LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

Operational and Algebraic Semantics of Concurrent Processes

by

Robin Milner

LFCS Report Series

ECS-LFCS-88-46
(also published as CSR-258-88)

February 1988

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Copyright © 1987, LFCS

Operational and Algebraic Semantics of Concurrent Processes

Robin Milner
University of Edinburgh
November 1987

Chapter for the
Handbook of Theoretical Computer Science

1. Introduction

This Chapter is concerned with a calculus for expressing and analysing concurrent processes which may communicate with one another. The view taken is that processes are not mathematical objects whose nature is universally agreed, and that therefore we cannot undertake this study by starting from a well-understood domain of entities known as processes. Rather we adopt the approach by which the lambda calculus was first introduced. We begin by defining a small language whose constructions reflect simple operational ideas; the meaning of these constructions is presented by means of structured operational semantics [Plo]. But although we take the liberty of calling the closed terms of our language *processes*, we do not claim to have defined the semantic notion of process until we have introduced a suitable congruence relation over these terms; then a (semantic) process is understood to be a congruence class of terms.

This congruence relation is therefore interpreted as equality of processes. Much of the Chapter is taken up with developing the algebra of this equality, in particular with the existence and uniqueness of solutions of equations which define processes recursively. The essence of the congruence is that it is built upon the idea of *observing* a process, which we take to mean communicating with it; the idea is that processes are equal iff they are indistinguishable in any experiment based upon observation.

After establishing the congruence and developing its algebra in Sections 2-4, in Section 5 it is demonstrated that observation equivalence is characterised by a simple modal logic. Finally in Section 6 appears a brief study of the ideas of *confluence* and *determinacy* of processes.

The theory presented here is in contrast with that of Petri Nets; see [Rei] for example. The first difference is that the idea of causal independence (called concurrency by Net Theorists) is not here taken as fundamental; in our discipline of observation, causality is not directly observable and therefore the concurrent occurrence of two observable actions is not distinguished from their occurrence in arbitrary sequence. The second difference is that the combinational structure of process expressions, reflecting perhaps the physical structure of a machine which performs a process, plays a prominent part in our theory. The relative advantages of the two approaches are not discussed, and the discussion is indeed difficult; the aim here is to pursue the algebraic theory in its own terms as far as space permits. To this end, we give only small illustrative examples, rather than case studies in application which can be found elsewhere.

There are many different approaches to the algebra of processes. The differences are in the constructions employed and in the notion of equivalence. The author owes much to the ideas of, and discussions with, those who have proposed

different algebras, including Tony Hoare, Matthew Hennessy, Jan Bergstra and Jan Willem Klop, Gerard Berry, and their colleagues. It is another hard task to compare all these approaches and the task is not attempted here, though Section 7 outlines some of the issues. In choosing to pursue one approach in depth here the author does not wish to imply that other methods or definitions may not be equally appropriate.

2. The basic language

We begin by defining the *expressions* of our process language. Having done this we shall present its operational semantics as a *Labelled Transition System*. This leads to the notion of a *derivation tree*, which records the successive transitions or actions which may be performed by a given process. We introduce the notion of *sort*, which provides a convenient means of classifying processes; a sort is a set of labels, and we attribute a sort L to a process if L contains all the labels which occur in the derivation tree of the process. This leads to the notion of *flow graph*, a means of depicting the structure of a process as composed from sub-processes which perform concurrently and may communicate with one another. We end the section with a richer language, derived from the basic language, in which the passage of data values between processes can be conveniently expressed.

2.1 Syntax and notation

We presuppose an infinite set $\mathcal{A}$ of *names* $a, b, c, \ldots$. Then $\overline{\mathcal{A}}$ is the set of *co-names* $\overline{a}, \overline{b}, \overline{c}, \ldots$; $\mathcal{A}$ and $\overline{\mathcal{A}}$ are disjoint and are in bijection via $\bar{}$; we declare $\bar{\bar{a}} = a$. $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$ is the set of *labels*; we shall use ℓ, ℓ' to range over $\mathcal{L}$, and K, L to range over subsets of $\mathcal{L}$, and we define $\overline{L} = \{\overline{\ell} : \ell \in L\}$. Labels will identify the actions which may be performed by a process. We also introduce a distinguished *silent action* $\tau \notin \mathcal{L}$; this special action is considered to be unobservable. We set $Act = \mathcal{L} \cup \{\tau\}$, and we shall use α, β to range over Act.

A function $f : \mathcal{L} \to \mathcal{L}$ is called a *relabelling function* provided $f(\overline{\ell}) = \overline{f(\ell)}$; we extend f to Act by decreeing that $f(\tau) = \tau$. We shall often write f in the form $(\ell'_1/\ell_1, \ldots, \ell'_n/\ell_n)$ when $f(\ell_i) = \ell'_i$, $f(\overline{\ell_i}) = \overline{\ell'_i}$ and otherwise f is the identity function; here the names of ℓ_i are distinct.

We presuppose a countable set $\mathcal{X}$ of *process variables* $X, Y, \ldots$. We shall often use I to stand for an arbitrary (possibly infinite) indexing set. Then the set $\mathcal{E}$ of *process expressions* $E, F, \ldots$ (also called terms) is the smallest set including $\mathcal{X}$ and the following expressions – when E, E_i are already in $\mathcal{E}$:

$$
\begin{array}{rll}
\alpha.E, & \text{an } \textit{Action} & (\alpha \in \textit{Act}) \\
\Sigma_{i\in I}E_i, & \text{a } \textit{Summation} & \\
E_0|E_1, & \text{a } \textit{Composition} & \\
E\backslash L, & \text{a } \textit{Restriction} & (L \subseteq \mathcal{L}) \\
E[f], & \text{a } \textit{Relabelling} & (f \text{ a relabelling function}) \\
\mathrm{FIX}_j\{X_i = E_i : \ i \in I\}, & \text{a } \textit{Recursion} & (j \in I)
\end{array}
$$

In the final form (Recursion) the variables X_i are *bound* variables. For any expression E, its *free* variables $fv(E)$ are those which occur unbound in E. We say E is *closed* if $fv(E) = \emptyset$; in this case we say E is a *process*, and we shall use P, Q, R to range over the processes $\mathcal{P}$.

Roughly, the meaning of the process constructions is as follows. $\alpha.P$ is the process which performs α and then behaves as P; $\Sigma_{i\in I}P_i$ is the process which may behave as any of P_i; $P_0|P_1$ represents P_0 and P_1 performing concurrently, with possible communication; $P\backslash L$ behaves like P but with actions in $L \cup \overline{L}$ prohibited; $P[f]$ behaves like P but with the actions relabelled by f; finally, $\mathrm{FIX}_j\{X_i = E_i : i \in I\}$ is the j^{th} component of a distinguished "solution" of the recursive process equations $X_i = E_i \ \ (i \in I)$.

We shall usually be concerned only with finite Summations $\Sigma_{i\in I}P_i$, and indeed confine ourselves to the cases $I = \emptyset$ and $I = \{0,1\}$. When $I = \emptyset$ we write $\mathbf{0}$, which we call *Inaction*; it is the process which can do nothing. When $I = \{0,1\}$ we write $P_0 + P_1$.

We shall often write $\tilde{E}$ for $\{E_i : i \in I\}$ when I is understood, and write $\tilde{X} = \tilde{E}$ for an I-indexed set of equations. Other abbreviations are $\Sigma\tilde{E}$ for a Summation and $\mathrm{FIX}_j\tilde{X}\tilde{E}$ for a Recursion. Also $\mathrm{FIX}\,\tilde{X}\tilde{E}$ means $\{\mathrm{FIX}_j\tilde{X}\tilde{E} : j \in I\}$. For the simultaneous substitution of E_i for free occurrences of X_i in E we write $E\{\tilde{E}/\tilde{X}\}$, assuming that bound variables are changed where necessary to avoid clashes.

2.2 Operational Semantics

A *Labelled Transition System* (LTS) is of the form $(S, A, \{\xrightarrow{a} : a \in A\})$ where S is a set of *states*, A is a set of *actions* and each $\xrightarrow{a}$ is a subset of $S \times S$, called an *action relation* over S. We call $s \xrightarrow{a} s'$ an *action* of $s \in S$; in this case (a, s') is a *successor* of s and s' is an *a-successor* (or just *successor*) of s. For $t = a_1 \ldots a_n \in A^*$ we define $\xrightarrow{t} = \xrightarrow{a_1} \ldots \xrightarrow{a_n}$, and we call $s \xrightarrow{t} s'$ a *transition* of s; then (t, s') is a *derivative* of s and s' is a *t-derivative* (or just *derivative*) of s.

We shall define the LTS $(\mathcal{E}, \mathit{Act}, \{\xrightarrow{\alpha} : \alpha \in \mathit{Act}\})$. The action relations $\xrightarrow{\alpha}$ are defined by induction upon the structure of expressions; thus, for example, the

actions of a Composition $P|Q$ are inferred only from the actions of P and of Q. The action relations are the smallest which obey the following rules, in which the action below the line is to be inferred from those above the line:

$$\mathrm{ACT}: \frac{}{\alpha.E \xrightarrow{\alpha} E} \qquad \mathrm{SUM}_j: \frac{E_j \xrightarrow{\alpha} E'_j}{\Sigma_{i\in I} E_i \xrightarrow{\alpha} E'_j}(j \in I)$$

$$\mathrm{COM}_0: \frac{E_0 \xrightarrow{\alpha} E'_0}{E_0|E_1 \xrightarrow{\alpha} E'_0|E_1} \qquad \mathrm{COM}_1: \frac{E_1 \xrightarrow{\alpha} E'_1}{E_0|E_1 \xrightarrow{\alpha} E_0|E'_1}$$

$$\mathrm{COM}_2: \frac{E_0 \xrightarrow{\ell} E'_0 \qquad E_1 \xrightarrow{\bar{\ell}} E'_1}{E_0|E_1 \xrightarrow{\tau} E'_0|E'_1}$$

$$\mathrm{RES}: \frac{E \xrightarrow{\alpha} E'}{E\backslash L \xrightarrow{\alpha} E'\backslash L}(\alpha \notin L \cup \bar{L}) \qquad \mathrm{REL}: \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$$

$$\mathrm{REC}_j: \frac{E_j\{\mathrm{FIX}\,\tilde{X}\tilde{E}/\tilde{X}\} \xrightarrow{\alpha} E'_j}{\mathrm{FIX}_j\tilde{X}\tilde{E} \xrightarrow{\alpha} E'_j}(j \in I)$$

In most cases these rules are easily seen to realise the rough description of the process constructions given in 2.1. For Summation, note that the first action of $\Sigma_{i\in I}E_i$ determines which alternative E_j is selected, the others being discarded. For Composition, the rules COM_0 and COM_1 permit the concurrent performance of E_0 and E_1; the rule COM_2 permits a synchronising communication between E_0 and E_1 whenever they may perform complementary actions.

The rule REC_j states that the actions of (the j^{th} component of) the distinguished "solution" of $\tilde{X} = \tilde{E}$ are just those inferable by unwinding the Recursion. In effect, this rule *defines* the distinguished solution, though as we see later there are other solutions in general.

For simplicity we sometimes use an alternative formulation of Recursion. Instead of the FIX construction we introduce *process constants* $A, B, \ldots$ into the language. We then admit a set $\tilde{A} \stackrel{\mathrm{def}}{=} \tilde{P}$ of *defining equations* for the constants; of course $\tilde{P}$ may also contain the constants $\tilde{A}$. Then in place of the rules REC_j we have a simple rule

$$\mathrm{REC}': \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \text{ whenever } A \stackrel{\mathrm{def}}{=} P$$

We shall often need to prove properties with the logical form "if $E \xrightarrow{\alpha} E'$ then ..." . In the absence of Recursion such properties can be proved by induction on

the structure of E, but this is infeasible in general because, with either Recursion rule, an action for an expression is inferred from an action of a "larger" expression. So we use instead a stronger form of induction, namely upon the depth of the inference by which $E \xrightarrow{\alpha} E'$ is inferred. We call this *action induction*; it is a sound principle just because inference trees – i.e. compound inferences via the rules – are well-founded (in fact finite).

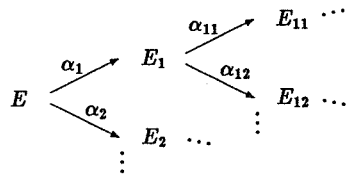
In what follows we shall omit many parentheses in process expressions, by declaring that Restriction and Relabelling have strongest binding power, then Action, then Composition, then Summation. Thus for example

$$P|\tau.Q\backslash a + R[f] \text{ means } (P|(\tau.(Q\backslash a))) + (R[f])$$

This also illustrates that $Q\backslash \ell$ means $Q\backslash\{\ell\}$.

2.3 Derivation trees and Transition graphs

A *derivation tree* (of E) is of the form



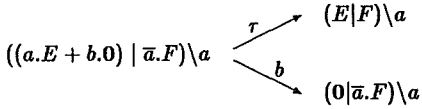
where the outgoing arcs from each non-leaf node are all the actions of the expression at that node. The tree can be infinite; it is *total* if the expressions at leaf nodes have no actions, otherwise it is *partial*.

The following shows that the derivatives of a process are themselves processes, so that the derivation tree of a process contains no free variables.

2.3.1 Proposition If $E \xrightarrow{\alpha} E'$ and $fv(E) = \emptyset$ then $fv(E') = \emptyset$.

Proof An easy action induction. □

2.3.2 Example A simple partial derivation tree is the following:

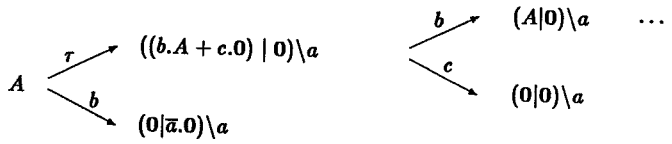


It may be checked that all actions of the root expression are present, but to extend the tree E and F must be made explicit. □

2.3.3 Example Now consider the Recursive definition

$$A \stackrel{\text{def}}{=} ((a.(b.A + c.0) + b.0) \mid \overline{a}.0)\backslash a$$

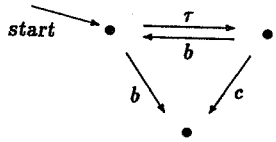
whose right-hand side arises from setting $E \equiv b.A + c.0, F \equiv 0$ in Example 2.3.2. Then, using REC′, the (infinite) total derivation tree of A can be developed:



Now it will follow from our theory of equality of processes that

$$\begin{aligned}(A|0)\backslash a &= A \\ (0|\overline{a}.0)\backslash a &= (\overline{a}.0)\backslash a = 0\end{aligned}$$

and hence the behaviour of A can be represented by the transition graph



Moreover, this transition graph also unfolds into the derivation tree of B, where

$$B \stackrel{\text{def}}{=} \tau.(b.B + c.0) + b.0$$

We shall not treat transition graphs formally. We mention them here for comparison with the classical theory of automata, where the meaning of an automaton (transition graph) is the set of finite strings which it accepts. By contrast, we pay more attention to the branching structure of derivation trees.

2.4 Sorts

2.4.1 Definition *P has sort $L \subseteq \mathcal{L}$*, or $P : L$, if every label ℓ in the total derivation tree of P is a member of L. □

For example, $A : \{b, c\}$ in Example 2.3.3. The essential property of sorts is the following simple consequence of the definition:

2.4.2.Proposition

(1) If $P : K$ and $K \subseteq L$ then $P : L$.

(2) $P : L$ iff, whenever $P \xrightarrow{\alpha} P'$, then $\alpha \in L \cup \{\tau\}$ and $P' : L$. □

It is also obvious that if $P : L_i$ for each $i \in I$ then $P : \bigcap_{i \in I} L_i$, so every process has a minimum sort. In general, this minimum sort cannot be effectively computed. However, sorts can be assigned to process expressions by simple syntactic rules; for this purpose we first assume that to each variable X and each constant A are assigned sorts $\mathcal{L}(X)$ and $\mathcal{L}(A)$.

2.4.3. Definition The (*distinguished*) *sort* $\mathcal{L}(E)$ of each E is given by the following rules:

$$\begin{array}{ll} \mathcal{L}(\alpha.E) = \mathcal{L}(E) \cup \{\alpha\} - \{\tau\} & \mathcal{L}(\Sigma_{i \in I} E_i) = \bigcup_{i \in I} \mathcal{L}(E_i) \\ \mathcal{L}(E_0 | E_1) = \mathcal{L}(E_0) \cup \mathcal{L}(E_1) & \mathcal{L}(E \backslash L) = \mathcal{L}(E) - (L \cup \overline{L}) \\ \mathcal{L}(E[f]) = f(\mathcal{L}(E)) & \mathcal{L}(\mathrm{FIX}_j \tilde{X} \tilde{E}) = \mathcal{L}(X_j) \end{array}$$

with the proviso that, in the last case, $\mathcal{L}(E_i) \subseteq \mathcal{L}(X_i)$ for each $i \in I$; also for each defining equation $A \stackrel{\text{def}}{=} P$ we require $\mathcal{L}(P) \subseteq \mathcal{L}(A)$. □

Note that $\mathcal{L}(E)$ is not always minimum; for example $(a.b.\mathbf{0})\backslash a : \emptyset$, while $\mathcal{L}((a.b.\mathbf{0})\backslash a) = \{b\}$. The following proposition supports the definition:

2.4.4 Proposition $P : \mathcal{L}(P)$.

Proof A simple action induction, using the characterisation of Prop 2.4.2(2).

□

The importance of sorts is that certain equations will hold only under an assumption about sorts; for example $P\backslash b = P$ if $b, \overline{b} \notin \mathcal{L}(P)$.

2.5 Flow graphs

We introduce flow graphs to represent the compositional structure of processes in which several sub-processes are performing concurrently; they record the *spatial* structure of processes, in contrast to derivation trees or transition graphs which record their *temporal* structure.

2.5.1. Definition A *flow graph* is a graph in which

1. Each *node* is labelled by a process (a generator);
2. Each node labelled P possesses a *port* with *internal label* ℓ, for each $\ell \in \mathcal{L}(P)$;

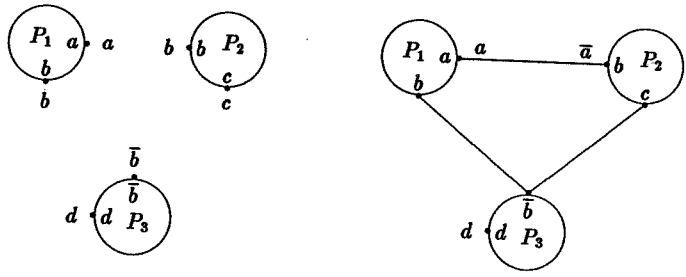
3. Each arc joins two ports of different nodes;

4. Each port may possess an *external label* $\in \mathcal{L}$;

5. Two ports joined by an arc either possess no external labels, or possess complementary external labels.

The *sort* $\mathcal{L}(G)$ of a flow graph G is its set of external port labels. □

2.5.2. Example Let $P_1 : \{a, b\}$, $P_2 : \{b, c\}$, $P_3 : \{\overline{b}, d\}$. Then the processes P_1, P_2 and P_3 by themselves, and the process $P \equiv ((P_1 \mid P_2[\overline{a}/b, b/c]) \mid P_3)\backslash b$, are depicted by the following four flow graphs:



Notice that the flow graph for P_1 by itself has internal and external labels equal. But the internal and external labels of a port may differ if a Relabelling is involved, and the external label may be absent if a Restriction is involved. Notice also that the sort $\mathcal{L}(G) = \{a, \overline{a}, d\}$ of the flow graph G depicting P is exactly the sort $\mathcal{L}(P)$ defined by 2.4.3. □

Flow graphs are indispensable as a tool for understanding complex parallel processes, though we shall not require them in our semantic treatment. They are formal objects, and indeed the operations of Composition, Restriction and Relabelling – which we call the *static constructors* – can be interpreted very simply over flow graphs in the following way:

- $G_0|G_1$ is obtained by joining each port in G_0 with external label ℓ to every port in G_1 with external label $\overline{\ell}$;
- $G\backslash L$ is obtained from G by deleting all external labels in $L \cup \overline{L}$;
- $G[f]$ is obtained from G by replacing each external label ℓ in G by $f(\ell)$.

It can be checked that G in Example 2.5.2 is built in this way. Many identities hold in this algebra of flow graphs. Indeed, the *static laws* (involving the static constructors), which we prove in the next section for processes, also hold in flow graph algebra. Further, it can even be shown – at least under the condition that the relabelling functions are injective when restricted to the sorts of their operands – that the static laws are *complete* for flow graph algebra. Thus flow graphs are the free algebra for the static laws; our process semantics can therefore be regarded as the unique morphism from process algebra to flow graph algebra induced by the depiction of the generators.

2.6 The extended language

For many applications it is important to represent the passage of data values between processes. So first assume that V is the set of data values (we do not trouble to consider different types of value), and that there are *value variables* $x, y, \ldots$ and *value expressions* e built from constants and standard functions (e.g. arithmetic or boolean operations) over V.

To represent input of an arbitrary value at a port with name a, we introduce the construction

$$ax.E$$

in which x is a *bound* value variable. We now consider a not as a single name, but standing for a family of names $\{a_x : x \in V\}$. Similarly we introduce the construction

$$\overline{a}e.E$$

to represent output of the value of e at the port with co-name $\overline{a}$; again $\overline{a}$ now stands for the family $\{\overline{a}_x : x \in V\}$. Next, we allow V-indexed families of process constants, $\{A_x : x \in V\}$, and introduce defining equations of the form

$$A(x) \stackrel{\text{def}}{=} P$$

where x will normally occur free in P; this admits processes which are parametric upon V. Finally, it is convenient to allow a conditional construction

$$\texttt{if } e \texttt{ then } E_{\text{true}} \texttt{ else } E_{\text{false}}$$

2.6.1. Example Consider a process $A : \{a, \overline{b}\}$ which alternately inputs an integer v at a, and outputs at $\overline{b}$ the least even integer $\geq v$. A can be defined by:

$$\begin{aligned} A &\stackrel{\text{def}}{=} ax.A'(x) \\ A'(x) &\stackrel{\text{def}}{=} \texttt{if even}(x) \texttt{ then } \overline{b}x.A \texttt{ else } \overline{b}(x+1).A \end{aligned}$$

□

The translation of expressions E of the extended language into the basic language is defined by induction on the structure of expressions. The translation $\widehat{E}$ of E, for the four new constructions, is as follows:

E	$\widehat{E}$
$ax.F$	$\Sigma_{x \in V} a_x.\widehat{F}$
$\overline{a}e.F$	$\overline{a}_e.\widehat{F}$
if e **then** $E_{\mathbf{true}}$ **else** $E_{\mathbf{false}}$	$\Sigma_{x \in \{e\}} \widehat{E_x}$
$A(e)$	A_e

(translation distributes over the other constructions). Thus the translation of Example 2.6.1 is given by

$$A \stackrel{\text{def}}{=} \Sigma_{x \in V} a_x.A'_x$$

$$A'_x \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} \overline{b}_x.A & (\text{if } x \text{ is even}) \\ \overline{b}_{x+1}.A & (\text{if } x \text{ is odd}) \end{array} \right\} \text{ for each } x \in V$$

The translation is possible just because, in the basic language, both Summations and Recursions have been indexed by arbitrary sets I. Most of our theory applies to this infinitary language; it is only when we consider complete axioms for finite-state processes that we wish to confine ourselves to the finitary language in which the indexing sets are kept finite.

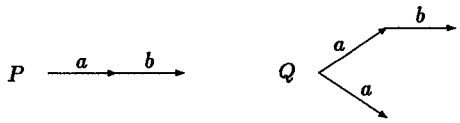
3. Strong congruence of processes

3.1 Discussion

In this section we seek an appropriate equivalence relation over processes, which gives no special status to the silent τ-action. The following section will present a weaker (larger) equivalence which reflects the idea that the τ-action should indeed be silent, i.e. unobservable; but for the stronger equivalence of this section we shall already be able to derive several useful algebraic laws.

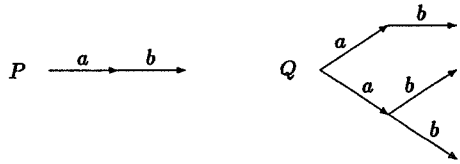
Perhaps the most obvious equivalence of processes is one which requires merely that they should possess the same transitions. More exactly, we might declare P and Q to be equivalent just when, for all $t \in Act^*$, $P \xrightarrow{t}$ iff $Q \xrightarrow{t}$; here we use $P \xrightarrow{t}$ to mean that $P \xrightarrow{t} P'$ for some P'.

But consideration of deadlock leads us to reject this proposal. For P and Q would be equivalent if they have the following total derivation trees:



But in this case, after performing a, P will always be able to perform b while Q may not. Thus, in an "environment" which *demands* b after a, P will not deadlock while Q may. So apparently our proposed equivalence is too large.

On the other hand we may be *too* restrictive; we may take P and Q to be equivalent just when their derivation trees are isomorphic. This would deny the equivalence of P and Q with trees



even though, at each stage, the same actions are possible.

We therefore seek an intermediate notion, with the following property:

> P and Q are equivalent iff, for all $\alpha \in Act$, each α-successor of P is equivalent to some α-successor of Q, and conversely. (*)

We shall see that such an equivalence exists; moreover it is indeed a congruence, and has many useful algebraic properties.

3.2 Strong bisimulation

(i) Whenever $P \xrightarrow{\alpha} P'$ then for some Q', $Q \xrightarrow{\alpha} Q'$ and $\langle P', Q' \rangle \in \mathcal{S}$;
(ii) Whenever $Q \xrightarrow{\alpha} Q'$ then for some P', $P \xrightarrow{\alpha} P'$ and $\langle P', Q' \rangle \in \mathcal{S}$. □

3.2.1 Definition Let $\mathcal{F}$ be the function over binary relations $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ defined as follows: $\langle P, Q \rangle \in \mathcal{F}(\mathcal{S})$ iff, for all $\alpha \in Act$,

(i) Whenever $P \xrightarrow{\alpha} P'$ then for some Q', $Q \xrightarrow{\alpha} Q'$ and $\langle P', Q' \rangle \in \mathcal{S}$;
(ii) Whenever $Q \xrightarrow{\alpha} Q'$ then for some P', $P \xrightarrow{\alpha} P'$ and $\langle P', Q' \rangle \in \mathcal{S}$. □

It is easy to see that an equivalence relation $\sim$ satisfies $(*)$ iff it is a fixed-point of $\mathcal{F}$ – i.e. $\sim = \mathcal{F}(\sim)$. But we must ensure that such a relation exists.

3.2.2 Definition $S \subseteq P \times P$ is a *strong bisimulation* if $S \subseteq \mathcal{F}(S)$. □

3.2.3 Proposition $\mathcal{F}$ is monotone; that is, $S \subseteq S'$ implies $\mathcal{F}(S) \subseteq \mathcal{F}(S')$. □

3.2.4 Definition Converse and composition of relations are defined as follows:

$$\begin{aligned} S^{-1} &= \{\langle Q, P\rangle : PSQ\} \\ SS' &= \{\langle P, R\rangle : \text{for some } Q,\ PSQ \text{ and } QS'R\} \end{aligned}$$

□

3.2.5 Proposition If S_i $(i \in I)$ are strong bisimulations, then so is $\bigcup_{i\in I} S_i$.

Proof An easy application of 3.2.3. □

3.2.6 Proposition If S and S' are strong bisimulations, then so are Id_P, S^{-1} and SS'.

Proof Immediate from 3.2.1. □

We are now ready to define $\sim$, our *strong congruence* relation.

3.2.7 Definition $\sim = \bigcup\{S : S \text{ is a strong bisimulation}\}$. □

3.2.8 Proposition $\sim$ is the largest strong bisimulation, and is also an equivalence relation.

Proof The first part follows from 3.2.5; for the second part, reflexivity, symmetry and transitivity follow from 3.2.6. □

Now we can show that $\sim$ is the largest equivalence satisfying $(*)$.

3.2.9 Proposition $\sim = \mathcal{F}(\sim)$, and $\sim$ is the largest fixed point of $\mathcal{F}$.

Proof From 3.2.8 we have $\sim \subseteq \mathcal{F}(\sim)$. Hence $\mathcal{F}(\sim) \subseteq \mathcal{F}(\mathcal{F}(\sim))$ by monotonicity, so $\mathcal{F}(\sim)$ is a strong bisimulation, and by 3.2.8 is included in $\sim$; hence $\sim = \mathcal{F}(\sim)$. But all fixed points of $\mathcal{F}$ are strong bisimulations, so the largest strong bisimulation – being a fixed point – is also the largest fixed point. □

The rest of this section is organised as follows. First, we shall derive several equational laws for $\sim$. The first group consists of simple properties of Summation; the second group consists of the static laws mentioned in Section 2; the third group is a single theorem, the *Expansion Theorem*, which relates the *static constructors* (Composition, Restriction and Relabelling) to the *dynamic constructors* (Action

and Summation), and which yields several simple corollaries.

Next we introduce the notion of strong bisimulation *up to* $\sim$, which provides a useful proof technique. We also extend $\sim$ from processes to process expressions. This allows us to discharge our obligation to prove that $\sim$ is preserved by all the process constructors; that is, that it *is* a congruence.

Finally, we show that under a simple condition a set of equations $\tilde{X} \sim \tilde{E}$ possesses a unique solution up to strong congruence.

3.3 Equational properties of strong congruence

Our first result is simple, namely that $(P/\sim, +, \mathbf{0})$ is a monoid with absorption.

3.3.1 Proposition (monoid laws)

(1) $P + Q \sim Q + P$ (2) $P + (Q + R) \sim (P + Q) + R$
(3) $P + P \sim P$ (4) $P + \mathbf{0} \sim P$

Proof Consider (1) only (the others are similar). Let $P + Q \xrightarrow{\alpha} R$; then this action must be inferred either from $P \xrightarrow{\alpha} R$ by SUM_0 or from $Q \xrightarrow{\alpha} R$ by SUM_1 ; in either case the opposite rule yields also $Q + P \xrightarrow{\alpha} R$. Hence both sides have identical α-successors, so $P + Q \sim Q + P$. □

For strong congruence, we find no other laws satisfied by the dynamic constructors Action and Summation. One possibility would be the distributive law $\alpha.(P + Q) \sim \alpha.P + \alpha.Q$; however, we have implicitly rejected this law. For if it were true then with the help of 3.3.1(4) we could deduce $a.b.\mathbf{0} \sim a.b.\mathbf{0} + a.\mathbf{0}$, and we have rejected this equivalence in 3.1 because of deadlock considerations. In fact the reader can easily check that property $(*)$ denies this equivalence.

However, when we weaken the equivalence in the next section to allow for the unobservability of τ, then we shall find a few laws for the τ-action.

We now proceed to deduce the static laws for Composition, Restriction and Relabelling. We call these the static constructors because they are preserved by transition; for example if $(P|Q)\backslash L \xrightarrow{\alpha} R$ then R must be of the form $(P'|Q')\backslash L$. It follows that these constructors represent spatial rather than temporal structure, so it is natural to find that they can be interpreted over flow graphs.

The static laws depend sometimes on a sort condition; we consider an example. In general we do not have $(P|Q)\backslash a \sim P\backslash a \mid Q\backslash a$. The reader may check that this fails in the flow graph interpretation, when for example $P : \{a, b\}$ and $Q : \{\overline{a}\}$. It also fails in the process interpretation; this can be seen by taking $P \equiv a.b.\mathbf{0}$ and $Q \equiv \overline{a}.\mathbf{0}$, for then the left-hand side is equivalent to $\tau.b.\mathbf{0}$ and the right-hand side to $\mathbf{0}$. Intuitively a Restriction may only be distributed over a Composition when it does not affect a possible "communication link" between the components; this

is reflected in the side condition on (7) below.

3.3.2 Proposition (Static Laws)

(1) $P|Q \sim Q|P$

(2) $P|(Q|R) \sim (P|Q)|R$

(3) $P|\mathbf{0} \sim P$

(4) $P\backslash L \sim P$ if $\mathcal{L}(P) \cap (L \cup \overline{L}) = \emptyset$

(5) $P\backslash K\backslash L \sim P\backslash(K \cup L)$

(6) $P[f]\backslash L \sim P\backslash f^{-1}(L)[f]$

(7) $(P|Q)\backslash L \sim P\backslash L \mid Q\backslash L$ if $\mathcal{L}(P) \cap \overline{\mathcal{L}(Q)} \cap (L \cup \overline{L}) = \emptyset$

(8) $P[\mathrm{Id}_{\mathcal{L}}] \sim P$

(9) $P[f] \sim P[f']$ if $f\restriction\mathcal{L}(P) \sim f'\restriction\mathcal{L}(P)$

(10) $P[f][f'] \sim P[f' \circ f]$

(11) $(P|Q)[f] \sim P[f] \mid Q[f]$ if $f\restriction(L \cup \overline{L})$ is injective, where $L = \mathcal{L}(P|Q)$

Proof All the laws may be proved by exhibiting appropriate strong bisimulations. For (1), for example, we must prove that

$$S = \{\langle P_1|P_2, P_2|P_1\rangle \quad : \quad P_1, P_2 \in P\}$$

is a strong bisimulation. So let $P_1|P_2 \xrightarrow{\alpha} Q$; it is enough to find R such that $P_2|P_1 \xrightarrow{\alpha} R$ and $\langle Q, R\rangle \in S$ (with a symmetric argument). But $P_1|P_2 \xrightarrow{\alpha} Q$ must be inferred from COM_0, COM_1 or COM_2, so $Q \equiv P_1'|P_2'$ where *either* $P_1 \xrightarrow{\alpha} P_1'$ and $P_2 \equiv P_2'$, *or* $P_1 \equiv P_1'$ and $P_2 \xrightarrow{\alpha} P_2'$, *or* $\alpha = \tau$ and $P_1 \xrightarrow{\ell} P_1'$ and $P_2 \xrightarrow{\overline{\ell}} P_2'$. In each case we have $P_2|P_1 \xrightarrow{\alpha} R \equiv P_2'|P_1'$, and $\langle Q, R\rangle \in S$ as required.

Proofs of the other laws differ only in detail, not in principle. □

Some of the static laws are more readily understood in terms of simple special cases. Often, for example, a relabelling function $f = (\ell_1'/\ell_1, \ldots, \ell_n'/\ell_n)$ has ℓ_i and ℓ_i' all distinct, and in this case $f = (\ell_n'/\ell_n) \circ \cdots \circ (\ell_1'/\ell_1)$; hence $P[f] = P[\ell_1'/\ell_1]\cdots[\ell_n'/\ell_n]$ by 3.3.2(10). So it is often enough to deal with the simple relabelling functions of the form ℓ'/ℓ. A few simple consequences of 3.3.2 are as follows:

3.3.3 Corollary

(1) $P[\ell'/\ell] \sim P$ if $\ell, \overline{\ell} \notin \mathcal{L}(P)$

(2) $P\backslash\ell \sim P[\ell'/\ell]\backslash\ell'$ if $\ell', \overline{\ell'} \notin \mathcal{L}(P)$

(3) $P\backslash L[\ell'/\ell] \sim P[\ell'/\ell]\backslash L$ if $\ell, \ell' \notin L \cup \overline{L}$ □

These consequences show that Restriction and Relabelling behave very much like binding and syntactic substitution. But Relabelling is a constructor (operator) of the calculus, not a meta-syntactic operation; it could not be treated as the latter, since - as 3.3.2(7) shows - it does not always distribute over Composition, but only under a side-condition on sorts.

In many applications, a set of processes $P_1, \ldots, P_n$ are connected in some way represented by a construction $C[P_1, \ldots, P_n]$, where $C[\ ]$ involves just the static constructors. So this construction can be depicted as a flowgraph with $P_1, \ldots, P_n$ labelling the nodes. For semantic analysis it turns out very often (in fact, whenever Relabelling is only applied injectively) that static constructions $C[P_1, \ldots, P_n]$ can be transformed by the static laws into the form

$$(P_1[f_1] \mid \cdots \mid P_n[f_n])\backslash L$$

which we call a *standard concurrent form*. The next theorem indicates how the successors of such a form are derived from the successors of its components P_i . In fact, the theorem – and its corollaries – cover everything that we normally need to know about the interaction between the static and dynamic constructions.

3.3.4 Proposition (the Expansion theorem)

Let $P \equiv (P_1[f_1] \mid \cdots \mid P_n[f_n])\backslash L$, $n \geq 1$. Then

$$\begin{aligned} P \ \sim \ & \Sigma\{f_i(\alpha).(P_1[f_1] \mid \cdots \mid P_i'[f_i] \mid \cdots \mid P_n[f_n])\backslash L : \\ & \qquad P_i \xrightarrow{\alpha} P_i',\ i \leq n,\ f_i(\alpha) \notin L \cup \overline{L}\} \\ + \ & \Sigma\{\tau.(P_1[f_1] \mid \cdots \mid P_i'[f_i] \mid \cdots \mid P_j'[f_j] \mid \cdots \mid P_n[f_n])\backslash L : \\ & \qquad P_i \xrightarrow{\ell_1} P_i',\ P_j \xrightarrow{\ell_2} P_j',\ i < j \leq n,\ f_i(\ell_1) = \overline{f_j(\ell_2)}\} \end{aligned}$$

Remark. The first Summation corresponds to actions performed by a single component of P, while the second Summation corresponds to communications performed by pairs of components.

Proof (outline) It is quite easy to prove first, by induction upon n, that if $Q \equiv Q_1 \mid \cdots \mid Q_n$ then

$$\begin{aligned} Q \ \sim \ & \Sigma\{\alpha.(Q_1 \mid \cdots \mid Q_i' \mid \cdots \mid Q_n) : \ i \leq n,\ Q_i \xrightarrow{\alpha} Q_i'\} \\ + \ & \Sigma\{\tau.(Q_1 \mid \cdots \mid Q_i' \mid \cdots \mid Q_j' \mid \cdots \mid Q_n) : \ i < j \leq n,\ Q_i \xrightarrow{\ell} Q_i',\ Q_j \xrightarrow{\overline{\ell}} Q_j'\} \end{aligned}$$

The general case is then obtained by setting $Q_i \equiv P_i[f_i]$ and applying the Restriction $\backslash L$ to both sides. □

Several useful corollaries of this theorem arise from setting $n = 1$, and either $S_i = \mathrm{Id}_{\mathcal{L}}$ or $L = \emptyset$. They show how Restriction and Relabelling "distribute" over Action and Summation.

3.3.5 Corollary

$$(1)\ (\alpha.P)\backslash L \sim \begin{cases} \mathbf{0} \text{ if } \alpha \in L \cup \overline{L} \\ \alpha.P\backslash L \text{ otherwise} \end{cases} \quad (2)\ (\alpha.P)[f] \sim f(\alpha).P[f]$$

$$(3)\ (P+Q)\backslash L \sim P\backslash L + Q\backslash L \qquad (4)\ (P+Q)[f] \sim P[f] + Q[f] \quad \square$$

3.4 Substitutivity of strong congruence

We must now demonstrate that $\sim$ is indeed a congruence relation, i.e. that it is substitutive everywhere. First, we refine the proof technique of strong bisimulation a little. We know that to show $P \sim Q$ it is enough to show $P\mathcal{S}Q$ for some strong bisimulation $\mathcal{S}$. But for this purpose, we need only a slightly weaker property of $\mathcal{S}$.

3.4.1 Definition $\mathcal{S}$ is a *strong bisimulation up to* $\sim$ if $\sim \mathcal{S} \sim$ is a strong bisimulation. □

3.4.2 Proposition If $\mathcal{S}$ is a strong bisimulation up to $\sim$ then $\mathcal{S} \subseteq \sim$.

Proof Let $P\mathcal{S}Q$. Then clearly $P \sim \mathcal{S} \sim Q$, so $P \sim Q$ since $\sim \mathcal{S} \sim \subseteq \sim$. □

The next result shows that a property of $\mathcal{S}$ slightly weaker than $\mathcal{S} \subseteq \mathcal{F}(S)$ is enough to show that $\mathcal{S}$ is a strong bisimulation up to $\sim$ and hence that $\mathcal{S} \subseteq \sim$.

3.4.3 Proposition If $\mathcal{S} \subseteq \mathcal{F}(\sim \mathcal{S} \sim)$ then $\mathcal{S} \subseteq \sim$

Proof It is enough to show that $\sim \mathcal{S} \sim \subseteq \mathcal{F}(\sim \mathcal{S} \sim)$ from the hypothesis. For this purpose, let $P \sim \mathcal{S} \sim Q$, i.e. $P \sim P_1\ \mathcal{S}\ Q_1 \sim Q$. Let $P \xrightarrow{\alpha} P'$; we wish to find Q' so that $Q \xrightarrow{\alpha} Q'$ and $P' \sim \mathcal{S} \sim Q'$. This is done by filling in the bottom row of the following diagram from left to right, starting with the given P':

$$\begin{array}{ccc} P \sim P_1 & \mathcal{S} & Q_1 \sim Q \\ \alpha\downarrow \quad \alpha\downarrow & & \downarrow\alpha \quad \downarrow\alpha \\ P' \sim P_1' & \sim\mathcal{S}\sim & Q_1' \sim Q' \end{array}$$

This – with a symmetric argument – complete the proof. □

The first proposition about substitutivity is the following.

3.4.4 Proposition Let $P_1 \sim P_2$. Then

(1) $\alpha.P_1 \sim \alpha.P_2$ (2) $P_1 + Q \sim P_2 + Q$ (3) $P_1|Q \sim P_2|Q$
(4) $P_1\backslash L \sim P_2\backslash L$ (5) $P_1[f] \sim P_2[f]$

Proof (1) and (2) are easily established by working directly with property (*) in 3.1. For (3), it is enough to show that

$$\mathcal{S} = \{\langle P_1|Q, P_2|Q\rangle \ : \ P_1 \sim P_2\}$$

is a strong bisimulation. The details are routine, and (4) and (5) can be established by a similar method. □

This tells us that $\sim$ is substitutive everywhere except under Recursion. For the latter, we first extend $\sim$ to expressions.

3.4.5 Definition Let E and F have free variables $\tilde{X}$. Then $E \sim F$ if, for all $\tilde{P}$, $E\{\tilde{P}/\tilde{X}\} \sim F\{\tilde{P}/\tilde{X}\}$. □

3.4.6 Proposition Let $\tilde{E} \sim \tilde{F}$. Then $\mathrm{FIX}\tilde{X}\tilde{E} \sim \mathrm{FIX}\tilde{X}\tilde{F}$.

Proof It will be enough to assume that $\tilde{E}, \tilde{F}$ have free variables $\tilde{X}$ (at most), since by use of 3.4.5 we can then extend the result to the general case in which $\mathrm{FIX}\tilde{X}\tilde{E}$ contains (other) free variables. Now let

$$\tilde{P} \equiv \mathrm{FIX}\tilde{X}\tilde{E}, \quad \tilde{Q} \equiv \mathrm{FIX}\tilde{X}\tilde{F}$$

Then it will be enough to show that

$$\mathcal{S} = \{\langle G\{\tilde{P}/\tilde{X}\}, G\{\tilde{Q}/\tilde{X}\}\rangle \ : \ fv(G) \subseteq \tilde{X}\}$$

is a strong bisimulation up to $\sim$: for the result follows by taking $G \equiv X_i$ for each i, where $\tilde{X} = \{X_i : i \in I\}$.

For this purpose, it is enough to show that if $G\{\tilde{P}/\tilde{X}\} \xrightarrow{\alpha} P'$ then, for some Q', $G\{\tilde{Q}/\tilde{X}\} \xrightarrow{\alpha} Q'$ and $P'\mathcal{S} \sim Q'$. This property can be established by action induction, by case analysis on the form of G. The interesting case is when $G \equiv X_i$; then we have $G\{\tilde{P}/\tilde{X}\} \equiv P_i \xrightarrow{\alpha} P'$, so $E_i\{\tilde{P}/\tilde{X}\} \xrightarrow{\alpha} P'$ by a shorter inference, so by induction $E_i\{\tilde{Q}/\tilde{X}\} \xrightarrow{\alpha} Q''$ with $P'\mathcal{S} \sim Q''$; hence because $E_i \sim F_i$, $F_i\{\tilde{Q}/\tilde{X}\} \xrightarrow{\alpha} Q'$ with $Q'' \sim Q'$; hence finally $G\{\tilde{Q}/\tilde{X}\} \equiv Q_i \xrightarrow{\alpha} Q'$ also, and $P'\mathcal{S} \sim Q'$. □

This completes our proof of the substitutivity of $\sim$, and allows us to treat it as an equality relation. It remains to see the behaviour of fixed points with respect to this relation.

3.5 Unique fixed points up to strong congruence

The first result about fixed points is that the distinguished "solution" of $\tilde{X} = \tilde{E}$ is indeed a solution, up to $\sim$.

3.5.1 Proposition $\mathrm{FIX}\tilde{X}\tilde{E} \sim \tilde{E}\{\mathrm{FIX}\tilde{X}\tilde{E}/\tilde{X}\}$.

Proof By REC, clearly $\mathrm{FIX}_j\tilde{X}\tilde{E}$ and $E_j\{\mathrm{FIX}\tilde{X}\tilde{E}/\tilde{X}\}$ have identical successors for each j. □

Now, in general, solutions of $\tilde{X} \sim \tilde{E}$ cannot be unique; the equations $\tilde{X} = \tilde{X}$ have *all* processes as solutions. But a simple constraint ensures uniqueness.

3.5.2 Definition X is *weakly guarded* in E if each free occurrence of X is within some subexpression $\alpha.F$ of E. E is *weakly guarded* if all variables are weakly guarded in E. □

Note that X is weakly guarded in $\tau.X$. In the next section we need a stronger notion of guardedness, which requires the guard to be a member of $\mathcal{L}$, to ensure uniqueness of fixed points for the weaker notion of equality developed there.

3.5.3 Proposition Let the variables $\tilde{X}$ be weakly guarded in $\tilde{E}$, and let $\tilde{F} \sim \tilde{E}\{\tilde{F}/\tilde{X}\}$. Then $\tilde{F} \sim \mathrm{FIX}\tilde{X}\tilde{E}$.

Proof Using definition 3.4.5, it will be enough to consider the case in which $fv(\tilde{E}) \subseteq \tilde{X}$, so that $\tilde{E}$ are weakly guarded expressions. Then it will be enough to prove that, if $\tilde{P} \sim \tilde{E}\{\tilde{P}/\tilde{X}\}$ and $\tilde{Q} \sim \tilde{E}\{\tilde{Q}/\tilde{X}\}$, then $\tilde{P} \sim \tilde{Q}$. For this, it will again be enough to show that

$$\mathcal{S} = \{\langle G\{\tilde{P}/\tilde{X}\}, G\{\tilde{Q}/\tilde{X}\}\rangle : fv(G) \subseteq \tilde{X}\} \cup \mathrm{Id}_{\mathcal{P}}$$

is a strong bisimulation up to $\sim$; the result follows by taking $G \equiv X_i$ for each i.

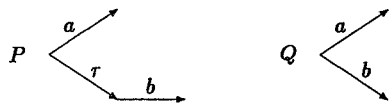
To prove this, we need to show that if $G\{\tilde{P}/\tilde{X}\} \xrightarrow{\alpha} P'$ then, for some Q', $G\{\tilde{Q}/\tilde{X}\} \xrightarrow{\alpha} Q'$ and $P' \sim S \sim Q'$. As in Prop 3.4.6, this can be established by action induction, and again the interesting case is when $G \equiv X_i$; the details are rather similar to 3.4.6, and we leave the reader to fill them in. □

The outcome of this section is that strong congruence provides a tractable notion of equality for processes, and allows many non-trivial equalities to be derived. But it is deficient in a vital respect; it treats τ on the same basis as all other actions, and properties which we would expect to hold if τ is unobservable, such as $\alpha.\tau.P = \alpha.P$, do not hold if "=" is taken to mean strong congruence. The next section, which follows the lines of the present section in many ways, will remove this defect by widening the notion of equality.

4. Observation Congruence of Processes

4.1 Observation equivalence

What does it mean for τ to be silent, or unobservable? A first answer might be that two processes should be equivalent if they become strongly congruent when the τ-actions are excised from their derivation trees. Under this proposal we would equate $P \equiv a.0 + \tau.b.0$ with $Q \equiv a.0 + b.0$:



But this leads to difficulty. Since τ is to be unobservable, which we take to mean uncontrollable by the environment, P can perform τ autonomously and thus forego its ability to perform a; Q however preserves this ability. So, in an environment which demands a, Q will not deadlock, while P may. So τ, though unobservable directly, can affect the observability of visible actions.

We shall capture this indirect effect by requiring that, for P and Q to be observation equivalent, each action by P must by matched by a sequence of actions by Q with the same visible content, and conversely. The following definitions make this precise.

4.1.1 Definition Let $t = \alpha_1 \cdots \alpha_n \in Act^*$. Then

(1) $\xrightarrow{t} \stackrel{\text{def}}{=} \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$

(2) $\hat{t} \in Act^*$ is the result of removing all τ's from t

(3) $\stackrel{\hat{t}}{\Rightarrow} \stackrel{\text{def}}{=} (\xrightarrow{\tau})^* \xrightarrow{\alpha_1} (\xrightarrow{\tau})^* \cdots (\xrightarrow{\tau})^* \xrightarrow{\alpha_n} (\xrightarrow{\tau})^*$ □

Note that $\hat{\tau} = \varepsilon$, the empty sequence, and $\stackrel{\hat{\tau}}{\Rightarrow} = (\xrightarrow{\tau})^*$. Also $\stackrel{\hat{t}}{\Rightarrow}$ represents all sequences of actions with the same visible content as $\xrightarrow{t}$. We shall write $\Rightarrow$ for $\stackrel{\hat{\varepsilon}}{\Rightarrow}$.

4.1.2 Definition For $t \in Act^*$, E' is a *t-descendant* of E if $E \stackrel{\hat{t}}{\Rightarrow} E'$. □

We therefore seek an equivalence with the following property:

> P and Q are equivalent iff, for all $\alpha \in Act$, each α-successor of P is equivalent to some α-descendant of Q, and conversely (*)

We arrive at this, which we call *observation equivalence*, in exactly the same way as we arrived at strong congruence.

4.2 Bisimulation

4.2.1 Definition Let $\mathcal{G}$ be the function over binary relations $S \subseteq P \times P$ defined as follows: $\langle P, Q \rangle \in \mathcal{G}(S)$ iff, for all $\alpha \in Act$,

(i) Whenever $P \xrightarrow{\alpha} P'$ then for some Q', $Q \stackrel{\hat{\alpha}}{\Rightarrow} Q'$ and $\langle P', Q' \rangle \in S$;
(ii) Whenever $Q \xrightarrow{\alpha} Q'$ then for some P', $P \stackrel{\hat{\alpha}}{\Rightarrow} P'$ and $\langle P', Q' \rangle \in S$. □

So observation equivalence will be a fixed point of $\mathcal{G}$.

4.2.2 Definition $S \subseteq P \times P$ is a (*weak*) *bisimulation* if $S \subseteq \mathcal{G}(S)$ □

4.2.3 Proposition $\mathcal{G}$ is monotone. □

4.2.4 Proposition If S_i $(i \in I)$ are bisimulations, so is $\cup_{i \in I} S_i$ □

4.2.5 Proposition If S, S' are bisimulations, so are Id_P, S^{-1} and $S\,S'$ □

We now define $\approx$, *observation equivalence* or *bisimilarity*:

4.2.6 Definition $\approx = \bigcup\{S : S \text{ is a bisimulation}\}$ □

4.2.7 Proposition $\approx$ is the largest bisimulation, and is an equivalence. □

4.2.8 Proposition $\approx = \mathcal{G}(\approx)$, and $\approx$ is the largest fixed point of $\mathcal{G}$. □

Thus $\approx$ is the largest relation satisfying $(*)$. To establish $P \approx Q$ it is clearly enough to find a bisimulation $\mathcal{S}$ such that $P\mathcal{S}Q$, since $\mathcal{S} \subseteq \approx$. This is the most frequently useful technique for showing $P \approx Q$; however, there is another characterisation of $\approx$ which is intuitively satisfying since it is expressed solely in terms of $\stackrel{s}{\Rightarrow}$ $(s \in \mathcal{L}^*)$ rather than in terms of $\stackrel{\alpha}{\rightarrow}$ and $\stackrel{\hat{\alpha}}{\Rightarrow}$:

4.2.9 Proposition $P \approx Q$ iff, for all $s \in \mathcal{L}^*$,

(i) Whenever $P \stackrel{s}{\Rightarrow} P'$ then, for some Q', $Q \stackrel{s}{\Rightarrow} Q'$ and $P' \approx Q'$;
(ii) Whenever $Q \stackrel{s}{\Rightarrow} Q'$ then, for some P', $P \stackrel{s}{\Rightarrow} P'$ and $P' \approx Q'$.

Moreover, $\approx$ is the largest relation with this property. □

4.2.10 Proposition $P \sim Q$ implies $P \approx Q$

Proof It follows directly from the definitions that every strong bisimulation is also a bisimulation. □

4.2.11 Proposition $P \approx \tau.P$

Proof It is easy to show that $\{\langle P, \tau.P\rangle\} \cup \mathrm{Id}_P$ is a bisimulation. □

This result shows us that observation equivalence is not a congruence. For it is not preserved by Summation; we have $b.\mathbf{0} \approx \tau.b.\mathbf{0}$, but if $a \neq b$ it is easily seen that $a.\mathbf{0} + b.\mathbf{0} \not\approx a.\mathbf{0} + \tau.b.\mathbf{0}$, as we would expect from the discussion at the beginning of 4.1. However, observation equivalence is preserved by the static constructors:

4.2.12 Proposition If $P \approx Q$ then $P|R \approx Q|R$, $P\backslash L \approx Q\backslash L$ and $P[f] \approx Q[f]$

Proof For composition, it is enough to show that $\{\langle P|R, Q|R\rangle : P \approx Q,\ R \in P\}$ is a bisimulation; a similar argument also works in the other cases. □

Later we shall see that the Action constructor even strengthens the relation.

For further work, we also need to deal with the more general notion of bisimulation *up to* $\approx$, analogous to strong bisimulation up to $\sim$.

4.2.13 Definition $\mathcal{S}$ is a *bisimulation up to* $\approx$ if $\approx \mathcal{S} \approx$ is a bisimulation. □

4.2.14 Proposition If $\mathcal{S}$ is a bisimulation up to $\approx$ then $\mathcal{S} \subseteq \approx$ □

4.2.15 Proposition If $\mathcal{S} \subseteq \mathcal{G}(\approx \mathcal{S} \approx)$ then $\mathcal{S} \subseteq \approx$ □

Finally we extend $\approx$ to expressions in the obvious way.

4.2.16 Definition Let $fv(E), fv(F) \subseteq \tilde{X}$. Then $E \approx F$ if, for all $\tilde{P}, E\{\tilde{P}/\tilde{X}\} \approx F\{\tilde{P}/\tilde{X}\}$. □

4.3 Observation congruence

We must now tackle the difficulty that $\approx$ is not a congruence. We look for a congruence which is as close to $\approx$ as possible. The definition is as follows, and we use the equality symbol "=" for the congruence relation since it is our final choice of congruence over P.

4.3.1 Definition $P = Q$, i.e. P and Q are *strongly congruent*, if for all $\alpha \in Act$

(i) Whenever $P \xrightarrow{\alpha} P'$ then for some Q', $Q \stackrel{\hat{\alpha}}{\Rightarrow} Q'$ and $P' \approx Q'$;
(ii) Whenever $Q \xrightarrow{\alpha} Q'$ the, for some P', $P \stackrel{\hat{\alpha}}{\Rightarrow} P'$ and $P' \approx Q'$. □

Note that this is a direct definition in terms of $\approx$, and that the clauses differ from those for $\approx$ only in the use of $\stackrel{\hat{\alpha}}{\Rightarrow}$ in place of $\stackrel{\hat{\alpha}}{\Rightarrow}$; thus each action of P or Q must be matched by at *least* one action of the other. But this only applies to the initial actions. We immediately extend equality to expressions in the obvious way:

4.3.2 Definition Let $fv(E), fv(F) \subseteq \tilde{X}$; then $E = F$ if, for all $\tilde{P}, E\{\tilde{P}/\tilde{X}\} = F\{\tilde{P}/\tilde{X}\}$. □

We can give a simple alternative characterisation of equality as follows:

4.3.3 Proposition $P = Q$ iff, for all R, $P + R \approx Q + R$.

Proof($\Rightarrow$) It is easy to show that $\{\langle P + R, Q + R\rangle\} \cup \approx$ is a bisimulation.
($\Leftarrow$) Suppose $P \neq Q$. Then, for example, there are α and P' such that $P \xrightarrow{\alpha} P'$ but, whenever $Q \stackrel{\hat{\alpha}}{\Rightarrow} Q'$, then $P' \not\approx Q'$. Choose $R \equiv \ell.\mathbf{0}$, $\ell \notin \mathcal{L}(P) \cup \mathcal{L}(Q)$. Clearly $P + R \xrightarrow{\alpha} P'$; it remains to show that whenever $Q + R \stackrel{\hat{\alpha}}{\Rightarrow} Q'$ then $P' \not\approx Q'$. If $\alpha = \tau$ and $Q' \equiv Q + R$ then $P' \not\approx Q'$ since $Q + R \xrightarrow{\ell} \mathbf{0}$ while P cannot perform ℓ; otherwise $Q + R \stackrel{\hat{\alpha}}{\Rightarrow} Q'$, and since $R \stackrel{\hat{\alpha}}{\Rightarrow} Q'$ is impossible (because $\alpha \neq \ell$) we must have $Q \stackrel{\hat{\alpha}}{\Rightarrow} Q'$, so $P' \not\approx Q'$ by assumption. □

4.3.4 Proposition $P \sim Q$ implies $P = Q$ implies $P \approx Q$.

Proof For the first part, use property (*) of strong congruence (3.1), together with the definition of quality (4.3.1) and Prop 4.2.10. For the second part, $P = Q$ implies $P + \mathbf{0} \approx Q + \mathbf{0}$ by 4.3.3; now use $P + \mathbf{0} \sim P$. □

4.3.5 Proposition Observation congruence (=) is an equivalence relation.

Proof Use 4.3.3. □

4.3.6 Proposition $P \approx Q$ implies $\alpha.P = \alpha.Q$.

Proof Direct from 4.3.1. □

Thus the Action constructor not only preserves $\approx$, but strengthens it to equality. We are now ready to show that equality is a congruence.

4.3.7 Proposition If $P = Q$ then $\alpha.P = \alpha.Q$, $P + R = Q + R$, $P|R = Q|R$, $P\backslash L = Q\backslash L$ and $P[f] = Q[f]$.

Proof The first follows from 4.3.6, and the second from 4.3.3 using associativity of Summation. The remainder follow directly from the definition of equality (4.3.1) together with 4.2.12. □.

It remains to show that Recursion preserves equality.

4.3.8 Proposition If $\tilde{E} = \tilde{F}$ then $\mathrm{FIX}\,\tilde{X}\tilde{E} = \mathrm{FIX}\,\tilde{X}\tilde{F}$.

Proof It is enough to treat the case in which $fv(\tilde{E}), fv(\tilde{F}) \subseteq \tilde{X}$; the general case then follows from 4.3.2. So let $\tilde{P} \equiv \mathrm{FIX}\,\tilde{X}\tilde{E}$ and $\tilde{Q} \equiv \mathrm{FIX}\,\tilde{X}\tilde{F}$. We proceed, as in Prop 3.4.6, to show that

$$\mathcal{S} = \{\langle G\{\tilde{P}/\tilde{X}\}, G\{\tilde{Q}/\tilde{X}\}\rangle : fv(G) \subseteq \tilde{X}\}$$

is a bisimulation up to $\approx$. In fact we need something slightly stronger (since this would only show $\tilde{P} \approx \tilde{Q}$, by setting $G \equiv X_i$ for each i); we shall prove that

$$\begin{array}{l}\text{If } G\{\tilde{P}/\tilde{X}\} \xrightarrow{a} P' \text{ then, for some } Q', \\ G\{\tilde{Q}/\tilde{X}\} \stackrel{a}{\Rightarrow} Q' \text{ and } P'\,\mathcal{S} \approx Q'\end{array} \qquad (*)$$

This clearly makes $\mathcal{S}$ a bisimulation up to $\approx$, and by setting $G \equiv X$; for each i we get $\tilde{P} = \tilde{Q}$ by using the definition of equality (4.3.1).

Just as in Prop 3.4.6, we prove $(*)$ by action induction; the details are only slightly more refined than in 3.4.6. □

We now turn to two further results which show how close the relations of bisimilarity and equality are to each other.

4.3.9 Definition P is *stable* if $P \xrightarrow{\tau} P'$ is impossible for all P' □

4.3.10 Proposition If $P \approx Q$ and both are stable then $P = Q$.

Proof An easy consequence of Def 4.3.1. □

The second of the two results is rather surprising:

4.3.11 Proposition $P \approx Q$ iff $(P = Q$ or $P = \tau.Q$ or $\tau.P = Q)$

Proof ($\Leftarrow$) Easy, by 4.2.11 and 4.3.4.
($\Rightarrow$) We distinguish three cases:

(1) $P \xrightarrow{\tau} P^* \approx Q$ for some P^*

(2) $Q \xrightarrow{\tau} Q^* \approx P$ for some Q^*

(3) Neither (1) nor (2) obtains.

In cases (1) and (2) we prove $P = \tau.Q$ and $\tau.P = Q$ respectively, while in case (3) we prove $P = Q$. □

The results which we have obtained in this section so far provide a rich repertoire of tools for proving bisimilarity or equality. The most powerful of these is to prove $P \approx Q$ by exhibiting a bisimulation containing the pair $\langle P, Q \rangle$; often this can be strengthened to an equality by stability (4.3.10) or by applying the Action constructor (4.3.6).

In the course of proofs we can freely use the static laws (3.3.2) and the Expansion Theorem (3.3.4) and their corollaries and also the monoid properties of Summation (3.3.1), since strong congruence is stronger than equality. To enrich the algebra of equality, we now extend the four laws of 3.3.1 by three more concerning both Action and Summation; they can be regarded as absorption laws due to the unobservability of τ, and we call these seven laws together the *dynamic laws*. It is remarkable that our algebraic laws can be classified into three classes; the dynamic laws, the static laws, and the single Expansion Theorem which captures the interaction between the dynamic and static constructors.

4.3.12 Proposition (τ-laws)

(1) $\alpha.\tau.P = \alpha.P$

(2) $P + \tau.P = \tau.P$

(3) $\alpha.(P + \tau.Q) + \alpha.Q = \alpha.(P + \tau.Q)$

Proof Easy application of the definition of equality (4.3.1). □

Later we shall see that the dynamic laws are complete for equality of sequential processes, i.e. those not involving the static constructors.

4.4 Unique fixed points up to equality

In 3.5 we saw that equations $\tilde{X} \sim \tilde{E}$ possess a unique solution up to $\sim$, provided that the variables $\tilde{X}$ are weakly guarded in $\tilde{E}$. We now ask when the equations $\tilde{X} = \tilde{E}$ possess a unique solution up to $=$. Immediately we see that weak guarding is not a strong enough condition; for if we consider the equation $X = \tau.X$, then we see that any process of the form $\tau.P$ is a solution, since by 4.3.12(1) $\tau.P = \tau.\tau.P$. Similarly, we can show that $X = \tau.X + a.\mathbf{0}$ has $\tau.(P + a.\mathbf{0})$ as solution for any P. This leads us to strengthen the guarding condition:

4.4.1 Definition X is (*strongly*) *guarded* in E if every free occurrence of X lies within a subexpression $\ell.F$ of E. □

But we now meet another difficulty. In the presence of the static constructors, it is not enough to demand that X be (strongly) guarded in E to ensure that $X = E$ has a unique solution. Consider $X = (\bar{a}.\mathbf{0}|a.X)\backslash a$. This simplifies to $X = \tau.(X\backslash a)$, and again $\tau.P$ will always be a solution provided that $a, \bar{a} \notin \mathcal{L}(P)$. So we look for a further constraint.

4.4.2 Definition X is *sequential* in E if no free occurrence of X lies within a static constructor in E. □

Thus, for example, X is sequential in $a.(P\,|\,Q)\backslash b + \tau.X$ (though not guarded), but not in $a.(X\backslash a)$.

For the rest of this subsection we shall be concerned often with expressions G such that $fv(G) \subseteq \tilde{X}$, and we shall often write $G(\tilde{P})$ for $G\{\tilde{P}/\tilde{X}\}$. We first need a lemma, which states that under a certain condition G's successors are independent of $\tilde{P}$.

4.4.3 Lemma Let G have free variables $\subseteq \tilde{X}$, all guarded and sequential in G, and let $G(\tilde{P}) \xrightarrow{\alpha} P'$. Then for some H with $fv(H) \subseteq \tilde{X}$, $G \xrightarrow{\alpha} H$ and $P' \equiv H(\tilde{P})$; also, for any $\tilde{Q}$, $G(\tilde{Q}) \xrightarrow{\alpha} H(\tilde{Q})$. Moreover the variables $\tilde{X}$ are sequential in H, and if $\alpha = \tau$ they are also guarded in H.

Proof By action induction. □

Now we are ready to prove the main result.

4.4.4 Proposition Let the variables $\tilde{X}$ be guarded and sequential in $\tilde{E}$, and let $\tilde{F} = \tilde{E}\{\tilde{F}/\tilde{X}\}$. Then $\tilde{F} = \mathrm{FIX}_{\tilde{X}}\tilde{E}$.

Proof By Definition 4.3.2 it will be enough to assume $fv(\tilde{E}) \subseteq \tilde{X}$. Then it will be enough to prove that if $\tilde{P} = \tilde{E}(\tilde{P})$ and $\tilde{Q} = \tilde{E}(\tilde{Q})$ then $\tilde{P} = \tilde{Q}$. For this we

shall show that

$$\mathcal{S} = \{(G(\tilde{P}), G(\tilde{Q})) : fv(G) \subseteq \tilde{X},\ \tilde{X} \text{ sequential in } G\} \cup \mathrm{Id}_p$$

is a bisimulation up to $\approx$. Actually we prove the stronger fact that

$$\begin{array}{l} \text{If } G(\tilde{P}) \xrightarrow{\alpha} P' \text{ then, for some } Q', \\ G(\tilde{Q}) \stackrel{\alpha}{\Rightarrow} Q' \text{ and } P' \approx \mathcal{S} \approx Q' \end{array} \qquad (*)$$

This makes $\mathcal{S}$ a bisimulation up to $\approx$, and gives us $\tilde{P} = \tilde{Q}$ by setting $G \equiv X_i$ for each i.

We first prove something like $(*)$ for silent actions:

$$\begin{array}{l} \text{If } G(\tilde{P}) \Rightarrow P' \text{ then, for some } Q', \\ G(\tilde{Q}) \Rightarrow Q' \text{ and } P' \approx \mathcal{S} \approx Q' \end{array} \qquad (**)$$

so let $G(\tilde{P}) \Rightarrow P'$. Now $G(\tilde{P}) = G(\tilde{E}(\tilde{P}))$, so

$$G(\tilde{E}(\tilde{P})) \Rightarrow P'' \approx P'$$

But $\tilde{X}$ are guarded and sequential in G, so by Lemma 4.4.3 repeatedly $\tilde{X}$ are guarded and sequential in some G' where

$$P'' \equiv G'(\tilde{P}) \text{ and } G(\tilde{E}(\tilde{Q})) \Rightarrow G'(\tilde{Q})$$

and since $G(\tilde{Q}) = G(\tilde{E}(\tilde{Q}))$ we also have

$$G(\tilde{Q}) \Rightarrow Q' \approx G'(\tilde{Q})$$

which establishes $(**)$

We now prove $(*)$. Let $G(\tilde{P}) \xrightarrow{\alpha} P'$, so that

$$G(\tilde{E}(\tilde{P})) \Rightarrow\!\xrightarrow{\alpha}\!\Rightarrow P'' \approx P'$$

Now by Lemma 4.4.3 repeatedly, taking us through $\Rightarrow\!\xrightarrow{\alpha}$, there is G' with free sequential variables $\tilde{X}$ such that this derivation is of form

$$G(\tilde{E}(\tilde{P})) \Rightarrow\!\xrightarrow{\alpha} G'(\tilde{P}) \Rightarrow P'' \approx P'$$

and also

$$G(\tilde{E}(\tilde{Q})) \Rightarrow\!\xrightarrow{\alpha} G'(\tilde{Q})$$

and by $(**)$ this derivation can be continued by

$$G'(\tilde{Q}) \Rightarrow Q'' \text{ with } P'' \approx \mathcal{S} \approx Q''$$

Finally, since $G(\tilde{E}(\tilde{Q})) = G(\tilde{Q})$, we deduce

$$G(\tilde{Q}) \stackrel{a}{\Rightarrow} Q' \approx Q''$$

So $P' \approx S \approx Q'$ and the proof is complete. □

This result is important in practice. Often we have a specification FIX $(\tilde{X} = \tilde{E})$ where $\tilde{X}$ are guarded and sequential in $\tilde{E}$, and we wish to prove a highly concurrent system $\tilde{P}$ equal to the specification by proving $\tilde{P} = \tilde{E}(\tilde{P})$.

4.5 Completeness of equational laws

We now turn to the question of whether we have "enough" equational laws. We have alluded in 2.5 to one positive result; under a mild constraint the static laws of 3.3.2 are complete for the algebra of flow graphs, which is evidence that we have "enough" laws concerning the static constructors alone. Here we consider mainly the dynamic constructors and recursion; we shall only be able to summarise the results, due to space limitation.

Our first result is a negative one. The mixture of Recursion and Composition (with the other constructors) is enough to allow us effectively to translate the i^{th} Turing machine, in any enumeration, to a process TM_i such that $\mathrm{TM}_i \sim \mathrm{FIX}\, X(\tau.X)$ iff the machine diverges starting from blank tape, and hence the set of valid strong congruences is not recursively enumerable, so cannot be effectively axiomatised. This translation is best carried out by first defining a process representing a stack and then modelling a Turing machine by a pair of stacks, and only employs finitely-indexed Summations and Recursions. A similar result holds for equality in place of strong congruence. We therefore look at subsets of P; for the remainder of this section we assume first that all Summations and Recursions are finitely indexed.

4.5.1 Definition E is *finite* if it contains no Recursion; E is *sequential* if it contains no Composition, Restriction or Relabelling. □

4.5.2 Proposition The Expansion theorem (3.3.4) and the monoid laws for Summation (3.3.1) are complete for strong congruence of finite processes.

Proof (outline) In the absence of Recursion, the Expansion theorem can be used to convert every finite process to a strongly congruent sequential process. This can be placed in the standard form $\Sigma\alpha_i.P_i$ where each P_i is also in standard form, using 3.3.1(4) to remove redundant occurrences of $\mathbf{0}$. Then it can be shown that the other monoid laws are enough to prove all valid strong congruences between standard forms. □

4.5.3 Proposition The Expansion theorem, the monoid laws and the τ-laws (4.3.12) are complete for equality of finite processes.

Proof (outline) We convert each process to standard form $P \equiv \Sigma \alpha_i.P_i$ as in 4.5.2. Then, by 4.3.12(2) and (3) we can convert P to a "full" standard form in which, whenever $P \stackrel{a}{\Rightarrow} P'$, $\alpha.P$ is a summand of P – i.e. $P \stackrel{\alpha}{\rightarrow} P'$. Finally, it can be shown that the monoid laws with 4.3.12(1) are enough to prove all valid equations between full standard forms. □

Now we turn to processes which are sequential but not finite. There are three laws for Recursion, which are easily proved sound, and whose purpose is to convert all Recursions to those in which the expressions $\tilde{E}$ are weakly or strongly guarded. For simplicity we state these only for single Recursions $\mathrm{FIX}\, XE$.

4.5.4 Proposition $\mathrm{FIX}\, X(X + E) \sim \mathrm{FIX}\, XE$ □

4.5.5 Proposition

(1) $\mathrm{FIX}\, X(\tau.X + E) = \mathrm{FIX}\, X(\tau.E)$

(2) $\mathrm{FIX}\, X(\tau.(X + E) + F) = \mathrm{FIX}\, X(\tau.X + E + F)$ □

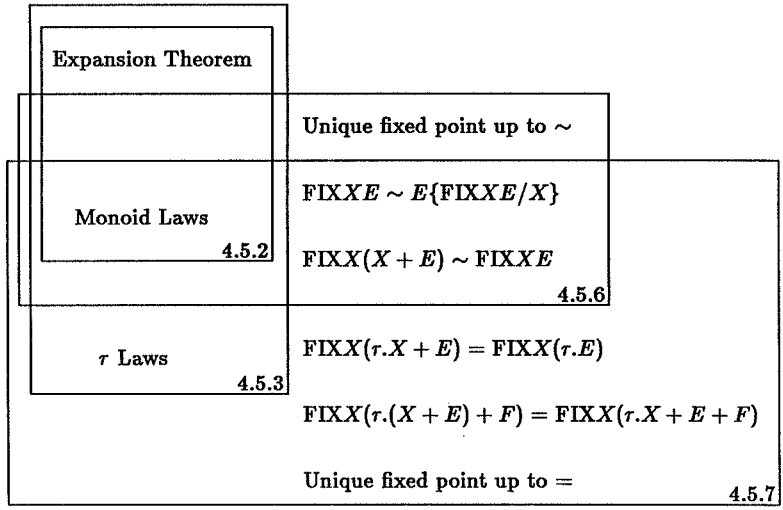
The point of these laws is that, once Recursions are weakly or strongly guarded, the appropriate unique fixed point rule (3.5.3 or 4.4.4) can be applied.

4.5.6 Proposition The monoid laws, together with the recursion laws for strong congruence (3.5.1, 3.5.3 and 4.5.4) are complete for strong congruence of sequential processes. □

The proof of this result is quite detailed and we omit it here; we also omit the proof of our final completeness result, the following:

4.5.7 Proposition The monoid laws, the τ-laws and the Recursion laws for equality (3.5.1, 4.5.4, 4.5.5 and 4.4.4) are complete for equality of sequential processes. □

The following diagram summarises the completeness results:



COMPLETE AXIOMATISATIONS

5. Analysis of bisimulation equivalences

5.1 Hierarchies of equivalence

Both strong congruence and bisimilarity are largest fixed points of a function $\mathcal{H}$ over binary relations, which can be defined for any Labelled Transition System $(P, A, \{\xrightarrow{a}: a \in A\})$ as follows:

5.1.1 Definition For any binary relation S over P, $\langle P, Q\rangle \in \mathcal{H}(S)$ iff, for all $a \in A$,

(i) Whenever $P \xrightarrow{a} P'$ then, for some Q', $Q \xrightarrow{a} Q'$ and $P' S Q'$
(ii) Whenever $Q \xrightarrow{a} Q'$ then, for some P', $P \xrightarrow{a} P'$ and $P' S Q'$ □

Here the set P need have no structure – except that provided by the action relations $\xrightarrow{a}$. If we take P to be the processes of our calculus, then strong congruence arises as the largest fixed point of $\mathcal{H}$ when $A = Act^*$ and $\xrightarrow{t}$ is as defined in 4.1.1(1) for $t \in Act^*$; bisimilarity arises by taking $A = \mathcal{L}^*$ and using the relations $\stackrel{s}{\Rightarrow}$ $(s \in \mathcal{L}^*)$ as defined in 4.1.1(3). Actually we used $\xrightarrow{\alpha}$ $(\alpha \in Act)$ in defining

strong congruence, but it is easy to see that using $\xrightarrow{t}$ does not alter the congruence; similarly - as hinted by Prop 4.2.9 - the action relations $\stackrel{s}{\Rightarrow}$ can be seen to yield bisimilarity. By choosing action relations indexed by sequences we get a more interesting hierarchy, defined for our arbitrary LTS as follows:

5.1.2 Definition The relations $\simeq_\kappa$ over P, for each ordinal κ, are defined as follows:

(1) $\simeq_0 = P \times P$

(2) $\simeq_{\kappa+1} = \mathcal{H}(\simeq_\kappa)$

(3) $\simeq_\lambda = \bigcap_{\kappa<\lambda} \simeq_\kappa$, for a limit ordinal λ □

We use O for the ordinals, and define in general:

5.1.3 Definition $\simeq \;=\; \bigcap_{\kappa\in O} \simeq_\kappa$ □

The following is a straightforward, partly standard, result:

5.1.4 Proposition

(1) If $\lambda > \kappa$ then $\simeq_\lambda \subseteq \simeq_\kappa$

(2) $\simeq_\kappa$ and $\simeq$ are equivalence relations

(3) $\simeq$ is the largest fixed point of $\mathcal{H}$ □

Thus both strong congruence and bisimilarity of processes are limits of decreasing sequences of equivalence relations. The following results about $\simeq_1$ are interesting:

5.1.5 Proposition

(1) $P \sim_1 Q$ iff, for all $t \in Act^*$, $(P \xrightarrow{t}$ iff $Q \xrightarrow{t})$

(2) $P \approx_1 Q$ iff, for all $s \in \mathcal{L}^*$, $(P \stackrel{s}{\Rightarrow}$ iff $Q \stackrel{s}{\Rightarrow})$ □

Thus, at the first level of the hierarchies which approximate strong congruence and bisimilarity, P and Q are equivalent just when they can perform the same sequences of *actions* in the first case, and of *visible actions* in the second case. We already know that $\sim$ and $\approx$ are smaller than $\sim_1$ and $\approx_1$; now we shall see that the hierarchies are properly decreasing at least as far as $\omega + 1$.

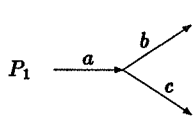
5.1.6 Proposition The sequence $\sim_0, \sim_1, \cdots, \sim_\omega$ is properly decreasing.

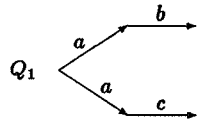
Proof(outline) For $i < \omega$, let P_i and Q_i be as follows:

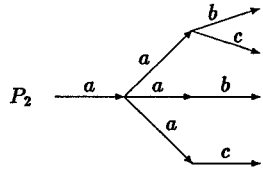
$P_0 \equiv b.\mathbf{0}$ $\qquad$ $Q_0 \equiv c.\mathbf{0}$

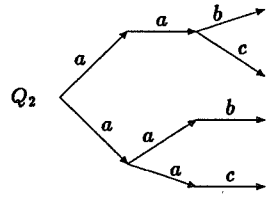
$P_{i+1} \equiv a.(P_i + Q_i)$ $\qquad$ $Q_{i+1} \equiv a.P_i + a.Q_i$

where $b \neq c$.









Then by induction it is easy to prove

$$P_i \sim_i Q_i \sim_i P_i + Q_i$$
$$P_i \not\sim_{i+1} Q_i \not\sim_{i+1} P_i + Q_i \not\sim_{i+1} P_i$$

which implies the stated result. □

5.1.7 Proposition The sequence $\approx_0, \approx_1, \cdots, \approx_\omega$ is properly decreasing.

Proof The same proof suffices, since $\sim_\kappa$ and $\approx_\kappa$ coincide in the absence of τ-actions. □

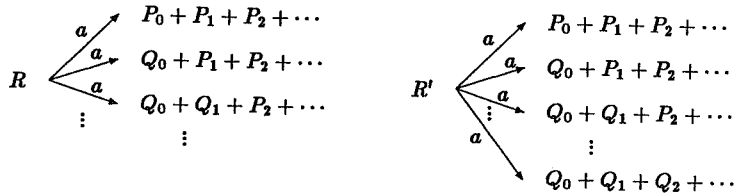
We now show that the sequences decrease still further if we allow infinite sums.

5.1.8 Proposition $\sim_{\omega+1} \neq \sim_\omega$ and $\approx_{\omega+1} \neq \approx_\omega$

Proof Using P_i and Q_i as above, let R_i and R_ω be as follows:

$$\begin{aligned} R_0 &\equiv \Sigma_{0 \leq i < \omega} P_i \\ R_1 &\equiv Q_0 + \Sigma_{1 \leq i < \omega} P_i \\ R_2 &\equiv Q_0 + Q_1 + \Sigma_{2 \leq i < \omega} P_i \\ &\cdots \\ R_\omega &\equiv Q_0 + Q_1 + Q_2 + \cdots \end{aligned}$$

Then set $R \equiv \Sigma_{i<\omega} a.R_i$ and $R' \equiv R + a.R_\omega$



First, it follows from the analysis in 5.1.6 that $R_\omega \sim_i R_i$ but $R_\omega \not\sim_{i+1} R_i$. We now show that $R \sim_i R'$ for each $i < \omega$, and hence $R \sim_\omega R'$. For clearly all derivatives of R are exactly matched by R', and all derivatives of R' are exactly matched by R except $R' \xrightarrow{a} R_\omega$. But we can choose $R \xrightarrow{a} R_i$ to match this up to $\sim_i$, and this is enough to ensure $R \sim_i R'$.

On the other hand there is no a-successor of R which matches the a-successor R_ω of R' up to $\sim_\omega$, since $R_\omega \not\sim_\omega R_i$; hence $R \not\sim_{\omega+1} R'$.

We conclude that $\sim_{\omega+1} \neq \sim_\omega$, and also $\approx_{\omega+1} \neq \approx_\omega$ since $\sim_\kappa$ and $\approx_\kappa$ coincide in the absence of τ. □

It is interesting to note that this example has infinite branching but only finite paths. It can be converted to an example with only finite branching but infinite paths (involving τ), to show that $\approx_{\omega+1} \neq \approx_\omega$. But we shall show that $\sim_{\omega+1}$ and $\sim_\omega$, and also $\approx_{\omega+1}$ and $\approx_\omega$, coincide on finite processes.

5.2 Logical characterisation of the hierarchy

The distinction between P_2 and Q_2 in 5.1.6 can be described by the following property enjoyed by Q_2 but not by P_2:

> An a-action is *possible* such that after any further a-action it is *necessary* that both a b-action and a c-action are *possible*. (*)

This triple alternation of *possibility* and *necessity* seems to be what is needed to express a property which will distinguish P_2 from Q_2, with respect to $\sim_3$. We therefore introduce the following logical language $\mathcal{PL}$ to characterise the members of our arbitrary LTS:

5.2.1 Definition $\mathcal{PL}$ is the smallest class of formulae such that

(1) If $F \in \mathcal{PL}$ then $\neg F \in \mathcal{PL}$

(2) If $F_i \in \mathcal{PL}$ for each i in an indexing set I, then $\bigwedge_{i\in I} F_i \in \mathcal{PL}$

(3) If $F \in \mathcal{PL}$ and $a \in A$, then $\langle a \rangle F \in \mathcal{PL}$ □

$\mathcal{PL}$ is non-empty, since (2) ensures that the empty conjunction $\bigwedge_{i\in\emptyset} F_i$, which we write as TRUE, is present. We also define FALSE $\equiv \neg$TRUE, $\bigvee_{i\in I} F_i \equiv \neg \bigwedge_{i\in I} \neg F_i$, and $[a]F \equiv \neg\langle a \rangle \neg F$. The property (*) above will be formalised as

$$\langle a \rangle [a](\langle b \rangle \text{TRUE} \wedge \langle c \rangle \text{TRUE}) \qquad (**)$$

In fact, to ensure that this captures the meaning intended by (*), we interpret $\mathcal{PL}$ by the *satisfaction* relation $\models \subseteq \mathcal{P} \times \mathcal{PL}$ as follows:

5.2.2 Definition

(1) $P \models \neg F$ iff not $P \models F$

(2) $P \models \bigwedge_{i\in I} F_i$ iff, for all $i \in I$, $P \models F_i$

(3) $P \models \langle a \rangle F$ iff, for some P', $P \xrightarrow{a} P' \models F$ □

Thus for P_2 and Q_2 in 5.1.6 and the formula F at (**) we have $P_2 \not\models F$, $Q_2 \models F$.

We now wish to stratify $\mathcal{PL}$. We first define the *depth* of F, an ordinal number representing the maximum nesting of diamonds in F:

5.2.3 Definition

(1) $\text{depth}(\neg F) = \text{depth}(F)$

(2) $\text{depth}(\bigwedge_{i\in I} F_i) = \sup_{i\in I} (\text{depth}(F_i))$

(3) $\text{depth}(\langle a \rangle F) = \text{depth}(F)+1$ □

Then we define the levels of $\mathcal{PL}$ as follows:

5.2.4 Definition $\mathcal{PL}_\kappa = \{F\colon \text{depth}(F) \leq \kappa\}$ for $\kappa \in \mathcal{O}$ □

Now we are ready for the main result, that each level in the equivalence hierarchy is exactly characterised by the corresponding level in $\mathcal{PL}$.

5.2.5 Proposition

(1) $P \simeq_\kappa Q$ iff, for every $F \in \mathcal{PL}_\kappa$, $(P \models F \text{ iff } Q \models F)$

(2) $P \simeq Q$ iff, for every $F \in \mathcal{PL}$, $(P \models F \text{ iff } Q \models F)$.

Proof(outline). (2) follows directly from (1) since $\mathcal{PL} = \bigcap_\kappa \mathcal{PL}_\kappa$. The proof of (1) is by induction on κ. The crucial step is in proving (1) for $\kappa + 1$; assuming $P \not\approx_{\kappa+1} Q$ we have to find $F \in \mathcal{PL}_{\kappa+1}$ such that $P \models F$ and $Q \not\models F$. By assumption, there is w.l.o.g. an action $P \xrightarrow{a} P'$ such that whenever $Q \xrightarrow{a} Q'$ then $P' \not\approx_\kappa Q'$. So let $\{Q_i : i \in I\}$ be the set of Q' such that $Q \xrightarrow{a} Q'$; then (with the help of negation) there exists $F_i \in \mathcal{PL}_\kappa$, for each $i \in I$, such that $P' \models F_i$ and $Q_i \not\models F_i$. Take $F \equiv \langle a \rangle \bigwedge_{i \in I} F_i$; clearly $P \models F$ and $Q \not\models F$.

The remainder of the proof, including the inductive step for limit ordinals, is routine. □

The only part of this proof which requires a possibly infinite conjunction of formulae is the part we have detailed. From this we learn that for finite processes, since the set $\{Q_i : i \in I\}$ in the proof must be finite, the finitary language $\mathcal{PL}^{\text{fin}}$ (i.e. only finite conjunctions) is enough. But every formula in this subset has finite depth, so $\mathcal{PL}^{\text{fin}}_\omega = \mathcal{PL}^{\text{fin}}$; hence we deduce the following:

5.2.6 Proposition On finite processes, $\sim_\omega$ coincides with $\sim$ and $\approx_\omega$ coincides with $\approx$. □

6. Confluent processes

6.1 Determinism

We would like to define a notion of determinism for processes, and to identify a set of constructions which preserve determinism. A benefit from this would be a sub-language of processes which can be seen to be deterministic from their syntactic form, and which admit special proof techniques for establishing equivalence and other properties.

A natural first attempt at a definition of determinism might be as follows: P is deterministic if whenever $P \stackrel{s}{\Rightarrow} P'$ and $P \stackrel{s}{\Rightarrow} P''$, then $P' \approx P''$. This would immediately yield that, whenever $P \Rightarrow P'$, then $P \approx P'$; an advantage would be that, in checking whether some S is a bisimulation up to $\approx$, only a subset of the pairs in S would need to be tested.

We certainly hope that our eventual definition will imply the above property, but the property as it stands is too weak to be preserved by simple constructions. For let $P \equiv a.0 + b.0$ and $Q \equiv \bar{a}.0$; both clearly possess the property, but (if $a \neq b$) $(P|Q)\backslash a = \tau.0 + b.0$ does not. We therefore seek something stronger. Since we do wish to allow this kind of restricted composition, it appears that summation

should only be admitted when each alternative action does not preclude the later occurrence of any of the others; this means that $P \equiv a.0 + b.0$ should be excluded, but $a.b.0 + b.a.0$ should be admitted.

6.2 Confluence

We proceed to define a notion of confluent process.

6.2.1 Definition The set C of *confluent processes* is the largest set closed under successor and such that, for all $P \in C$ and derivatives P', P'' as shown, Q can be found to complete each of the following diagrams:

$$\begin{array}{ccc} P & \xrightarrow{\tau} & P' \\ \Downarrow & & \Downarrow \\ P'' & \Rightarrow\approx & Q \end{array} \qquad \begin{array}{ccc} P & \xrightarrow{\tau} & P' \\ \ell\Downarrow & & \ell\Downarrow \\ P'' & \Rightarrow\approx & Q \end{array} \qquad \begin{array}{ccc} P & \xrightarrow{\ell} & P' \\ \ell'\Downarrow & & \ell'\Downarrow \\ P'' & \stackrel{\ell}{\Rightarrow}\approx & Q \end{array} \qquad \begin{array}{ccc} P & \xrightarrow{\ell} & P' \\ \ell\Downarrow & & \Downarrow \\ P'' & \Rightarrow\approx & Q \end{array}$$

$$(\ell \neq \ell')$$

□

To justify the definition, we note that there is a *largest* such set – namely the union of all sets satisfying the conditions. We present the definition in this form since it shows just how each (single) action of P should "commute up to $\approx$" with a sequence of other actions. Note however that the last diagram, involving two possibly different ℓ-descendants, only requires common τ-descendants (up to $\approx$). There is another way of characterising confluence which involves the idea of the *excess* of $r \in \mathcal{L}^*$ over $s \in \mathcal{L}^*$; this excess is gained by working through r from left to right deleting any member of s – but respecting multiplicity of occurrence. Thus the excess of $r = aba$ over $s = bacb$ is a, while the excess of s over r is cb.

6.2.2 Definition r/s, the *excess of* r *over* s, is defined as follows:

$$\begin{aligned} \varepsilon/s &= \varepsilon \\ (\ell r)/s &= \begin{cases} \ell(r/s) & \text{if } \ell \notin s \\ r/(s/\ell) & \text{if } \ell \in s \end{cases} \end{aligned}$$

□

6.2.3 Proposition P is confluent iff, for all $r, s \in \mathcal{L}^*$, the following diagram can be completed in the sense of 6.2.1:

$$\begin{array}{ccc} P & \stackrel{r}{\Longrightarrow} & P' \\ s\Downarrow & & s/r\Downarrow \\ P'' & \stackrel{r/s}{\Rightarrow}\approx & Q \end{array}$$

□

The essential properties of confluence are captured by three results relating it to bisimilarity. First, we wish to know that it is really a property of observation equivalence classes.

6.2.4 Proposition If $P \approx Q$ and P is confluent then so is Q. □

Next, we wish to know that silent actions have "no effect" up to $\approx$ for confluent processes:

6.2.5 Proposition If P is confluent and $P \xrightarrow{\tau} P'$ then $P \approx P'$. □

Finally, we find that bisimilarity of confluent processes reduces to accepting the same sequences of visible actions.

6.2.6 Proposition If P and Q are confluent then $P \approx Q$ iff $P \approx_1 Q$ □

These results indicate that confluent processes are very well behaved.

6.3 Constructions preserving confluence

Let us first see that Summation and Composition fail to preserve confluence in general. For the first, clearly $a.\mathbf{0}$ and $b.\mathbf{0}$ are confluent, but $a.\mathbf{0}+b.\mathbf{0}$ is not (see the third diagram of 6.2.1). For the second, we need only observe that $a.\mathbf{0}|\bar{a}.\mathbf{0}$ is not confluent, since it is equal to $a.\bar{a}.\mathbf{0}+\bar{a}.a.\mathbf{0}+\tau.\mathbf{0}$. The other constructors do preserve confluence, so we look for constrained forms of Summation and Composition.

We first observe that $P|Q$ may not be confluent, even if P and Q are, if $\mathcal{L}(P) \cap \mathcal{L}(Q) \neq 0$; an example is $a.b.\mathbf{0}|a.\mathbf{0}$, for which we can check that the fourth condition of 6.2.1 fails. Next, $P|Q$ may not be confluent if $\mathcal{L}(P) \cap \overline{\mathcal{L}(Q)} \neq 0$; the example in the previous paragraph suffices. We are led to the following definition:

6.3.1 Definition $P||_L Q \stackrel{\text{def}}{=} (P|Q)\backslash L$ is a *Confluent Composition* if $\mathcal{L}(P) \cap \mathcal{L}(Q) = \emptyset$ and $\mathcal{L}(P) \cap \overline{\mathcal{L}(Q)} \subseteq L \cup \bar{L}$. □

For example, $a.b.\mathbf{0}||_a c.\bar{a}.\mathbf{0}$ is a Confluent Composition, and the reader may easily verify that it is confluent. We now turn to a mixture of Action and Summation:

6.3.2 Definition $(\alpha_1|\alpha_2|\cdots|\alpha_n).P \stackrel{\text{def}}{=} \Sigma_{1\leq i\leq n}\alpha_i.(\alpha_1|\cdots|\alpha_{i-1}|\alpha_{i+1}|\cdots|\alpha_n).P$ is a *Confluent Action.* □

The effect of this construction is to perform $\alpha_1, \cdots, \alpha_n$ in any order before entering P. A simple example is $(a|b).\mathbf{0} \equiv a.b.\mathbf{0} + b.a.\mathbf{0}$. Note that the Action constructor $\alpha.P$ is a special case with $n = 1$.

6.3.3 Definition E is in *confluent form* if it is built using only Confluent Action, Confluent Composition, Restriction, Relabelling and Recursion. □

We now come to the main result of this section.

6.3.4 Proposition Every process in confluent form is confluent. □

Space limitation forbids us to give the proof of this result, which is rather detailed especially in the case of Confluent Composition. Its importance is that many practical examples of systems can be built in confluent form, and are thus amenable to especially simple analysis based upon Propositions 6.2.5 and 6.2.6. There are indeed weaker, i.e. more general, notions of confluent form which cover a wider class of applications.

In general, however, we are forced to admit that even if a complete system is confluent, and may be proved so by adhoc analysis, it is often impossible or unnatural to build it in confluent form. One can find other examples of this general phenomenon in programming; for instance, a whole program may be always-terminating, but it may be most inconvenient to build it by adhering to a discipline – such as primitive recursion – which makes this property manifest from the form of the program.

7. Sources and Related Work

In this Chapter we have accumulated the main results of a particular theory of communicating systems, and we have omitted or only outlined several of the proofs. The main ideas were first presented in [Mil 1]. The most important new ingredient after this was the notion of bisimulation discovered by David Park [Par]; though the observation equivalence of [Mil 1] coincides with that based on bisimulation in most interesting cases, the conceptual framework of bisimulation is distinctly superior, and its proof technique is distinctly more tractable.

The algebra of flow graphs, and a completeness theorem for the static laws, was given in [Mil 2]. The notion of infinite Summation, and its use in deriving the extended language of 2.6, was introduced in [Mil 3] which also contains some more detailed proofs. The basic theory of bisimulations is covered in [Mil 4]. Proposition 4.3.11 is due to Matthew Hennessy, who also provided the idea for the simple proof of Proposition 4.5.3, completeness of laws for equality of finite processes. The completeness results for finite processes (Propositions 4.5.2 and

4.5.3) were first reported in [HM]; those for sequential processes were proved in [Mil 5,6]. The hierarchies of equivalence and the logical characterisations of Section 5 were also first treated in [HM], and also in [Mil 7]. The study of confluence was begun in [Mil 1], but its treatment is greatly improved here with the help of Park's bisimulation.

A limitation of the calculus presented here is that synchronisation (or handshake communication) only occurs between two processes. For some applications, multi-way synchronisation – or instantaneous broadcasting – is a valuable expressive device. This is present for example in the path expressions of [CH], and in the algebra of CSP [Hoa 2]. In [Mil 3] it was shown how the present calculus can be generalised to admit multi-way synchronisation by treating the set of atomic actions as a commutative monoid, whose product represents simultaneous occurence and whose unit is the silent action.

Other authors have treated algebras of processes which differ from the present in two respects; in the constructions used, and in the notion of equivalence or congruence. In [HBR] Hoare et al introduce an important equivalence, *refusals* equivalence, which lies strictly between trace equivalence (our $\approx_1$) and bisimilarity ($\approx$); its importance is that it appears to be the weakest tractable equivalence which is substitutive and which respects the possibility of deadlock. The pleasant feature of this notion is that it admits a useful set-theoretic model, the so-called *refusal sets*. This is Hoare's chosen model for his CSP, first presented in his pioneering paper [Hoa 1] in which the idea of handshake communication was first shown to be an important feature of concurrent programming, and which led to the rendezvous concept in Ada. The constructions in [HBR] admit multi-way communication. An attempt to compare different sets of constructors, and different equivalences, is made in [Mil 8].

Another equivalence, testing equivalence, was introduced by Hennessy and de Nicola [HdeN]. The idea is that two processes are equivalent iff they exhibit the same pattern of success when subjected to all possible testing processes. Interestingly, it has been shown by de Nicola [deN] that in essence this notion coincides with the refusals equivalence mentioned above; thus two rather different characterisations yield the same equivalence. Hennessy [Hen] has provided a model of trees for this interpretation.

The notion of testing considered by Hennessy and de Nicola is natural, but there are other testing notions which are stronger – i.e. they distiguish more processes. In a striking recent paper [Abr], Abramsky introduces a family of testing notions, of which the strongest exactly characterises bisimilarity.

Bergstra and Klop [BK] have developed a rich algebra of processes, in which many equivalences and many different constructions are studied within a single framework. A significant feature of their work is the smooth treatment of sequen-

tial composition of processes; though sequential composition is definable in terms of (parallel) Composition in the present calculus – which is why it is here omitted from the basic constructors, for the sake of parsimony – its presence as a basic constructor leads to a pleasant algebra. The emphasis in Bergstra's and Klop's work is more upon algebra, less upon operational semantics, and in more recent papers they have extensively studied the effect – including consistency – of many combinations of equational laws.

The present calculus is asynchronous in the sense that the relative speed of concurrent processes is not modelled. An algebra of synchronous processes was given in [Mil 3], but has been carried much further by Gerard Berry and his followers, starting with MEIJE [AB]. A noteworthy achievement of this group if the result of Simone [deS] that all constructors with a certain form of operational definition are definable in terms of a basic few.

The present calculus adopts the viewpoint that concurrency and temporal interleaving of actions are indistinguishable by observation; this is the essence of the Expansion Theorem. This intentional lack of distinction is strongly questioned by many, including particularly the Petri Nêt community. The *event structures* introduced by Plotkin et al [NPW] provide an important model in which these different views can be studied in the same framework; a recent paper by Boudol and Castellani carries this unifying approach further [BC].

This brief survey does not do justice to all the recent work on algebras of processes. The field is becoming a very rich one, and it is too early to predict what will survive and – in particular – which constructions and equivalences will find the most fruitful application. This Chapter has attempted not to compare different choices, but to present one approach in as much depth as space permits; it is only by pursuing one choice in depth that its coherence can be adequately assessed.

References

Abr S Abramsky, Observation Equivalence as a Testing Equivalence. J TCS 53, pp 225-241, 1987.

AB D Austry and G Boudol, Algèbre de processus et synchronisation. J TCS 30,1, pp91-131, 1984.

BC G Boudol and I Castellani, On the Semantics of Concurrency: Partial Orders and Transition Systems. Proc CAAP 87, Lecture Notes in Computer Science Vol 249, Springer Verlag, pp122-137, 1987.

BHR S D Brookes, C A R Hoare and A W Roscoe, A Theory of Communicating Sequential Processes. J ACM 31,3, pp 560-599, 1984.

BK J A Bergstra and J W Klop, Algebra of Communicating Processes with Abstraction. J TCS 37,1, pp77-121, 1985.

CH R Campbell and A Habermann, The Specification of Process Synchronisation by Path Expressions", in Lecture Notes in Computer Science Vol 16, Springer Verlag, 1974.

deN R de Nicola, *Testing Equivalences and Fully Abstract Models for Communicating Processes.* PhD Thesis, Report CST-36-85, Computer Science Department, University of Edinburgh, 1985.

deNH R de Nicola and M Hennessy, Testing Equivalences for Processes. J TCS 34,1, pp83-134, 1984.

deS R de Simone, Higher-level synchronising devices in MEIJE-SCCS, J TCS 37,3, pp245-267, 1985.

Hen M Hennessy, Acceptance Trees. J ACM 32,4, pp896-928, 1985.

Hoa1 C A R Hoare, Communicating Sequential Processes. C ACM 21,8, pp666-677, 1978.

Hoa2 C A R Hoare, *Communicating Sequential Processes.* Prentice-Hall, 1985.

HM M Hennessy and R Milner, Algebraic Laws for Nondeterminism and Concurrency, J ACM 32,1, pp 137-161, 1985.

Mil1 R Milner, *A Calculus of Communicating Systems.* Lecture Notes in Computer Science Vol 92, Springer Verlag, 1980.

Mil2 R Milner, Flowgraphs and Flow Algebra. J ACM 26,4,1979.

Mil3 R Milner, Calculi for Synchrony and Asynchrony. J TCS 25, pp267-310, 1983.

Mil4 R Milner, Lectures in a Calculus of Communicating Systems. In *Control Flow and Data Flow* (ed M Broy), Springer Study Edition, Springer Verlag, pp205-228, 1985.

Mil5 R Milner, A complete inference system for a class of regular behaviours. Journal of Computer and Systems Sciences, 28,3, pp439-466, 1983.

Mil6 R Milner, A Complete Axiomatisation for Observational Congruence of Finite State Behaviours. Report ECS-LFCS-86-8, Computer Science Dept, University of Edinburgh, 1986.

Mil7 R Milner, A Modal characterisation of observable machine-behaviour. Proc. 6th Colloquium on Trees in Algebra and Programming, Lecture Notes in Computer Science Vol 112, Springer Verlag, pp 25-34, 1981.

Mil8 R Milner, Process Constructors and Interpretations. Proc IFIP 86 Conference, Dublin, pp507-514, 1986.

NPW M Nielsen, G Plotkin and G Winskel, Petri Nets, Event Structures and Domains, J TCS 13, pp85-108, 1981.

Par D Park, Concurrency and Automata on Infinite Sequences. LNCS Vol 104, Springer Verlag, 1981.

Plo G Plotkin, A Structural Approach to Operational Semantics. Report DAIMI FN-19, Computer Science Dept, Aarhus University, 1981.

Rei W Reisig, *Petri Nets.* EATCS Monographs on Theoretical Computer Science, (eds W Brauer, G Rozenberg, A Salomaa), Springer Verlag, 1983.

Copyright © 1987, Laboratory for Foundations of Computer Science, University of Edinburgh. All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.