic observers in which the ω event only appears alone, in that if a node has an ω arc then it has no other arcs. This condition is equivalent to the *unavoidability* or *exclusiveness* of ω. This last set of observers regards termination as unambiguously possible or impossible: there is never any other action which could by-pass a possibility of success. These classes of test will figure prominently in the results which follow.

In investigating the relationship between these various preorders, it will be convenient to distinguish some special tests. We therefore introduce some notation.

Notation: For any sequence $s \in \Sigma^*$, the test $\mathbf{do}(s)$ is $\overline{s}\omega\mathrm{NIL}$:

$$\mathbf{do}(\langle\,\rangle) = \omega\mathrm{NIL}$$
$$\mathbf{do}(at) = \overline{a}\mathbf{do}(t)$$

Notice that the $\mathbf{do}(s)$ tests are precisely the sequential tests.

For any sequence $s \in \Sigma^*$ and any finite set X of events, the test $\mathbf{not}(s, X)$ is given by the rules:

$$\mathbf{not}(\langle\,\rangle, X) = \sum_{x \in X} \overline{x}\omega\mathrm{NIL}$$
$$\mathbf{not}(at, X) = \omega\mathrm{NIL} + \overline{a}\mathbf{not}(t, X).$$

In particular, the test $\mathbf{not}(\langle\,\rangle, \emptyset)$ is simply NIL. And the test $\mathbf{not}(a, \{\,b\,\})$ is $\omega\mathrm{NIL} + \overline{a}\overline{b}\omega\mathrm{NIL}$. Notice that these tests are deterministic but do not belong to $\mathcal{K}$.

The importance of these observers is that the $\mathbf{do}(s)$ tests are sufficient to investigate the possible traces of a process, while the $\mathbf{not}(s, X)$ tests will provide information about the failures of a process.

Lemma 6.2.4: For all processes S and tests O,

$$S \underline{\text{ may }} O \;\Leftrightarrow\; \exists s.\, s \in \text{traces}(S) \;\&\; s\omega \in \text{traces}(O).$$

Proof. By definition,

$$S \underline{\text{ may }} O \Leftrightarrow \exists U.\, (S|O) \stackrel{\tau}{\Longrightarrow} U \;\&\; (U \stackrel{\omega}{\longrightarrow}).$$

But, by definition of Milner's composition,

$$(S|O) \stackrel{\langle\,\rangle}{\Longrightarrow} U \;\Leftrightarrow\; S \stackrel{s}{\Longrightarrow} S' \;\&\; O \stackrel{\overline{s}}{\Longrightarrow} O' \;\&\; U = S'|O'$$

for some s, S', O'. Since ω is not in the alphabet of any process, only a *test* can perform it:

$$(S'|O') \stackrel{\omega}{\longrightarrow} \;\Leftrightarrow\; O' \stackrel{\omega}{\longrightarrow}.$$

Hence, since $S \stackrel{s}{\Longrightarrow} S'$ iff $s \in \text{traces}(S)$, and $O \stackrel{\overline{s}}{\Longrightarrow} O' \stackrel{\omega}{\longrightarrow}$ iff $\overline{s}\omega \in \text{traces}(O)$, we have:

$$S \underline{\text{ may }} O \Leftrightarrow \exists s.s \in \text{traces}(S) \;\&\; \overline{s}\omega \in \text{traces}(O)$$

That completes the proof. ∎

Lemma 6.2.5: For all processes S and all $s \in \Sigma^*$,

$$S \underline{\text{ may }} \mathbf{do}(s) \;\Leftrightarrow\; s \in \text{traces}(S).$$

Proof. Use Lemma 6.2.4. ∎

Corollary:

$$S \sqsubseteq_3 T \quad \Leftrightarrow \quad \text{traces}(S) \subseteq \text{traces}(T).$$

Proof. By Lemmas 6.2.4. and 6.2.5. ∎

Hennessy and de Nicola asked the question: what is the smallest set of observers Θ which generates each $\sqsubseteq_i$? This result shows that the observers $\mathbf{do}(s)$ suffice for $\sqsubseteq_3$. Thus the preorder $\sqsubseteq_3$ generated by the sequential tests $\mathcal{S}$ coincides with the preorder generated by all tests in Θ.

Lemma 6.2.6: For all S and T

$$S \sqsubseteq_3^{\mathcal{S}} T \quad \Leftrightarrow \quad S \sqsubseteq_3 T.$$

Now let us consider the tests of the form $\mathbf{not}(s, X)$. We will see that the test $\mathbf{not}(s, X)$ investigates whether or not a process can perform the sequence s and then be unable to perform any of the events in X; in other words, whether or not (s, X) is a possible failure of S. To give some idea of what this type of test characterises, consider the following example:

Example . The test $\mathbf{not}(a, \{b\})$ is $\omega\text{NIL} + \overline{a}\overline{b}\omega\text{NIL}$. Any tree whose initials do not contain a must pass this test, since the only possible computation is the trivial one. However, if a is initially possible, then the process must pass the test only if every a-action leads to a place where a b-action is then possible. For example, the process $ab\text{NIL}$ must pass the test, whereas $a\text{NIL}$ has an unsuccessful computation and fails the test.

Lemma 6.2.7: For all S, s, X

$$S \underline{\text{must}}\ \mathbf{not}(s, X) \quad \Leftrightarrow \quad (s, X) \notin \text{failures}(S).$$

Proof. By induction on length(s).

Base case. We want to prove that S $\underline{\text{must}}$ $\mathbf{not}(\langle\,\rangle, X)$ iff $(\langle\,\rangle, X)$ is not a failure of S. If X is empty, we have $\mathbf{not}(\langle\,\rangle, \emptyset) = \text{NIL}$. But no process has a successful computation with a test that cannot succeed. And NIL is such a test, since NIL cannot perform an ω-action. Thus S $\not{\underline{\text{must}}}$ NIL; since the pair $(\langle\,\rangle, \emptyset)$ is always a failure of S, the result holds.

Otherwise, when X is non-empty,

$$\mathbf{not}(\langle\,\rangle, X) = \sum_{x \in X} \overline{x}\omega\text{NIL}.$$

It is easy to check that

$$S \underline{\text{must}} \sum_{x \in X} \overline{x}\omega\text{NIL} \quad \Leftrightarrow \quad \forall T.\, S \stackrel{\tau}{\Longrightarrow} T \Rightarrow \text{initials}(T) \cap X \neq \emptyset.$$

This is because the composition $S|\sum_{x\in X}\overline{x}\omega\mathrm{NIL}$ cannot make a transition to a term which can succeed unless the S side is able to make an x-action for some $x \in X$. But this condition is precisely that $(\langle\rangle, X) \notin \mathrm{failures}(S)$. Again the result holds.

Inductive step Assume, for all T, X

$$T \underline{\text{must}}\ \mathbf{not}(t,X) \quad \Leftrightarrow \quad (t,X) \notin \mathrm{failures}(T).$$

We must show that, for all S,

$$S \underline{\text{must}}\ \mathbf{not}(at,X) \quad \Leftrightarrow \quad (at,X) \notin \mathrm{failures}(S).$$

But, by definition,

$$\mathbf{not}(at,X) = \omega\mathrm{NIL} + \overline{a}\mathbf{not}(t,X).$$

It is easy to see that

$S \underline{\text{must}}\ \mathbf{not}(at,X) \Leftrightarrow$ either $a \notin \mathrm{initials}(S)$

or $a \in \mathrm{initials}(S)\ \&\ \forall T.\ S \stackrel{a}{\Longrightarrow} T \ \Rightarrow\ T \underline{\text{must}}\ \mathbf{not}(t,X)$

and hence that

$$S \underline{\text{must}}\ \mathbf{not}(at,X) \Leftrightarrow \forall T.\ S \stackrel{a}{\Longrightarrow} T \ \Rightarrow\ T \underline{\text{must}}\ \mathbf{not}(t,X).$$

By the inductive hypothesis this gives

$$S \underline{\text{must}}\ \mathbf{not}(at,X) \Leftrightarrow \forall T.\ S \stackrel{a}{\Longrightarrow} T \ \Rightarrow\ (t,X) \notin \mathrm{failures}(T).$$

If (at, X) were a possible failure of S, there would be a U such that

$$S \stackrel{at}{\Longrightarrow} U\ \&\ \mathrm{initials}(U) \cap X = \emptyset.$$

But then there would also be a T such that

$$S \stackrel{a}{\Longrightarrow} T \stackrel{t}{\Longrightarrow} U\ \&\ (t,X) \in \mathrm{failures}(T).$$

This contradicts our assumption. Hence, (at, X) cannot be a failure of S. That completes the proof. ∎

Recall that the failures preorder on synchronisation trees was defined by

$$S \sqsubseteq T \ \Leftrightarrow\ \mathrm{failures}(S) \supseteq \mathrm{failures}(T).$$

We are now able to prove a connection between this preorder and the *must* preorder.

Lemma 6.2.8: For all S and T,

$$S \sqsubseteq T \quad \Rightarrow \quad S \sqsubseteq_2 T.$$

Proof:

Let $S \sqsubseteq T$ and suppose $S \not\sqsubseteq_2 T$. Then

(1) $\mathrm{failures}(S) \supseteq \mathrm{failures}(T)$,

(2) $S \underline{\text{must}}\ O$

(3) $\neg T \underline{\text{must}}\ O$

for some test O. Since T is divergence-free and O is finite, the reason for (3) must be some pair of derivations

$$T \stackrel{s}{\Longrightarrow} T' \quad \text{of } T,$$
$$O \stackrel{\overline{s}}{\Longrightarrow} O' \quad \text{of } O$$

such that $(T'|O') \not\xrightarrow{\tau}$ and $(T'|O') \not\xrightarrow{\omega}$. Let $X = \text{initials}(O')$. Then we have

$$T \stackrel{s}{\Longrightarrow} T' \ \& \ \text{initials}(T') \cap X = \emptyset,$$

because otherwise $T'|O'$ would have a τ-transition. Thus, (s, X) is a failure of T; but by (1) this gives (s, X) as a failure of S. Therefore, there must be an S' such that

$$S \stackrel{s}{\Longrightarrow} S' \ \& \ \text{initials}(S') \cap X = \emptyset.$$

Since S is divergence-free, we can assume without loss of generality that S' has no τ-transitions: replacing S' by a τ-derivative does not affect its ability to refuse X, and there are only finitely many τ-derivatives of S'. Then we have

$$(S|O) \stackrel{\tau}{\Longrightarrow} (S'|O') \not\xrightarrow{\tau}.$$

Since O' has no ω-transition, this is a failing computation of $S|O$, in contradiction to (2). Supposing that $S \sqsubseteq_2 T$ was false has produced a contradiction, so it must be the case that $S \sqsubseteq_2 T$. That completes the proof. ∎

Lemma 6.2.9: For all S and T,

$$S \sqsubseteq_2 T \quad \Rightarrow S \sqsubseteq T.$$

Proof. Suppose $S \sqsubseteq_2 T$. Then for all (s, X) if S must satisfy the test $\mathbf{not}(s, X)$ so must T. Thus, by Lemma 6.2.7, for all (s, X) we have

$$(s, X) \notin \text{failures}(S) \ \Rightarrow \ (s, X) \notin \text{failures}(T).$$

Hence, $\text{failures}(S) \supseteq \text{failures}(T)$, or equivalently $S \sqsubseteq T$ as required. ∎

Corollary: For all S and T,

$$S \sqsubseteq T \quad \Leftrightarrow \quad S \sqsubseteq_2 T.$$

There is, therefore, an extremely close relationship between failures and *necessary* properties of trees. As long as we are allowed to apply tests of the form $\mathbf{not}(s, X)$, the *must* preorder $\sqsubseteq_2$ coincides with the failures order.

It is also clear from these results that the observers of the form $\mathbf{not}(s, X)$ are sufficient to generate the preorder $\sqsubseteq_2$. Since they all belong to the set $\mathcal{D}$ of deterministic tests, we also see that deterministic tests are enough to generate this preorder. Let $\mathcal{N}$ be the set of all tests $\mathbf{not}(s, X)$. Then

Corollary: For all S and T,

$$S \sqsubseteq_2^{\mathcal{N}} T \quad \Leftrightarrow \quad S \sqsubseteq_2^{\mathcal{D}} T \quad \Leftrightarrow \quad S \sqsubseteq_2 T.$$

Since the sequential observers are also deterministic tests, and the sequential tests suffice to generate the *may* preorder, we have seen that the subsets $\mathcal{S}$ and $\mathcal{N}$ of $\mathcal{D}$ generate $\sqsubseteq_3$ and $\sqsubseteq_2$ respectively. The deterministic observers therefore generate all three of the preorders:

$$S \sqsubseteq_i^{\mathcal{D}} T \Leftrightarrow S \sqsubseteq_i T, \quad i = 1, 2, 3.$$

Using our characterisations of $\sqsubseteq_2$ and $\sqsubseteq_3$, we can give a characterisation of $\sqsubseteq_1$ in terms of failures and traces. Since this relation is simply the intersection of the other two, we have:

Corollary: For all S and T,

$$S \sqsubseteq_1 T \quad \Leftrightarrow \quad \text{failures}(S) \supseteq \text{failures}(T) \ \& \ \text{traces}(S) = \text{traces}(T).$$

Hennessy and de Nicola are also concerned with the *equivalences* generated by their preorders. They define equivalences $\equiv_i$ and $\equiv_i^{\theta}$ by:

Definition 6.2.10: For $i = 1, 2, 3$ and θ any set of tests,

$$S \equiv_i T \quad \Leftrightarrow \quad S \sqsubseteq_i T \ \& \ T \sqsubseteq_i S,$$
$$S \equiv_i^{\theta} T \quad \Leftrightarrow \quad S \sqsubseteq_i^{\theta} T \ \& \ T \sqsubseteq_i^{\theta} S.$$

Our results show that, on divergence-free terms, using deterministic tests at least, $\equiv_3$ is simply *trace-equivalence* and $\equiv_2$ is *failure-equivalence.* Moreover, in these circumstances $\equiv_1$ is identical to $\equiv_2$.

Corollary: For all S and T,

$$S \equiv_3 T \quad \Leftrightarrow \quad \text{traces}(S) = \text{traces}(T)$$
$$S \equiv_2 T \quad \Leftrightarrow \quad \text{failures}(S) = \text{failures}(T)$$
$$S \equiv_1 T \quad \Leftrightarrow \quad \text{failures}(S) = \text{failures}(T).$$

We end this section by remarking that our results here show that the three preorders are generated by simple classes of observers. Indeed, all of the observers used in proving the properties of processes have been *deterministic,* in that at each node in a test tree the arcs possess distinct, visible labels; no test ever has a τ-action or distinct possible derivatives on any particular action. Although we did not state this initially, we could have restricted attention to deterministic tests throughout, without changing the final outcome: the equivalence relations and preorders generated by the class of deterministic tests coincide with the standard ones. Note, however, the role played by ω in our tests: we allowed tests such as

$$\mathbf{not}(a, \{b\}) = \omega\text{NIL} + \overline{a}\overline{b}\omega\text{NIL},$$

in which ω appears at a node with other possible actions. We will see that this type of test is crucial if we are to be able to investigate failures of processes. If we insist that termination, whenever it is a *possible* action, is always the only possibility, the $\mathbf{not}(s, X)$ tests are no longer available. This restricted class $\mathcal{K}$ of tests was used by Kennaway, and we will investigate this later.

Alternative forms of testing.

We mentioned earlier that there are alternative plausible methods for treating tests. In particular, Hennessy and de Nicola discuss another formulation of *must*. In the version described above, a process *must* pass a test if every computation arising from the test terminates in a successful state; when the computation can progress no further, the remainder of the test must be able to signal success. An additional requirement, that the test applied to the process cannot succumb to divergence, is irrelevant here, as we are only considering convergent processes and finite tests.

In Hennessy and de Nicola's alternative formulation, a process *must* pass a test if every computation arising from the test must reach a successful state, and if in every such computation success occurs before any divergent state is reached. Note that this allows infinite, non-terminating computations provided the test reaches a successful state at some (finite) stage during the computation. The essential difference between this and the earlier formulation is that here a computation may proceed past a successful state without invalidating the success of the test.

Notice that, since we are using only finite tests, any divergence arises because of the process alone. The formal definition, reflecting these ideas, is as follows:

Definition 6.2.11: For all processes S and tests O,

$$S \underline{\text{must}}' O \Leftrightarrow \text{for all computations } (S|O) = S_0|O_0) \xrightarrow{\tau} (S_1|O_1) \xrightarrow{\tau} \cdots$$

$$(i) \quad \exists k. O_k \xrightarrow{\omega},$$

$$(ii) \quad S_n\uparrow \Rightarrow O_k \xrightarrow{\omega}, \text{ for some } k \leq n.$$

In the absence of divergence, this says that every computation must pass through or reach a successful state, although we do not insist that the *final* state is successful. It is obvious from the definition that a *must* test is at least as strong as a *must'* test: for all S and O,

$$S \underline{\text{must}} O \quad \Rightarrow \quad S \underline{\text{must}}' O.$$

The reverse implication does not, in general, hold. If tests can perform τ actions, the new formulation is not the same as the old: for example, we have

for all S,

$$S \underline{\text{must}}' \omega\text{NIL} + \tau\text{NIL}$$
$$\neg S \underline{\text{must}} \omega\text{NIL} + \tau\text{NIL}.$$

Even if we only consider *deterministic* tests, the results of testing will differ: for all $a \in \Sigma$, all non-empty X, and all processes S, we have

$$S \underline{\text{must}}' \omega\text{NIL} + aO$$
$$\text{for all } O.$$

It follows that the tests $\mathbf{not}(s, X)$ yield no useful results about failures when we use them this way. Moreover, a simple argument shows that any deterministic test O is equivalent as far as $\underline{\text{must}}'$ is concerned to the test O' formed by pruning away all other branches from every node of O at which a successful action appears, leaving just the ω arc there. This operation converts any deterministic test into a test in $\mathcal{K}$. Every (non-trivial) test $\mathbf{not}(s, X)$ is identified in this way with ωNIL. In other words, it makes no difference if we use the full set of deterministic tests or restrict to the cases when ω always excludes other actions. The *must* preorder corresponding to the new formulation of *must'* is therefore generated by $\mathcal{K}$. Let us give this version of the ordering a name.

Definition 6.2.12: For processes S and T,

$$S \sqsubseteq' T \Leftrightarrow \forall O.\,(S \underline{\text{must}}' O \Rightarrow T \underline{\text{must}}' O.)$$

We have indicated that the *must'* order is generated by the class $\mathcal{K}$ of tests. Indeed, the *must'* order and the *must* order agree when restricted to this class of observer.

Theorem 6.2.13: For all processes S and T,

$$S \sqsubseteq' T \Leftrightarrow S \sqsubseteq_2^{\mathcal{K}} T.$$

This class $\mathcal{K}$ of tests is suggested by the work of Kennaway, which we will now outline.

3. Kennaway's tests

In Kennaway's general model of nondeterminism [K2] we begin with a set M of *(deterministic) machines,* with typical member m, and a set T of *tests,* ranged over by t. We assume a predefined relation $\underline{\text{sat}}$ on machines and tests, specifying which machines pass which tests. A *nondeterministic machine* is

a certain kind of set of deterministic machines; we use M to range over sets of deterministic machines. The idea is that a nondeterministic machine can behave like any member of a set of deterministic machines, but the choice of which one cannot be influenced by the machine's environment; nor is it known in advance. If we are to be sure that a nondeterministic machine passes a test, then it must be the case that all possible choices of deterministic machine will satisfy the test. Thus, the satisfaction relation extends to sets of machines in the obvious way:

$$M \underline{\text{sat}}\ t \quad \Leftrightarrow \quad \forall m \in M.\, m \underline{\text{sat}}\ t.$$

Two sets M and M' are indistinguishable if they pass exactly the same set of tests. We may define an equivalence relation on nondeterministic machines accordingly:

$$M \equiv_K M' \quad \Leftrightarrow \quad \forall t \in \mathbf{T}.\, (M \underline{\text{sat}}\ t \Leftrightarrow M' \underline{\text{sat}}\ t).$$

Clearly there is a largest nondeterministic machine equivalent to any M : it is simply the set of all deterministic machines which satisfy precisely the same tests as M does. This distinguished member of the equivalence class of M will be denoted $\mathbf{c}(M)$; its definition is

$$\mathbf{c}(M) = \{\, m \mid \forall t.\, m \underline{\text{sat}}\ t \Leftrightarrow M \underline{\text{sat}}\ t \,\}.$$

Of course, it is always the case that

$$M \subseteq \mathbf{c}(M)$$
$$\mathbf{c}(\mathbf{c}(M)) = \mathbf{c}(M).$$

We will refer to $\mathbf{c}(M)$ as the *closure* of M. Kennaway chooses these *closed* sets for his nondeterministic machines. The superset order on nondeterministic machines clearly corresponds to the notion of "being more nondeterministic," and

$$M \supseteq M'$$

will be the case if and only if M' passes every test that M passes.

Kennaway applies his ideas to nondeterministic communicating processes in [K1]. For a deterministic machine he takes a tree over Σ, that is, a deterministic synchronisation tree. A test is again a finite deterministic synchronisation tree, but with the leaves partitioned into two disjoint sets: the *successful* leaves and the *unsuccessful* leaves. We can again represent this by using a special label ω to indicate success, if we allow ω labels only on terminal arcs and insist that such arcs, whenever they appear at a node, do so alone (so that success is *unambiguous* whenever it is possible), then we

can clearly translate a tree with partitioned leaves into a tree with an ω arc replacing every successful leaf. Thus we are not losing any generality by doing this. This class of observer was called $\mathcal{K}$ earlier. We will continue to use O to range over this set of observers.

Kennaway defines a notion of *safeness* on trees. A tree is *safe* if and only if every terminal arc is an ω-arc. This clearly captures the idea that a tree can only be guaranteed to succeed eventually if every path from its root leads to a successful action. We write safe(T) to indicate that T is safe. Since Kennaway regards a test and a process as executing in parallel, each action requiring cooperation by both process and test, a test applied to a process *must* succeed if and only if this parallel composition is safe. Thus we define, following Kennaway,

$$S \underline{\text{sat}}\ O \quad \Leftrightarrow \quad \text{safe}(S \,||\, O),$$

where the parallel composition $S \,||\, O$ of synchronisation trees is as defined earlier.

We can rephrase this definition in more familiar terms. Since there are no τ actions here, the above definition of satisfaction is equivalent to the following:

$$S \underline{\text{sat}}\ O \quad \Leftrightarrow \quad \forall s \in \Sigma^*.\, (S \,||\, O) \stackrel{s}{\Longrightarrow} (S' \,||\, O') = \text{NIL} \;\Rightarrow\; O' \stackrel{\omega}{\longrightarrow}.$$

For deterministic synchronisation trees this is precisely Kennaway's notion of satisfaction; we have given a version also applicable to general synchronisation trees.

For tests in $\mathcal{K}$, this satisfaction relation coincides with our two versions of *must*. The reason for this is simple. For any test O let $\overline{O}$ be the test obtained by replacing every visible label in O by its complement (leaving τ and ω arcs unchanged). Every computation of $S \,||\, O$ corresponds to a computation of $S|\overline{O}$, and vice versa. The following result makes this precise.

Lemma 6.3.1: For all S and for all $O \in \mathcal{K}$,

$$\exists a.\, (S \,||\, O) \stackrel{a}{\longrightarrow} (S' \,||\, O') \quad \Leftrightarrow \quad (S|\overline{O}) \stackrel{\tau}{\longrightarrow} (S'|\overline{O'}).$$

Proof. By definition,

$$\begin{aligned}(S \,||\, O) \stackrel{a}{\longrightarrow} (S' \,||\, O') \;\Leftrightarrow\; &\text{(i) } a \neq \tau,\ S \stackrel{a}{\longrightarrow} S',\ O \stackrel{a}{\longrightarrow} O' \\ &\text{or (ii) } a = \tau,\ S \stackrel{\tau}{\longrightarrow} S',\ O' = O \\ &\text{or (iii) } a = \tau,\ S' = S,\ O \stackrel{\tau}{\longrightarrow} O'.\end{aligned}$$

But these are precisely the conditions which guarantee a τ action of $S|\overline{O}$ to $S'|\overline{O'}$. ∎

Corollary: For all S and for all $O \in \mathcal{K}$, $S \,||\, O = \text{NIL}$ iff $S|\overline{O}$ has no τ actions.

Corollary: Every computation of $S \,||\, O$ corresponds to a computation of $S|\overline{O}$, and vice versa:

$$\exists s \in \Sigma^*.(S \,||\, O) \stackrel{s}{\Longrightarrow} (S' \,||\, O') \quad \Leftrightarrow \quad (S|\overline{O}) \stackrel{\langle\rangle}{\Longrightarrow} (S'|\overline{O'}).$$

It is also evident from these results that the successful computations also correspond in the two models. Hence we see that, for tests $O \in \mathcal{K}$

$$\exists s.(S \,||\, O) \stackrel{s}{\Longrightarrow} (S' \,||\, O') = \text{NIL} \;\&\; O' \stackrel{\omega}{\longrightarrow} \;\Leftrightarrow\; (S|\overline{O}) \stackrel{\langle\rangle}{\Longrightarrow} (S'|\overline{O'}) \not\stackrel{\tau}{\longrightarrow} \;\&\; \overline{O'} \stackrel{\omega}{\longrightarrow}.$$

In other words, Kennaway's satisfaction relation is identical with the *must* relation, when restricted to tests in $\mathcal{K}$.

Lemma 6.3.2: For all synchronisation trees S and all tests $O \in \mathcal{K}$

$$S \;\underline{\text{sat}}\; \overline{O} \;\Leftrightarrow\; S \;\underline{\text{must}}\; O.$$

∎

The only way for a process S to *fail* a test O is by executing a sequence of actions s which does not take O to a successful node, and then being able to *refuse* all of the events in which the test is then capable of participating.

Examples.

1. The test ωNIL is passed by every deterministic machine, because every machine can at least do nothing. If the test is prepared to ask no questions about the process's behaviour, it is trivial to satisfy.

2. The process NIL passes no non-trivial test. Any test other than ωNIL requires at least one visible action to take place before success, and NIL is incapable of any visible action. Put another way, a deadlocked process passes no non-trivial test.

3. The process $a\text{NIL} + b\text{NIL}$ passes the tests $\overline{a}\omega\text{NIL}$ and $\overline{b}\omega\text{NIL}$, as well as $\overline{a}\omega\text{NIL} + \overline{b}\omega\text{NIL}$. Of these tests, the process aNIL only satisfies the first and third, while the process bNIL passes the second and third.

4. The satisfaction relation extends to sets of deterministic processes. A nondeterministic machine M is a set of deterministic machines, and M $\underline{\text{sat}}$ t if and only if every member of M satisfies t. Using the above examples to

guide us, we see that every nondeterministic machine passes the trivial test, and every M which contains NIL passes no other test.

5. Let M and M' be the following machines:

$$M = \{\, a\text{NIL} + b\text{NIL} \,\},$$
$$M' = \{\, a\text{NIL},\ b\text{NIL} \,\}.$$

Then M satisfies precisely the same tests as the deterministic process $a\text{NIL} + b\text{NIL}$, as above. And M' satisfies a test if and only if both $a\text{NIL}$ and $b\text{NIL}$ pass it. Hence, we have

$$M \ \underline{\text{sat}}\ \overline{a}\omega\text{NIL} + \overline{b}\omega\text{NIL},\ \overline{a}\omega\text{NIL},\ \overline{b}\omega\text{NIL}$$
$$M' \ \underline{\text{sat}}\ \overline{a}\omega\text{NIL} + \overline{b}\omega\text{NIL}.$$

In Kennaway's model for communicating machines, we identify every set M of deterministic machines with its closure $\mathrm{c}(M)$, defined to be the largest set of machines which passes exactly the same tests as M does. Since deterministic machines are trees over Σ, and as such are uniquely determined as non-empty, prefix closed sets of traces, there is a connection, as noted earlier, between the sets of trees used here for modelling processes and the *implementation sets* of Chapter 3. Recall that we defined, for a CSP process P, a set of trees $\mathrm{imp}(P)$, its *implementation set:*

$$\mathrm{imp}(P) = \{\, T \mid P \sqsubseteq \mathrm{det}(T) \,\}.$$

For a finite alphabet Σ, the sets occurring as implementation sets of processes turned out to be precisely the *convex-closed* sets. Indeed, if the alphabet is finite, Kennaway's closed sets are exactly the convex sets. Thus two independently motivated models of communicating processes have arrived at the same basic model of processes. However, Kennaway's view of nondeterministic processes does not coincide entirely with ours. For him, any process which *may* deadlock (but *may* also have other actions) is equivalent to any other process which can deadlock; all such processes are represented by sets of trees M containing NIL, and we have seen that these sets pass precisely the same tests (only the trivial test).

Kennaway's model of nondeterministic processes can be obtained from ours by applying a *filtering* operation K, which identifies two processes iff they have the same behaviour up to deadlock: K is a function on failures sets, defined by

$$K(P) = \{\, (s,X) \mid (s,X) \in P \,\} \cup \{\, (st,X) \mid (s,\Sigma) \in P \,\}.$$

Thus, for example, $K(\text{STOP}) = K(\text{CHAOS})$. For any of our CSP processes P, the nondeterministic machine corresponding to it will be the closure of the set $\text{imp}(P)$. But since Kennaway's model makes more identifications, this will also be the closure of the set $\text{imp}(K(P))$. The partial order on our domain of processes is the superset order; this is also the case for Kennaway's model. Thus, Kennaway's nondeterministic machines form a space *isomorphic* to the subspace of CSP processes formed by applying the K operation. If we define KPROC to be the range of K, again ordered by superset, then the mappings:

$$\text{imp} : \text{KPROC} \to \mathbf{M}$$
$$\text{proc} : \mathbf{M} \to \text{KPROC}$$

are an isomorphism pair.

Although Kennaway was not concerned to extend his satisfaction relation to arbitrary synchronisation trees, as his formulation employed only deterministic trees and sets of them, it is quite easy to do so. In the definition given earlier of $S \underline{\text{sat}}\ t$ we were only considering deterministic processes. The version we gave was also applicable to arbitrary synchronisation trees: we need merely to take τ actions into consideration. This is a perfectly reasonable extension of **sat**. It is not quite in the spirit of Kennaway's work, however, because it does not explicitly define satisfaction for nondeterministic processes in terms of satisfaction for deterministic processes. We already have a method of associating a set of deterministic trees with an arbitrary synchronisation tree: map the synchronisation tree to a failure set and take the implementation set. This allows us to define an implementation set for a synchronisation tree T, by the definition:

$$\text{imp}(T) = \text{imp}(\text{failures}(T)).$$

If our version of **sat** for synchronisation trees is to agree with Kennaway's, then the following should hold:

$$S \underline{\text{sat}}\ t \Leftrightarrow \forall U \in \text{imp}(S).\ U \underline{\text{sat}}\ t.$$

This is indeed true, because of the following fact about implementation sets:

$$\text{failures}(S) = \bigcup \{ \text{failures}(T) \mid T \in \text{imp}(S) \}.$$

Theorem 6.3.3 : For all S and all $O \in K$,

$$S \underline{\text{sat}}\ O \ \Leftrightarrow\ \forall T \in \text{imp}(S).\ T \underline{\text{sat}}\ O.$$

Proof. Define $\mathrm{acc}(O) = \{\,(s,X) \mid O \xrightarrow{\bar{s}} O' \;\&\; \mathrm{initials}(O') = X \;\&\; O' \not\xrightarrow{\omega}.\,\}$ Clearly, by definition of sat there is only one reason why a process S might not pass a test O : if some set (s,X) in $\mathrm{acc}(O)$ is also a failure of S. But, by the above property of $\mathrm{imp}(S)$, this happens iff there is a tree T in $\mathrm{imp}(S)$ such that (s,X) is also a failure of T. Since this condition means that T also fails to satisfy O, the result follows. ∎

We can define an ordering on trees which reflects Kennaway's notion of passing tests:

Definition 6.3.4: For all processes S and T,

$$S \sqsubseteq_K T \;\Leftrightarrow\; \forall O \in K.\, S \;\underline{\mathrm{sat}}\; O \Rightarrow T \;\underline{\mathrm{sat}}\; O.$$

Putting together the result of Lemma 6.3.2 and this definition, we see that Kennaway's order $\sqsubseteq_K$ on synchronisation trees satisfies:

$$S \sqsubseteq_K T \;\Leftrightarrow\; K(\mathrm{failures}(S)) \supseteq K(\mathrm{failures}(T)),$$

and that it coincides with the order $\sqsubseteq_2^K$ and with $\underline{\mathrm{must}}'$.

A more concrete method of proving this result is suggested by trying to find a set of observers in K which test for failures, in a similar way to the properties of the $\mathbf{not}(s,X)$ tests. We have seen that these particular tests are useless here. However, we can define a variant, which we call $\mathbf{acc}(s,X)$. For simplicity, assume the alphabet is finite. Since every term in our language only uses finitely many labels, this is not restrictive. Define the tests $\mathbf{acc}(s,X)$ by induction on the length of s :

$$\mathbf{acc}(\langle\,\rangle, X) = \sum_{x \in X} \bar{x}\omega\mathrm{NIL}$$
$$\mathbf{acc}(at, X) = \bar{a}\mathbf{acc}(t,X) + \sum_{b \neq a} \bar{b}\omega\mathrm{NIL}.$$

For example, if the alphabet Σ is just $\{\,a, \bar{a}, b, \bar{b}\,\}$, then

$$\mathbf{acc}(\langle\,\rangle, \{\,a\,\}) = \bar{a}\omega\mathrm{NIL},$$
$$\mathbf{acc}(a, \{\,a\,\}) = \bar{a}\bar{a}\omega\mathrm{NIL} + \bar{b}\omega\mathrm{NIL}$$

and we can see that

$$a\mathrm{NIL} \;\underline{\mathrm{sat}}\; \mathbf{acc}(\langle\,\rangle, \{\,a\,\})$$

but aNIL does not satisfy the test $\mathbf{acc}(a, \{\,a\,\})$. These results are deducible from the following general property of these tests, which we call *acceptance* tests. Notice that, by construction, these tests belong to K.

Theorem 6.3.5: For all s, X, and all processes S,

$$S \underline{\text{sat}}\ \mathbf{acc}(s, X) \Leftrightarrow (s, X) \notin K(\text{failures}(S)).$$

Proof. By induction on the length of s.

Base case. If X is empty there is nothing to prove, because the pair $(\langle\,\rangle, \emptyset)$ is always a failure of any tree and the test NIL is not satisfiable. When X is non-empty, $S \underline{\text{sat}} \sum_{x \in X} \overline{x}\omega\text{NIL}$ if and only if every T in $\text{imp}(S)$ has an initial event in X, and this happens if and only if NIL is not an implementation of S and X is not a refusal; thus,

$$S \underline{\text{sat}}\ \mathbf{acc}(\langle\,\rangle, X) \Leftrightarrow (\langle\,\rangle, X) \notin K(\text{failures}(S)),$$

as required.

Inductive step. Assume that for t we have

$$T \underline{\text{sat}}\ \mathbf{acc}(t, X) \Leftrightarrow (t, X) \notin K(\text{failures}(T)),$$

for all T and X. The test $\mathbf{acc}(at, X)$ is $\overline{a}\mathbf{acc}(t, X) + \sum_{b \neq a} \overline{b}\omega\text{NIL}$. A tree S satisfies this test if

$$\begin{array}{ll} \text{(i)} & S \stackrel{\langle\rangle}{\Longrightarrow} S' \Rightarrow \text{initials}(S') \neq \emptyset \\ \text{(ii)} & S \stackrel{a}{\Longrightarrow} T \Rightarrow T \underline{\text{sat}}\ \mathbf{acc}(t, X). \end{array}$$

By the inductive hypothesis, these conditions can be rewritten:

$$\begin{array}{ll} \text{(i)} & S \stackrel{\langle\rangle}{\Longrightarrow} S' \Rightarrow \text{initials}(S') \neq \emptyset \\ \text{(ii)} & S \stackrel{a}{\Longrightarrow} T \Rightarrow (t, X) \notin K(\text{failures}(T)). \end{array}$$

But by definition of K, we see that the pair (at, X) is a K-failure of S if and only if either $(\langle\,\rangle, \Sigma)$ is a failure of S or there is a tree T such that

$$S \stackrel{a}{\Longrightarrow} T \ \&\ (t, X) \in K(\text{failures}(T)).$$

The first possibility arises if and only if there is an S' such that

$$S \stackrel{\langle\rangle}{\Longrightarrow} S' \ \&\ \text{initials}(S') = \emptyset.$$

Thus we have shown that

$$S \underline{\text{sat}}\ \mathbf{acc}(at, X) \Leftrightarrow (at, X) \notin K(\text{failures}(S)).$$

∎

4. Treating divergence

Now let us examine what happens when we try to extend the orders of the previous section to trees which may have the ability to diverge. In Hennessy and de Nicola's work, divergence corresponds (more or less) to the presence of an improperly guarded recursion, by which we mean a recursive term $\mu x.S$ in which S has a free occurrence of x guarded only by some number (possibly zero) of τs. Their language of terms allows, for example, the term $\mu x.(a\mathrm{NIL}+x)$, which diverges because there is an unguarded occurrence of x in the body. Similarly, the term $\mu x.(a\mathrm{NIL}+\tau x)$ diverges, because the occurrence of x is guarded only by τ. The first of these trees "unrolls" to give an infinitely branching tree, all of whose branches are aNIL; the divergence is reflected in the fact that this tree has infinite branching. In the second case, the term unrolls to give an infinitely deep but finitely branching tree; divergence here corresponds to the presence of an infinite path of τ arcs, and agrees with our version of divergence. For us, divergence corresponds to infinite internal chatter, and we are identifying τ with internal activity.

In Milner's original conception, synchronisation trees were finitely branching. If we take the view that a synchronisation tree should only be finitely branching, we must rule out recursions where the bound variable appears completely unguarded; but this does not rule out τ as a guard. In this section, then, we will consider trees which may have branches of the form τ^∞. By restricting attention to finitely branching trees of this type, we find some interesting relationships between the treatment of divergence in [HN] and our methods. From now on, we refer to our notion of divergence as infinite chatter, the presence of an infinite path of τ arcs. By Konig's Lemma, a finite synchronisation tree has an infinite τ branch if and only if it has arbitrarily long finite τ branches.

Definition 6.4.1: The unary predicate $\uparrow$ on trees is:

$$S\uparrow \quad \Leftrightarrow \quad S\xrightarrow{\tau^\infty}.$$

If $S\uparrow$ we say S *diverges,* and otherwise we write $S\downarrow$ (S *converges*).

It is also useful to generalise the notion of (immediate) divergence. We write $S\uparrow s$, and say that S diverges after s, if there is a prefix t of s and a tree T such that $S\stackrel{t}{\Longrightarrow}T$ & $T\uparrow$. The converse of this is written $S\downarrow s$. Read $S\uparrow s$ as saying that S may be diverging once it has performed s, and $S\downarrow s$ that S converges all the way to s. Notice that $S\uparrow\langle\rangle \Leftrightarrow S\uparrow$, and $S\downarrow\langle\rangle \Leftrightarrow S\downarrow$.

In order to extend the failure set semantics to cover divergent trees, we must introduce *divergence sets* as well; this treatment is consistent with

that of Chapter 5, where infinite chatter was failure-equivalent to CHAOS. There is an obvious connection between the divergence sets of trees and our earlier divergence set definitions for processes, since we can map processes to synchronisation trees as in Chapter 5. All the same, the treatment given here is self-contained.

Definition 6.4.2: For any tree S,

(i) $\text{failures}(S) = \{(s,X) \mid \exists T.\, S \stackrel{s}{\Longrightarrow} T \ \&\ \text{initials}(S) \cap X = \emptyset\} \cup \{(st,X) \mid S\uparrow s\}$

(ii) $\text{div}(S) = \{s \mid S\uparrow s\}$.

We still define the trace set of a tree to be:

$$\text{traces}(S) = \{s \mid (s,\emptyset) \in \text{failures}(S)\}.$$

In this section, we will write $S \sqsubseteq T$ to mean that the failures of S contain the failures of T and S can diverge whenever T can:

$$S \sqsubseteq T \quad \Leftrightarrow \quad \text{failures}(S) \supseteq \text{failures}(T) \ \&\ \text{div}(S) \supseteq \text{div}(T).$$

This is the natural extension of the failures order to divergent processes.

In the presence of divergence, the definitions of the previous section become:

Definition 6.4.3: For any term S and test O,

(i) $S \underline{\text{may}}\ O \quad \Leftrightarrow \quad \exists U.(S|O) \stackrel{\tau}{\Longrightarrow} U \ \&\ (U \stackrel{\omega}{\longrightarrow})$.

(ii) $S \underline{\text{must}}\ O \quad \Leftrightarrow \quad (S|O)\downarrow \ \&\ \forall U.\,(S|O) \stackrel{\tau}{\Longrightarrow} U \ \&\ (U \not\stackrel{\tau}{\longrightarrow}) \Rightarrow (U \stackrel{\omega}{\longrightarrow})$.

Notice that, with the new definition of failures(S) these definitions can again be reformulated in terms of failure sets, exactly as in Lemma 6.1.4.

In addition to the special classes of observers used earlier, $\mathbf{do}(s)$ and $\mathbf{not}(s,X)$, we will use a new set of tests designed to look for divergence. For any sequence s the observer $\mathbf{cnv}(s)$ is given by the rules:

$$\mathbf{cnv}(\langle\,\rangle) = \omega\text{NIL}$$
$$\mathbf{cnv}(at) = \omega\text{NIL} + \overline{a}\mathbf{cnv}(t).$$

For each s, $\mathbf{cnv}(s)$ is a test which is prepared to succeed after performing any prefix of $\overline{s}$. It fails if and only if a computation can diverge before completing a prefix of $\overline{s}$.

Attempting to find results analogous to those of the previous section, we proceed as follows. The proofs are straightforward modifications of earlier proofs.

Lemma 6.4.4: For all processes S and tests O,

$$S \underline{\text{may}}\ O \quad \Leftrightarrow \quad \text{either } \exists s.\ s \in \text{traces}(S)\ \&\ \overline{s}\omega \in \text{traces}(O)$$
$$\text{or } \exists \overline{s} \in \text{traces}(O).\ S\!\uparrow\! s.$$

Lemma 6.4.5: For all processes S and all $s \in \Sigma^*$,

$$S \underline{\text{may}}\ \mathbf{do}(s) \ \Leftrightarrow\ s \in \text{traces}(S).$$

Lemma 6.4.6: For all processes S and all $s \in \Sigma^*$,

$$S \underline{\text{must}}\ \mathbf{cnv}(s) \ \Leftrightarrow\ S\!\downarrow\! s.$$

Lemma 6.4.7: For all S and (s, X),

$$S \underline{\text{must}}\ \mathbf{not}(s, X) \ \Leftrightarrow\ (s, X) \notin \text{failures}(S)\ \&\ \forall x \in X.\ S\!\downarrow\! sx.$$

The importance of this result, which differs from the corresponding lemma on divergence-free trees, is that a test can only yield *positive* results when no divergence has occurred during the test. It is impossible to distinguish, in this framework, between a process which can *refuse* an event and a process which can diverge *after* first performing that event. For example, the trees S and T corresponding to the terms

$$a(\mu x.\tau x) + b(\mu x.\tau x)$$
$$a(\mu x.\tau x)$$

are

$$S = a\tau^\infty + b\tau^\infty$$
$$T = a\tau^\infty.$$

Clearly, T can refuse b but S cannot. However, any attempt to test for refusal of b, such as the test $\mathbf{not}(\langle\,\rangle, \{\,b\,\})$ fails to obtain an answer from S : indeed, $(S|b\omega\text{NIL})\!\uparrow$ and $S\!\uparrow\! b$. The tests O which S *must* satisfy are characterised by the conditions:

(i) $(\langle\,\rangle, \{\,\omega\,\}) \notin \text{failures}(O)$
(ii) $\overline{a} \notin \text{traces}(O)$
(iii) $\overline{b} \notin \text{traces}(O)$

Clearly T must satisfy any test having these properties, so we have $S \sqsubseteq_2 T$. But the failures of S and T are incomparable.

In the light of this example, it cannot be the case (as in Lemma 6.2.9) that $S \sqsubseteq_2 T$ always implies $S \sqsubseteq T$. Nevertheless, the converse implication still holds.

Lemma 6.4.8: For all trees S and T,

$$S \sqsubseteq T \quad \Rightarrow \quad S \sqsubseteq_2 T.$$

Proof. Suppose $S \sqsubseteq T$ and $S \not\sqsubseteq_2 T$. Then

(1) $\text{failures}(S) \supseteq \text{failures}(T)$
(2) $\text{div}(S) \supseteq \text{div}(T)$
(3) $S \underline{\text{must}}\ O$
(4) $\neg T \underline{\text{must}}\ O$

for some test O. If the reason for (4) is divergence, then there is a trace s of O such that $T\uparrow s$. But this would imply divergence of $S|O$, by (2). This contradicts (3). Hence, the reason for (4) must be a pair of derivations

$$T \stackrel{s}{\Longrightarrow} T'$$
$$O \stackrel{\bar{s}}{\Longrightarrow} O'$$

for which $T'|O'$ has no further τ-derivatives and has no ω-action. Put

$$X = \text{initials}(O').$$

Then (as in the proof of Lemma 6.2.8) the pair $(\dot{s}, X)$ is a failure of T. By (1), (s, X) is also a failure of S. Thus, there is a tree S' such that

$$S \stackrel{s}{\Longrightarrow} S' \ \&\ \text{initials}(S') \cap X = \emptyset.$$

The derivation $(S|O) \Longrightarrow (S'|O')$ is a failing computation, contradicting (3). That completes the proof. ∎

Now let us consider the *must'* relation, extended to divergent trees. Recall that S $\underline{\text{must}}'$ O iff every computation of this parallel composition passes through a successful state, and if the process reaches a divergent state during the computation the test must already have reached a successful state at some earlier stage.

Definition 6.4.9: For a process S and test O,

$$S \underline{\text{must}}'\ O \Leftrightarrow \text{whenever } (S|O) \stackrel{\tau}{\longrightarrow} (S_1|O_1) \stackrel{\tau}{\longrightarrow} \cdots \text{ is a computation}$$
$$\text{then } (i)\ \exists k. O_k \stackrel{\omega}{\longrightarrow}$$
$$(ii)\ S_n\uparrow \Rightarrow \exists k \leq n.\ O_k \stackrel{\omega}{\longrightarrow} \cdot$$

Under this definition, a divergent process passes just the trivial tests which are prepared to succeed without waiting for any response. For example, $\mu x.\tau x$ $\underline{\text{must}}'$ ωNIL.

The analogous extension of the sat relation to divergent trees is simply:

$$S \text{ sat } O \Leftrightarrow \text{whenever } (S \| O) \xrightarrow{c_1} (S_1 \| O_1) \xrightarrow{c_2} \cdots \text{ is a computation}$$
$$\text{then } (i)\ \exists k. O_k \xrightarrow{\omega} \ \&$$
$$(ii)\ S_n\uparrow \Rightarrow \exists k \leq n. O_k \xrightarrow{\omega}.$$

It is easy to check that again the tests in K suffice to generate the must′ ordering, and that the sat relation is identical with this version of *must*. Moreover, since the failures model identifies divergence with CHAOS, we still have

$$S \sqsubseteq_K T \Leftrightarrow K(\text{failures}(S)) \supseteq K(\text{failures}(T)).$$

Summary.

We have discussed various attempts to define the properties of processes which can reasonably be *observed* by testing. Several alternative notions of passing a test were covered. In each case we were able to make connections with the failures model of processes, and it appears that although the alternative testing ideas were originally motivated by different concerns, they have a common link with our approach. This is very interesting; it is important to discover and understand the inter-relationships between the various proposed models for concurrency in the literature. Here we have shown that there are some results to be obtained by classifying models using *assertions* such as *must* and *may*, and that different choices of the class of observers yield different models. This type of classification may be a step on the way to a more comprehensive grasp of the essential differences between the models for CCS and CSP discussed in this thesis, and may lead to future work encompassing yet other models for concurrency.

Chapter 7

Some examples

1. Dining Philosophers

One of the classic problems in the literature of concurrency is the problem of the five dining philosophers [Di1,H1]. Five independently active philosophers share access to a common dining table, in the centre of which is a bowl of food. Each philosopher has his own place at the table, and between each pair of adjacent places there is a fork; in order to partake of the food a philosopher must sit at his place and pick up both adjacent forks, since it is supposed that the food is sufficiently messy to require more than a single fork. Here is a situation in which there is a possibility of *deadlock,* since it is conceivable that all five philosophers grow hungry and assume their seats, and that each then picks up the fork to his left (say), whereupon there are no free forks and no-one can eat because none of the philosophers has two forks. This is a typical instance of the dangers of sharing resources amongst competing concurrent processes.

We can give an elegant formulation of the dining philosophers in our language of processes, by representing each philosopher and each fork as a process. As a template for a prototypical philosopher, we use the recursively defined process:

$$\begin{aligned} \text{PHIL} = (&\text{enter} \rightarrow \text{pickleft} \rightarrow \text{pickright} \rightarrow \text{eat} \rightarrow \\ &\text{putleft} \rightarrow \text{putright} \rightarrow \text{leave} \rightarrow \text{PHIL}). \end{aligned}$$

We have represented the actions of a philosopher as events: pickleft, for example, represents the act of picking up the fork to the left of the philosopher's place.

We make copies from this template for each philosopher, relabelling the events so as to distinguish between the different philosophers. For $i = 0 \ldots 4$

let f_i be the alphabet transformation which relabels events as follows, and otherwise leaves events unchanged:

$$\begin{aligned} f_i(\text{enter}) &= \text{i.enter} \\ f_i(\text{leave}) &= \text{i.leave} \\ f_i(\text{pickleft}) &= \text{i.pickup.i} \\ f_i(\text{pickright}) &= \text{i.pickup.i+1} \\ f_i(\text{putleft}) &= \text{i.putdown.i} \\ f_i(\text{putright}) &= \text{i.putdown.i+1} \\ f_i(\text{eat}) &= \text{i.eat.} \end{aligned}$$

These transformations relabel the events to conform with a standard numbering of forks: the i^{th} fork is on the left of the i^{th} philosopher's place, and philosopher i+1 sits to the right of the i^{th} philosopher.

The i^{th} philosopher is represented by the process PHIL_i :

$$\text{PHIL}_i = f_i(\text{PHIL}).$$

Note that the alphabets A_i of the PHIL_i are mutually disjoint, corresponding to our assumption that the philosophers act independently of each other. The event i.pickup.i represents the i^{th} philosopher picking up the fork to his left, and the other events are interpreted in a similar way.

We can represent a prototype fork by the recursive process

$$\text{FORK} = (\text{pickleft} \rightarrow \text{putleft} \rightarrow \text{FORK}) \,\Box\, (\text{pickright} \rightarrow \text{putright} \rightarrow \text{FORK}).$$

Using the relabellings g_i for $i = 0...4$, which leave events unchanged except for the clauses:

$$\begin{aligned} g_i(\text{pickleft}) &= \text{i-1.pickup.i} \\ g_i(\text{pickright}) &= \text{i.pickup.i} \\ g_i(\text{putleft}) &= \text{i-1.putdown.i} \\ g_i(\text{putright}) &= \text{i.putdown.i,} \end{aligned}$$

the i^{th} fork is represented by the process FORK_i :

$$\text{FORK}_i = g_i(\text{FORK}).$$

Again the alphabets B_i of individual forks are mutually disjoint, but each fork event requires participation by a philosopher, since B_i is contained in $A_{i-1} \cup A_i$.

We represent the independently executing philosophers as an interleaving, and similarly for the forks:

$$\text{PHILS} = \text{PHIL}_0 ||| \ldots ||| \text{PHIL}_4$$
$$\text{FORKS} = \text{FORK}_0 ||| \ldots ||| \text{FORK}_4.$$

Let A be the alphabet of the philosophers and B the alphabet of the forks:

$$A = \bigcup_{i=0}^{4} A_i,$$
$$B = \bigcup_{i=0}^{4} B_i.$$

Then B is a subset of A, corresponding to the fact that every fork action needs cooperation by a philosopher. We may represent the whole system as a parallel composition in which the philosophers use alphabet A and the forks use alphabet B :

$$\text{PHILS}_A \,||_B\, \text{FORKS}.$$

A sequence of actions possible for the system and leading to deadlock is

$$s = \langle 0.\text{enter}, \ldots, 4.\text{enter}, 0.\text{pickup}.0, \ldots, 4.\text{pickup}.4 \rangle.$$

One can establish this formally, by showing that

$$(\text{PHILS}_A \,||_B\, \text{FORKS}) \,\underline{\text{after}}\, s = \text{STOP}.$$

We have a large collection of theorems on process equivalence which can be used in such a proof; the identity

$$(P_A \,||_B\, Q) \,\underline{\text{after}}\, s = (P \,\underline{\text{after}}\, s\lceil A)_A \,||_B\, (Q \,\underline{\text{after}}\, s\lceil B)$$

will be useful here. The reader is invited to find a proof.

Alternatively, one can prove that for any process Q, the equality

$$(\text{PHILS}_A \,||_B\, \text{FORKS}) \,||\, (s \rightarrow Q) = (s \rightarrow \text{STOP})$$

holds, indicating that the sequence s of events always leads to deadlock.

Avoiding deadlock.

We can produce a deadlock-free version of the dining philosophers problem, by adding to the system a scheduling process to guarantee that no more than four philosophers can ever be simultaneously at the table. This solution,

due to Dijkstra, prevents deadlock, because five forks shared between four philosophers will always guarantee that some philosopher gets two forks.

A scheduling process suitable for this task is the "butler" defined as follows. The process ADMIT performs a loop, each cycle of which allows one philosopher to enter and subsequently to leave. The BUTLER process is obtained by interleaving the actions of four independent copies of the admitting process, and thus allows up to four philosophers to be in the dining room at any time:

$$\begin{aligned} \text{ADMIT} &= \Box_{i=0}^{4}(\text{i.enter} \rightarrow \text{i.leave} \rightarrow \text{ADMIT}) \\ \text{BUTLER} &= \text{ADMIT}_1 \,|||\, \ldots \,|||\, \text{ADMIT}_4, \end{aligned}$$

where each ADMIT_i is a copy of the ADMIT process. Letting C be the alphabet of the butler process, if we run the system in parallel with the butler we get

$$(\text{PHILS}_A \,\|_B\, \text{FORKS})_A \,\|_C\, \text{BUTLER}.$$

The trace s which led to deadlock in the previous system is no longer possible, since it is not a possible trace for the butler: more formally, we can show that

$$\text{initials}(\text{BUTLER} \underline{\text{after}} \langle 0.\text{enter}, \ldots, 3.\text{enter} \rangle) = \{\, \text{i.leave} \mid i = 0 \ldots 3 \,\},$$

so that the last philosopher will not be allowed into the room. This rules out the bad trace s. More importantly, one can prove that for any trace t of this system of processes, the number of philosophers in the dining room, as recorded by the difference between the numbers of "enter" events and "leave" events in t, never exceeds 4. This property will guarantee freedom from deadlock.

As is well known, this solution to the dining philosophers problem is still not entirely satisfactory. It is still possible for a particular philosopher to starve, because the other philosophers could "conspire" to exclude him from the forks. It is quite conceivable for the sequence of actions

$$t = \langle 1.\text{enter}, \ldots, 3.\text{enter}, 1.\text{pickup}.1, \ldots, 3.\text{pickup}.3 \rangle$$

to occur, setting up the table with three philosophers, each having picked up one fork. Now the fourth philosopher can enter, take his place, eat and leave, since there are two free forks, one each side of his place. This happens when the system performs the sequence

$$u = \langle 4.\text{enter}, 4.\text{pickup}.4, 4.\text{pickup}.0, 4.\text{eat}, 4.\text{putdown}.4, 4.\text{putdown}.0, 4.\text{leave} \rangle.$$

There is nothing now stopping this philosopher from repeating this cycle ad infinitum, if (perhaps) he always grows hungry before any of the seated philosophers has managed to grab a vacant fork. Thus, one philosopher can be kept out of the room unless the butler is *fair* in distributing his entry favours; this is not possible in our language of processes, as we did not stipulate that we were using a fair nondeterministic choice operator. Fairness is an issue that we are unable to treat adequately in this framework. The major disadvantage to this solution is that even though a philosopher has gained access to the table, and holds one fork, there is no guarantee that the other fork will ever become free for him to pick up; for example, philosopher 3 suffers this fate in the above scenario.

Other, more complicated, solutions to the philosophers problem have been found which do not suffer from the starvation problem. The interested reader is referred to [Di1] and [BA] for more details. [BA] contains a survey of solutions to this and other problems in the literature.

2. The mutual exclusion problem

In the mutual exclusion problem (see, for example, [Di1]) each of a group of processes is continually trying to execute its *critical section,* an area of code in which access is made to a shared resource. Because of the properties of the resource, no two processes should ever be allowed to enter their critical sections together, since only one process should access the resource at any time. A classical solution to this problem uses *semaphores* [Di1] to ensure that at any time at most one process is executing its critical section.

We may model a typical process using events to stand for the beginning and end of the critical sections, and (for simplicity) a single event for the noncritical sections; we introduce "p" and "v" events corresponding to the semaphore operations of the same name, and we also model the semaphore as a process. Consider the case when there are n processes, for $n \geq 2$. Suppose the i^{th} process P_i gains access to its critical section by executing the semaphore operation p_i and releases the semaphore with a v_i event: P_i alternates between executing its noncritical section and its critical section, punctuated by requests to the semaphore:

$$P_i = (\text{noncrit}_i \rightarrow p_i \rightarrow \text{bcrit}_i \rightarrow \text{ecrit}_i \rightarrow v_i \rightarrow P_i).$$

The semaphore simply allows a process to increment it by a p event and then waits for the process to release it with a v event:

$$\text{SEM} = \Box_{i=1}^{n}(p_i \rightarrow v_i \rightarrow \text{SEM}).$$

The system of processes, each acting independently of the others, is modelled by the interleaving

$$P = P_1 ||| \ldots ||| P_n.$$

This system, communicating with the semaphore, can be represented as:

$$Q = P_A \,||_B\, \text{SEM},$$

where A and B are the alphabets of the collection of processes and the semaphore, respectively. We may wish to hide the interactions with the semaphore, obtaining

$$Q \backslash B.$$

The traces of this process are built from the events bcrit_i, ecrit_i and noncrit_i, and the mutual exclusion condition will be satisfied if and only if in every such trace every no bcrit_j event occurs after a bcrit_i event but before the first following ecrit_i event; in other words, once a critical section is entered by one of the processes, no other process should be allowed to enter its critical section until the first one has exited. This condition can easily be formalised.

3. Networks of communicating processes

Suppose we wish to model and reason about a network of named processes, linked by channels. Work along these lines has been done by many authors, notably Chandy and Misra [CM], and Hoare and Zhou [Z,ZH1]. Each node in the network is a process, and each process communicates with its neighbours or with the environment by sending messages along the channels. Thus, an arc or channel linking two nodes represents a communication channel between two internal nodes of the network, while a channel with an unattached end represents a communication link to the environment of the network. We assume that channels intended for interaction with the environment are *named,* with channel names ranging over some set Chan. The internal channels are unnamed, since we regard an internal communication as invisible to the environment. For simplicity we suppose that each internal channel links precisely two nodes, but there may be more than one link between any two nodes or between any node and the environment.

If we wish to distinguish the direction of message passing, to model output and input, we will represent a communication by a process with its environment as an event incorporating the name of the channel, the value of the message, and the direction of the message. An event $c!v$ represents output by the process of message v along the channel named c. An event $c?v$ represents input by the process of the message v along channel c. Transmission of a message along an internal channel is regarded as an action invisible to the environment; the subsequent behaviour of the network may, of course, depend on the value of the message, so we will use an event of the form v to represent such an action. There is no need to record the direction of message passing here, as the environment is not required to participate in these actions. We assume that message passing is a synchronized activity, so that an event $c!v$ can occur only when the sending process and its environment are both ready, the environment accepting the message from the process; similarly for an input event. Internal events are uncontrollable as far as the environment is concerned, so we will consider them to be *hidden* from the environment.

A simple language for describing networks of processes contains output and input commands, choice, renaming channels, hiding channels and linking channels. Assuming that the message values range over the finite set V, and the channel names range over Chan, the universal alphabet can be taken to be the set

$$\Sigma = V \cup \{\, c!v, c?v \mid c \in \text{Chan} \;\&\; v \in V \,\}.$$

For the syntax of our language, we take

$$P ::= \text{STOP} \mid (c!v \to P) \mid (c?x \to P(x)) \mid P \,\Box\, P \mid P \sqcap P \\ \mid P[c \setminus d] \mid P[\Phi]P,$$

where Φ is a channel linking function, a partial function from channel names to channel names.

The channel set of P, $\text{chan}(P)$ is defined by structural induction:

$$\begin{aligned} \text{chan}(\text{STOP}) &= \emptyset \\ \text{chan}(c!v \to P) &= \{ c \} \cup \text{chan}(P) \\ \text{chan}(c?x \to P(x)) &= \{ c \} \cup \bigcup \{ \text{chan}(P(x)) \mid x \in V \} \\ \text{chan}(P \sqcap Q) &= \text{chan}(P) \cup \text{chan}(Q) \\ \text{chan}(P \,\Box\, Q) &= \text{chan}(P) \cup \text{chan}(Q) \\ \text{chan}(P[\Phi]Q) &= (\text{chan}(P) \cup \text{chan}(Q)) - \text{chans}(\Phi), \end{aligned}$$

where $\text{chans}(\Phi) = \text{dom}(\Phi) \cup \text{range}(\Phi)$ is the set of channel names affected by Φ. Similarly, we may define the input channel set $\text{inputs}(P)$ and the output channel set $\text{outputs}(P)$ of a term P. We impose a syntactic constraint that no channel is used by a process for both input and output, that is, the sets $\text{inputs}(P)$ and $\text{outputs}(P)$ are disjoint.

Each term in this language can be considered as representing a process whose events denote communications, either input or output, with its environment.

Most of the syntactic constructs are familiar from earlier sections. In this setting, the intuitions behind the various constructs are as follows. We explain each operation in terms of previously defined operations.

Inaction. STOP is the process which is incapable of performing any action.

Output. A process of the form

$$(c!v \to P)$$

wishes first to output the message v on channel c; this is a simple prefixing operation.

Input. A process

$$(c?x \to P(x))$$

is initially willing to input any message at all (ranging over the set V) on channel c, and its behaviour thereafter is dependent on the value input. In this syntactic construction x is understood to be a bound variable ranging over V, the set of all possible message values. The terms $P(x)$ denote a process for each $x \in V$, and if the value input initially is v the process continues by behaving like $P(v)$. We regard this input process as having a conditional choice between the various inputs on the channel c, the choice being determined by the environment in which the process is placed. We therefore set

$$(c?x \to P(x)) = \Box_{v \in V}(c?v \to P(v)).$$

Choice. The two forms of choice, nondeterministic and conditional, are again the $\sqcap$ and $\Box$ operations: for example, the process

$$(a?x \to P(x)) \sqcap (b?x \to Q(x))$$

can choose arbitrarily to wait for input on one of its two channels, and if the environment can only send along the a channel (say), there is a possibility of deadlock. On the other hand, the process

$$(a?x \to P(x) \Box (b?x \to Q(x))$$

must initially accept input from the environment on either channel, and cannot commit itself to listening along only one of these channels.

Renaming. The renaming operation $P[c \setminus d]$, where $c, d \in \mathrm{Chan}$, simply changes channel names: all events of the form $c!v$ and $c?v$ become $d!v$ and $d?v$ respectively; all other events are unaffected. This is an example of an alphabet transformation: the appropriate transformation is f_c^d, defined by

$$\begin{aligned} f_c^d(v) &= v, \\ f_c^d(e!v) &= d!v, \quad \text{if } e = c, \\ &= e!v, \quad \text{otherwise} \\ f_c^d(e?v) &= d?v, \quad \text{if } e = c, \\ &= e?v, \quad \text{otherwise,} \end{aligned}$$

for all $v \in V$. We define

$$P[c \setminus d] = f_c^d[P].$$

Linking. A linking operation Φ is a partial injective function on channel names. We use the notation

$$[c_1 \Leftrightarrow d_1, \ldots, c_n \Leftrightarrow d_n],$$

where the c_i and d_j are assumed distinct, for the function mapping c_i to d_i $(i = 1, \ldots n)$. This particular linking function joins the channels named c_i in one process to the corresponding d_i in another. We will only allow use of a linking operation when the processes involved have disjoint channel sets, and when the linkage will pair input channels of one with outputs of the other: in any combination $P[\Phi]Q$ we require that Φ be injective, chan(P) and chan(Q) be disjoint, and for all c, d,

$$c \in \text{inputs}(P) \ \&\ \Phi(c) = d \ \Rightarrow d \in \text{outputs}(Q),$$
$$c \in \text{outputs}(P) \ \&\ \Phi(c) = d \ \Rightarrow d \in \text{inputs}(Q).$$

The trivial linking operation, which makes no connections between channels, can be modelled by interleaving, since we are assuming that the processes involved have disjoint channel sets. In order to link two channels, say input channel c of P and output channel d of Q, we first rename all pairs of events which should be synchronized, by applying the alphabet transformation

$$g(c?v) = v,$$
$$g(d!v) = v,$$
$$g(e) = e, \quad \text{otherwise.}$$

Then we form the parallel composition

$$P_A \|_B Q,$$

where A and B are the sets of events:

$$A = \alpha(P) \cup V,$$
$$B = \alpha(Q) \cup V.$$

This allows P and Q to progress independently on their individual external channels, but forces them to synchronize on their internal communications. Finally, we hide all internal actions: thus, we set

$$P[c \Leftrightarrow d]Q = (P_A \|_B Q)\backslash V,$$

in this case. In general, $P[\Phi]Q$ is obtained by a composition of renamings, followed by a restricted parallel composition and hiding. Define for a linking function Φ an alphabet transformation h by:

$$h(c?v) = v, \quad \text{if } c \in \text{chans}(\Phi),$$
$$h(c!v) = v, \quad \text{if } c \in \text{chans}(\Phi),$$
$$h(e) = e, \quad \text{otherwise.}$$

Applying h to P renames all events along the channels to be linked; similarly for Q. Letting A and B be the appropriate alphabets, we define

$$P[\Phi]Q = (h[P]_A \|_B h[Q])\backslash V.$$

Examples.

Example 1. A simple process which repeats a cycle of inputting a message on channel "in" and then outputting the same message on channel "a" can be defined by recursion:

$$P = \mu p.(\text{in}?x \rightarrow \text{a}!x \rightarrow p).$$

This defines a deterministic process satisfying the equation

$$P = (\text{in}?x \rightarrow \text{a}!x \rightarrow P).$$

It can be thought of as a simple "buffer" process with capacity 1, since it can store at most one value before outputting it: the process is initially able to input any value whatsoever, whereupon its only possible next action is to output that value.

Example 2. Let P be the buffer process above, and let Q be another buffer process which repeatedly inputs on channel "b" and outputs on channel "out":

$$Q = \mu q.(\text{b}?x \rightarrow \text{out}!x \rightarrow q).$$

If we form a link, joining output channel a of P to input channel b of Q, we get

$$R = P[a \Leftrightarrow b]Q.$$

Intuitively, we are chaining together two buffers, each of capacity 1; it should not be surprising that the result behaves like a buffer with capacity 2. R can store two values before refusing further input. Using the fixed point properties of P and Q, and the fact that all operations involved are continuous, we get:

$$R = \Box_{v \in V}(\text{in}?v \rightarrow R_v),$$

where for each $v \in V$ the process R_v satisfies:

$$\begin{aligned} R_v = &\Box_{w \in V}\ (\text{in}?w \rightarrow \text{out}!v \rightarrow R_w) \\ &\Box (\text{out}!v \rightarrow R). \end{aligned}$$

This set of mutually recursive equations can be taken as *defining* the processes R and R_v, for $v \in V$, or as theorems provable from the fixed point equations. Intuitively, R_v represents the behaviour of R after it has input the value v. From the equation for R_v we see that at this stage the process can either output v immediately (and then become the empty buffer) or else input a further value w; the behaviour of a full buffer, containing values v and w, is

simply specified: no further input is permitted until after output of the value v, when the buffer will only contain the value w.

Essentially the same example was discussed in [HBR], where the notation

$$(P \gg Q)$$

was used for the result of forcing everything output by P to be simultaneously input by Q (and hiding these internal communications). Roscoe has developed useful proof techniques for establishing properties of recursively defined processes, using *recursion induction rules*. These buffer processes provide many interesting examples. For example, one can formalise the condition for a process to be a buffer of length n, (parameterised on n) and prove that the result of chaining together n copies of a single length buffer is a buffer of length n. In the notation of [HBR], if $B1$ is a buffer of length 1 then the process B_i given in the sequence of definitions

$$B_1 = B1,$$
$$B_{n+1} = (B_n \gg B1),$$

is a buffer of length i. A proof can be based on the collection of process identities built up in Chapter 2, or perhaps by adapting the proof system of Chapter 5. Many other interesting properties can be established, such as the fact that chaining together a buffer of length n and a buffer of length m produces a buffer of length $n + m$:

$$(B_n \gg B_m) = B_{n+m}.$$

The proofs rely on the associativity of the $\gg$ operator, which only holds in the failure set model in the absence of divergence. Applying the chaining operation can only produce divergence if the two processes could indulge in infinite chatter along the channels being linked. This would imply the ability of the first process to produce arbitrarily long sequences of output after only a finite amount of input had been consumed; this problem cannot, therefore, arise when chaining buffer processes of finite capacity, as here. For more details, see [HBR] and [R].

Chapter 8

Operational semantics

0. Introduction

The semantics for processes presented in this thesis has been continually motivated by appealing to intuitions about how parallel execution of communicating processes can be modelled. We have been arguing in terms of sequences of actions and their consequences. This is an operationally oriented approach; it is not surprising, therefore, that we can give a formal operational semantics, in the style of Plotkin [P1] or Hennessy and Milner [HM], to the terms in our language. Since the language has not involved any imperative constructs (such as assignment) there is no need to model machine states; the language is "applicative." We may give an operational semantics by defining an appropriate set **Conf** of configurations, representing the partial results or stages in a computation, and a set of *transition relations* between configurations, specifying how configurations can alter during execution of a process. Since we are dealing with an event-based model, it is natural to follow Plotkin and use a *labelled transition system,* in which transitions are associated with event labels.

1. Operational semantics

The presentation which follows is a variant on the semantics given in [HBR], where the failure set semantics of our language was introduced using operational style definitions. Here we extend these ideas to cover the full language of this thesis, including recursion and divergence. To be precise, we will consider the language of processes defined by the grammar:

$$P ::= \mathrm{STOP} \mid \mathrm{SKIP} \mid \bot \mid (a \to P) \mid P \sqcap P \mid P \,\Box\, P \mid P \| P$$
$$P |\!|\!| P \mid P;P \mid f[P] \mid f^{-1}[P] \mid P/a \mid p \mid \mu p.P,$$

where, as usual, a ranges over Σ and p ranges over a set of process identifiers. For the set of configurations we take the closed terms in this language, that is, the terms in which no identifier occurs free. Let

$$\mathbf{Conf} = \{\, P \mid P \text{ is a closed term} \,\}.$$

Transitions will have one of two forms: a transition labelled with an event a,

$$P \xrightarrow{a} Q$$

can be interpreted "P can transform to process Q by performing an a-action"; this will be called a *visible* transition. An *invisible* or *silent* transition has the form

$$P \longrightarrow Q$$

and indicates that P can decide autonomously to behave like Q; the labels on transitions represent single atomic actions, either visible or invisible, so that we can think of a transition as occurring because of a single atomic action or nondeterministic decision. Formally, we are working with a labelled transition system

$$(\mathbf{Conf},\ \{ \xrightarrow{a} \mid a \in \Sigma \},\ \longrightarrow).$$

In the usual way, following Milner and [HBR], we also define the transition relations $\stackrel{s}{\Longrightarrow}$ for $s \in \Sigma^*$.

$$P \stackrel{s}{\Longrightarrow} Q$$

will mean that it is possible for P to transform to Q by performing the sequence s of visible actions, possibly with invisible actions on the way. We set, for $s = \langle a_1, \ldots, a_n \rangle$,

$$P \xrightarrow{s} Q \;\Leftrightarrow\; P \longrightarrow^* P_0 \xrightarrow{a_1} P_1 \longrightarrow^* P_2 \xrightarrow{a_2} \longrightarrow^* \cdots \xrightarrow{a_n} P_{n+1} \longrightarrow^* Q,$$

for some sequence of terms P_i. In particular,

$$P \stackrel{\langle\rangle}{\Longrightarrow} Q$$

means that P can transform itself invisibly to Q.

The transition system in [HBR] was given in terms of these $\stackrel{s}{\Longrightarrow}$ relations, but with different notation. Here we prefer to begin by giving axioms and rules for the *single step* behaviour of terms. We will see that the axioms and rules of the [HBR] transition system are derivable.

We will also need a predicate $\uparrow$ on terms to distinguish the *divergent* configurations: roughly,

$$P\uparrow$$

means that P contains an unguarded recursion or unguarded occurrence of the subterm $\bot$.

The transition relations and divergence predicate are generated from the following formal system. More precisely, the valid transitions are those derivable within this formal system, so the transition relations are defined to be the smallest relations in which all provable transitions hold; similarly, the divergence predicate is the smallest predicate consistent with the set of provable divergences. The axioms and inference rules are intended to correspond closely to the operational ideas behind the syntactic constructs. In each case except for recursion we state a "one step" version of an axiom or rule given in [HBR]. For recursive terms we add rules allowing the transitions of $\mu p.P$ to be deduced from the transitions of P.

Transition system.

We give a set of transition axioms and inference rules for each syntactic construct.

(SKIP) $\quad \mathrm{SKIP} \xrightarrow{\surd} \mathrm{STOP}$

(PRE) $\quad (a \to P) \xrightarrow{a} P$

(OR_1) $\quad (P \sqcap Q) \longrightarrow P \qquad (P \sqcap Q) \longrightarrow Q$

(OR_2) $\quad \dfrac{P \xrightarrow{a} P'}{(P \sqcap Q) \xrightarrow{a} P'} \qquad \dfrac{Q \xrightarrow{a} Q'}{(P \sqcap Q) \xrightarrow{a} Q'}$

(COND_1) $\quad \dfrac{P \longrightarrow P',\ Q \longrightarrow Q'}{(P \,\Box\, Q) \longrightarrow (P' \,\Box\, Q')}$

(COND_2) $\quad \dfrac{P \xrightarrow{a} P'}{(P \,\Box\, Q) \xrightarrow{a} P'} \qquad \dfrac{Q \xrightarrow{a} Q'}{(P \,\Box\, Q) \xrightarrow{a} Q'}$

(PAR_1) $\quad \dfrac{P \longrightarrow P'}{(P \parallel Q) \longrightarrow (P' \parallel Q)} \qquad \dfrac{Q \longrightarrow Q'}{(P \parallel Q) \longrightarrow (P \parallel Q')}$

(PAR_2) $\quad \dfrac{P \xrightarrow{a} P',\ Q \xrightarrow{a} Q'}{(P \parallel Q) \xrightarrow{a} (P' \parallel Q')}$

(INT_1) $\quad \dfrac{P \longrightarrow P'}{(P \,|||\, Q) \longrightarrow (P' \,|||\, Q)} \qquad \dfrac{Q \longrightarrow Q'}{(P \,|||\, Q) \longrightarrow (P \,|||\, Q')}$

(INT_2) $\quad \dfrac{P \xrightarrow{a} P'}{(P \,|||\, Q) \xrightarrow{a} (P' \,|||\, Q)} \qquad \dfrac{Q \xrightarrow{a} Q'}{(P \,|||\, Q) \xrightarrow{a} (P \,|||\, Q')}$

$$(\mathrm{SEQ}_1)\qquad \frac{P \longrightarrow P'}{(P;Q) \longrightarrow (P';Q)}$$

$$(\mathrm{SEQ}_2)\qquad \frac{a \neq \surd,\ P \xrightarrow{a} P'}{(P;Q) \xrightarrow{a} (P';Q)}$$

$$(\mathrm{SEQ}_3)\qquad \frac{P \xrightarrow{\surd} .}{(P;Q) \longrightarrow Q}$$

$$(\mathrm{ALPH}_1)\qquad \frac{P \longrightarrow P'}{f[P] \longrightarrow f[P']}$$

$$(\mathrm{ALPH}_2)\qquad \frac{P \xrightarrow{a} Q,\ f(a) = b}{f[P] \xrightarrow{b} f[Q]}$$

$$(\mathrm{INV}_1)\qquad \frac{P \longrightarrow Q}{f^{-1}[P] \longrightarrow f^{-1}[Q]}$$

$$(\mathrm{INV}_2)\qquad \frac{P \xrightarrow{a} Q,\ f(b) = a}{f^{-1}[P] \xrightarrow{b} f^{-1}[Q]}$$

$$(\mathrm{HIDE}_1)\qquad \frac{P \longrightarrow Q}{(P/a) \longrightarrow (Q/a)}$$

$$(\mathrm{HIDE}_2)\qquad \frac{P \xrightarrow{a} Q}{(P/a) \longrightarrow (Q/a)}$$

$$(\mathrm{HIDE}_3)\qquad \frac{P \xrightarrow{b} Q,\ b \neq a}{(P/a) \xrightarrow{b} (Q/a)}$$

$$(\mu_1)\qquad \frac{P \longrightarrow Q}{(\mu p.P) \longrightarrow (Q[(\mu p.P) \setminus p])}$$

$$(\mu_2)\qquad \frac{P \xrightarrow{a} Q}{(\mu p.P) \xrightarrow{a} (Q[(\mu p.P) \setminus p])}$$

where $Q[R \setminus p]$ denotes as usual the term obtained by substituting R for all free occurrences of p in Q. As usual a and b range over Σ and f ranges over alphabet transformations.

The divergence predicate is generated from the following rules and axioms:

$(D1)$ $\quad \bot\uparrow$

$(D2)$ $\quad p\uparrow$

$(D3)$ $\quad P\uparrow \Rightarrow f[P]\uparrow$

$(D4)$ $\quad P\uparrow \Rightarrow f^{-1}[P]\uparrow$

$(D5)$ $\quad P\uparrow \Rightarrow (P/a)\uparrow$

$(D6)$ $\quad P\uparrow \Rightarrow (\mu p.P)\uparrow$

$(D7)$ $\quad P\uparrow \Rightarrow (P;Q)\uparrow$

$(D7)$ $\quad P\uparrow \Rightarrow (P\ op\ Q)\uparrow,\ (Q\ op\ P)\uparrow \qquad (op = \sqcap, \Box, \|, |||)$

$(D8)$ $\quad P \stackrel{a^\infty}{\Longrightarrow} \cdot \Rightarrow (P/a)\uparrow$

$(D9)$ $\quad P \Longrightarrow Q\ \&\ Q\uparrow \Rightarrow P\uparrow$

where $P \stackrel{a^\infty}{\Longrightarrow} \cdot$ is an abbreviation for:

$$\forall n.\exists P_n.P \stackrel{a^n}{\Longrightarrow} P_n.$$

Clause $(D8)$ reflects our decision that a term which can engage in arbitrarily long a-actions can diverge when a is hidden. Because this divergent property is not deducible form one–step behaviour alone, we have introduced this infinitary rule into the transition system. The final clause $(D9)$ says that a term which can make a silent transition to a divergent term can itself diverge.

Examples.

Example 1. STOP has no transitions.

Example 2. The term P given below has the following transitions:

$$P = \mu p.((a \rightarrow p) \Box (b \rightarrow \text{STOP}))$$

$P \stackrel{a}{\longrightarrow} P$ by (μ_2)

$P \stackrel{b}{\longrightarrow} \text{STOP}$ by (μ_2)

Thus, the term P/a can diverge and also has the transitions:

$(P/a)\uparrow$ by $(D8)$

$(P/a) \longrightarrow (P/a)$ by (HIDE_2)

$(P/a) \stackrel{b}{\longrightarrow} (\text{STOP}/a)$ by (HIDE_3)

The term STOP/a has no transitions, because the transitions of a term of the form Q/a are defined from the transitions of Q, and STOP has no transitions.

Derived rules and axioms.

The following transition rules, essentially those given in [HBR], are easily seen to be derivable in the above system.

$$P \xrightarrow{\langle\rangle} P$$

$$P \xrightarrow{s} Q \xrightarrow{t} R \;\Rightarrow\; P \xrightarrow{st} R$$

$$\frac{P \xrightarrow{s} Q}{(a \to P) \xrightarrow{as} Q}$$

$$\frac{P \xrightarrow{s} P'}{(P \sqcap Q) \xrightarrow{s} P'} \qquad \frac{Q \xrightarrow{s} Q'}{(P \sqcap Q) \xrightarrow{s} Q'}$$

$$\frac{s \neq \langle\rangle,\ P \xrightarrow{s} P'}{(P \,\Box\, Q) \xrightarrow{s} P'} \qquad \frac{s \neq \langle\rangle,\ Q \xrightarrow{s} Q'}{(P \,\Box\, Q) \xrightarrow{s} Q'}$$

$$\frac{P \xrightarrow{\langle\rangle} P',\ Q \xrightarrow{\langle\rangle} Q'}{(P \,\Box\, Q) \xrightarrow{\langle\rangle} (P' \,\Box\, Q')}$$

$$\frac{P \xrightarrow{s} P',\ Q \xrightarrow{s} Q'}{(P \parallel Q) \xrightarrow{s} (P' \parallel Q')}$$

$$\frac{u \in \mathrm{merge}(s,t),\ P \xrightarrow{s} P',\ Q \xrightarrow{t} Q'}{(P \,|||\, Q) \xrightarrow{u} (P' \,|||\, Q')}$$

$$\frac{P \xrightarrow{s} Q}{(P/a) \xrightarrow{s \setminus a} (Q/a)}$$

$$\frac{P \xrightarrow{s} Q,\ f(s) = t}{f[P] \xrightarrow{t} f[Q]}$$

$$\frac{P \xrightarrow{t} Q,\ f(s) = t}{f^{-1}[P] \xrightarrow{s} f^{-1}[Q]}$$

$$\frac{P \xrightarrow{s} Q,\ s \text{ tick-free}}{P;Q \xrightarrow{s} P';Q}$$

$$\frac{P \xrightarrow{s\surd} \cdot,\ s \text{ tick-free}}{P;Q \xrightarrow{s} Q}$$

A derived rule for recursive terms is:

$$\frac{P \stackrel{s}{\Longrightarrow} Q}{(\mu p.P) \stackrel{s}{\Longrightarrow} Q[(\mu p.P) \setminus p]}$$

One can regard these laws and rules as defining a transition system

$$(\mathbf{Conf}, \{ \stackrel{s}{\Longrightarrow} \mid s \in \Sigma^* \})$$

in the sense that each $\stackrel{s}{\Longrightarrow}$ relation is taken to be the smallest relation consistent with the axioms and rules of the formal system.

Extracting failures.

Now that we have defined a transition system on terms, we can filter out the failures and divergence sets of a term (defined now on the set of transitions involving that term). We claim that precisely the same failure and divergence semantics is obtained as the version given in Chapter 5.

Definition 8.1.1: For a closed term P,

(i) $\text{failures}(P) = \{ (s, X) \mid \exists Q.P \stackrel{s}{\Longrightarrow} Q \ \&\ \forall x \in X.\neg(Q \stackrel{x}{\Longrightarrow} \cdot) \}$
$\qquad\qquad \cup \{ (st, X) \mid \exists Q.P \stackrel{s}{\Longrightarrow} Q \ \&\ Q\uparrow \}$

(ii) $\text{div}(P) = \{ st \mid \exists Q \mid P \stackrel{s}{\Longrightarrow} Q \ \&\ Q\uparrow \}$

For example, we see that

$$\text{failures}(\bot) = \{ (s, X) \mid s \in \Sigma^* \ \&\ X \in p\Sigma \},$$
$$\text{div}(\bot) = \Sigma^*,$$

because by (D1) $\bot\uparrow$.

Recall the semantic functions $\mathcal{F}$ and $\mathcal{D}$ used in Chapter 5. I have designed the transition system above with the intention that the failures and divergences defined on the transition set of all closed terms P should agree with the semantics given to the term in Chapter 5. It should be possible to prove that, for all closed terms P and all environments u, the following equations hold:

$$(1) \qquad \mathcal{F}[\![P]\!]u = \text{failures}(P)$$
$$(2) \qquad \mathcal{D}[\![P]\!]u = \text{div}(P).$$

Future work will tackle this problem, which seems to require a combination of structural induction on P and induction on the length of traces. Notice though that if we only consider finite terms (i.e., not containing any recursive subterm) then (1) and (2) hold: all divergence sets will be empty, and all finite terms have only finitely many transitions. A simple proof based on the length of the *proof* within the formal transition system that $P \stackrel{s}{\Longrightarrow} Q$ should suffice, together with a structural induction on P. Similar results are stated in [HBR].

Conclusions

In conclusion let us summarise briefly the contents of this thesis, pointing out the places where future work might be profitable, and making some remarks on relevant work reported elsewhere.

Beginning in Chapter 1 with a construction of a domain of *failure sets,* partially ordered by a nondeterminism ordering, we were able to use this as the basis for a semantics of an abstract version of CSP. Identifying a process with its failure set, we defined a collection of process operations in Chapter 2, including prefixing, two forms of alternative construct, various parallel compositions, hiding, alphabet transformation and sequential composition. These operations are similar to those described by Hoare in his work on CSP. In each case we proved process identities showing how various forms of process composition can be related. The hiding operation introduced a phenomenon called *internal chatter* or *divergence,* which can occur typically when a process is capable of performing arbitrarily long sequences of internal actions without responding to the communication requests of its environment. The treatment of divergence in Chapter 2 was not entirely satisfactory, because it identified divergent processes with certain non-divergent processes. The extended semantic model of Chapter 5 gives a more intuitively pleasing treatment of divergence, in which such identifications are no longer made. More work needs to be done here in order to complete this investigation. That this is a promising model is shown by the complete proof system (also in Chapter 5) for the extended nondeterminism relation on processes.

Chapter 3 contains a different formulation of the failures model, based on sets of trees. Since trees represent trace sets of deterministic processes, this approach models a nondeterministic process as a set of deterministic processes. We show that each process operation of Chapter 2 is uniquely determined by its effect on deterministic processes, a condition we called *implementability.* This formulation of the failures model is closely related to the process model of Kennaway [K1,2], as we show in Chapter 6.

Chapter 4 uses Milner's synchronisation trees to provide yet another representation for processes, based on a *failure equivalence* relation on trees. Milner used synchronisation trees and an *observational equivalence* relation to give a semantics to his CCS in [M1]. He showed that the congruence relation generated by observation equivalence and his CCS tree operations was axiomatizable, at least on finite synchronisation trees. We defined a set of CSP operations on trees which respect failure equivalence and faithfully mirror the CSP process operations. Thus, the failure equivalence relation is already a congruence with respect to the CSP operations. We also gave a complete proof system for failure equivalence on finite trees. Other relevant work along these lines is contained in [D]. The differences between these two axiomatic systems illustrate the different characteristics of the CSP failures equivalence and the CCS observation equivalence. Much of the content of this chapter also appears in [B]. More work is suggested by our results. We should consider whether or not we can extend these results to infinite synchronisation trees (i.e., by considering recursion). It might also be possible to extract some results on translating CSP into CCS, by mapping the original CSP notation into our abstract version of the language. Some effort in this area, although not using this line of attack, was reported in [HLP]. It would also be of interest to consider how the more recent work of Milner, for example [M2,3], involving finite delay operators, fits into this framework. In [RB2] we report some connections between programming logics and behavioural equivalence relations.

The complete proof system of Chapter 5 for assertions of the form $P \sqsubseteq Q$, interpreted to mean that the semantic value of P approximates that of Q, seems capable of deriving many useful process properties. We have yet to investigate its effectiveness when applied to large processes, although there is hope that it will turn out to be useful. As we noted, the proof system is infinitary: it seems likely that in general one needs to prove a property of all finite syntactic approximations of a recursive process in order to deduce properties of the recursive process. Nevertheless, some interesting subsystems should have decidable theories, and future investigation here is necessary. We conjecture that restricting to allow only properly guarded recursions would produce a decidable subsystem. A proof system for a version of CCS is contained in [HN], where again the system is infinitary.

Other authors have reported several proof systems for CSP-like languages. Apt, Francez and de Roever [AFR] gave a Hoare-style assertion system for the full imperative CSP as described by Hoare in his original paper. Chandy and Misra [CM] have published proofs for networks of communicating processes. Our model is most closely related to the proof systems of Hoare [H3] and Zhou [Z,ZH1,2]. Zhou gives a partial correctness proof for the HDLC

protocol in [ZH2].

Chapter 6 began from an investigation by the author into the work of Hennessy and de Nicola [HN]. It shows that some elegant characterizations can be given of the various equivalence relations on synchronisation trees underlying the various models: failure equivalence, observation equivalence, and Kennaway's equivalence. This is a very interesting result, because it illustrates how subtle the distinctions between these models are. It is not yet clear how far we can extend these ideas to cover still more semantic models for concurrency in the literature. In cases such as the *possible futures* model [RB1], where sequentiality and the representation of concurrency by nondeterministic interleaving are still assumed, we might expect further connections to exist. In addition, we would like to investigate the relationship of Winskel's *event structure* models [W1,2] and our models; perhaps some useful results may be obtained by attaching modal assertions to elements of these more general models in the same way that modal assertions were used to reason about synchronisation trees in Chapter 6. Much remains to be done in this area.

Chapter 7 applies the process identities established in Chapter 2 to some reasonably sized problems well known in concurrency. More examples can be found in [R].

The final chapter sketches an operational semantics for our abstract CSP, basically derived from that of [HBR] by adding laws for recursion. There is an obvious definition of failure sets and divergence sets in this operational framework, and we conjecture that precisely the same semantics would be obtained as in Chapter 5. Again there is further work to be done here.

Appendix A

A technical lemma

This Appendix contains the proof of a lemma used in proving properties of the hiding operators in Chapter 2. This lemma concerns application of a hiding operation to a sequence of traces. When each resulting trace is a prefix of some fixed trace, we show that one of two alternative conclusions can be drawn: either the sequence has a constant subsequence, or there is a subsequence consisting of arbitrarily long extensions of some fixed trace. This result was used in the proof of continuity for the hiding operation /.

Lemma A: If $\{ s_n \mid n \geq 0 \}$ is a set of traces and B is a finite set of events such that the traces $s_n \backslash B$ are all prefixes of some trace u, i.e., ,

$$s_n \backslash B \leq u, \quad \text{for all } n,$$

then either infinitely many s_n are identical or there is an infinite sequence $t \in B^\infty$ and a trace s such that

$$\forall k. \exists n.\ st_k \leq s_n,$$

where for each k t_k is the prefix of t having length k.

Proof. Let $\{ s_n \mid n \geq 0 \}$ and B satisfy the conditions above. Let u be a trace such that for all n

$$s_n \backslash B \leq u.$$

Since u has finite length, it has only a finite number of distinct prefixes; at least one of these prefixes must be equal to infinitely many $s_n \backslash B$, i.e., there is an infinite subsequence $\{ s_{n_k} \mid k \geq 0 \}$ and a prefix $v \leq u$ such that

$$\forall k.\ s_{n_k} \backslash B = v.$$

Using such a subsequence and prefix to replace the given ones if necessary, we can therefore assume without loss of generality that

$$\forall n.\ s_n \backslash B = u.$$

Also without loss of generality u contains no events in B, say

$$u = \langle a_1, \ldots, a_r \rangle,$$

where each $a_i \in \Sigma - B$. Each s_n must be obtained from u by adding sequences of events in B, and must therefore have the form

$$s_n = u_n^0 a_1 u_n^1 \ldots u_n^{r-1} a_r u_n^r \quad (1),$$

where each $u_n^i \in B^*$.

Now there are two possibilities to consider, depending on whether or not the s_n are of uniformly bounded length.

Case 1 If the s_n are uniformly bounded in length, we argue as follows. Since B is a finite set and the events $a_1, \ldots, a_r$ are fixed, there are only finitely many distinct traces with bounded length of the form (1) above. So some trace s must appear infinitely often in the list $\{ s_n \mid n \geq 0 \}$ and we have that infinitely many s_n must be identical.

Case 2 If the s_n are not uniformly bounded in length, then there must be some position i such that the traces u_n^i are unbounded in length. Taking the smallest such i, the argument for Case 1 applies to the sequence of truncated traces

$$s_n^i = u_n^0 a_1 \ldots u_n^{i-1} a_i,$$

so that there is an infinite subsequence of the s_n all of whose truncations s_n^i are identical. Replacing the original sequence by this subsequence, we may suppose that there is a trace s such that each s_n has form

$$s_n = s u_n^i w_n,$$

where $w_n = a_{i+1} u_n^{i+1} \ldots u_n^r$.

Since the u_n^i are assumed to be of unbounded length, B is a finite set, and each $u_n^i \in B^*$, we may apply Konig's Lemma to deduce the existence of an increasing subsequence of u_n^i : there is a subsequence indexed by k such that

$$u_{n_k}^i < u_{n_{k+1}}^i,$$

for all k. This sequence of finite traces has as limit an infinite trace t, which has the property required:

$$\forall k \exists n.\ st_k \leq s_n.$$

That completes the proof. ∎

References

[AFR] Apt, K.R., Francez, N., de Roever, W.P., A Proof System for Communicating Sequential Processes, ACM Transactions on Programming. Languages and Systems, Vol 2, No.3, July 1980.

[BA] Ben–Ari, M., Principles of Concurrent Programming, Prentice-Hall International (1982).

[B] Brookes, S.D., On the relationship of CCS and CSP, submitted to ICALP 1983.

[CM] Chandy, K.M., and Misra, J., An axiomatic proof technique for networks of communicating processes, Technical report TR-98, Dept. of Computer Science, Univ. of Texas at Austin (1979).

[D] Darondeau, Ph., An enlarged definition and complete axiomatization of observational congruence of finite processes, Springer LNCS Vol.137, pp. 47-62. (1982)

[Di1] Dijkstra, E.W., Cooperating sequential processes, *in* Programming Languages, ed. F. Genuys, Academic Press.

[Di2] Dijkstra, E.W., Guarded commands, nondeterminacy and a calculus for the derivation of programs, Communications of ACM, 1975 Vol.8.

[Gu] Guessarian, I., Algebraic semantics, Springer LNCS Vol.99 (1981).

[HLP] Hennessy, M., Li,W., and Plotkin, G., A first attempt at translating CSP into CCS, Proc. of 2^{nd} International Conference on Distributed Computer Systems, IEEE Computer Society Press (1981).

[HM] Hennessy, M. and Milner, R., On observing nondeterminism and concurrency, Springer LNCS Vol. 85. (1979).

[HN] Hennessy, M., and de Nicola, R., Testing Equivalences for Processes, Internal Report, University of Edinburgh, (July 1982).

[HP1] Hennessy, M.C.B. and Plotkin, G., A term model for CCS, Proc. 9^{th} Conference on Mathematical Foundations of Computer Science, Springer LNCS Vol. 88. (1980)

[H1] Hoare, C.A.R., Communicating Sequential Processes, CACM 21, vol. 8 (1978).

[H2] Hoare, C.A.R., A Model for Communicating Sequential Processes, Technical Report PRG-22, Programming Research Group, Oxford University Computing Lab. (1981)

[H3] Hoare, C.A.R., A calculus of total correctness for communicating processes, Technical Monograph PRG-23, Oxford University Computing Lab, Programming Research Group (1981).

[HBR] Hoare, C.A.R., Brookes, S.D., and Roscoe, A.W., A Theory of Communicating Processes, Technical Report PRG-16, Programming Research Group, University of Oxford (1981); to appear also in JACM.

[K1] Kennaway, J.R., Formal semantics of nondeterminism and parallelism, Ph.D thesis, University of Oxford (1981).

[K2] Kennaway, J.R., A theory of nondeterminism, Springer LNCS 85, pp. 338-350 (1980).

[LNS] Lassez, J.-L., Nguyen, V.L., and Sonenberg, E.A., Fixed Point Theorems and Semantics: A Folk Tale. Information Processing Letters, Vol. 14 No. 3 (May 1982).

[M1] Milner, R., A Calculus of Communicating Systems. Springer LNCS Vol. 92 (1980).

[M2] Milner, R., On relating Synchrony and Asynchrony, Technical report, Computer Science Department, University of Edinburgh (1980).

[M3] Milner, R., A finite delay operator in Synchronous CCS, Internal Report CSR-116-82, Computer Science Department, University of Edinburgh (1982)

[MM] Milne, G. and Milner, R., Concurrent processes and their syntax, JACM 26,2, (1979).

[P1] Plotkin, G., A structural approach to operational semantics. DAIMI FN–19 Comp. Sc. Dept., Aarhus University (1981).

[P2] Plotkin, G., An Operational Semantics for CSP, W.G.2.2 conference 1982.

[R] Roscoe, A.W., A Mathematical Theory of Communicating Processes, Ph.D thesis, University of Oxford (1982).

[RB1] Brookes, S.D., and Rounds, W.C., Possible futures, refusals and communicating processes, Proceedings of 22^{nd} Annual IEEE Symposium on Foundations of Computer Science (1981).

[RB2] Brookes, S.D., and Rounds, W.C., Behavioural Equivalence Relations Induced by Programming Logics, submitted to ICALP 1983.

[W1] Winskel, G., Event structure semantics of CCS and related languages, Springer LNCS Vol.140 (1982).

[W2] Winskel, G., Categories of synchronisation trees and transition systems, submitted to ICALP 1983.

[Z] Zhou Chaochen, The consistency of the calculus for total correctness of communicating processes, Technical Report PRG-26, Oxford University Programming Research Group (1982).

[ZH1] Zhou Chaochen, C.A.R.Hoare, Partial correctness of Communicating Sequential Processes, Proc. International Conference on Distributed Computing (April 1981).

[ZH2] Zhou Chaochen, C.A.R.Hoare, Partial correctness of Communications Protocols, Proc. INWG/NPL Workshop on Protocol Testing (May 1981).