CMU-CS-83-139

SYNCHRONISATION TREES

Glynn Winskel
Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa.

June 1983

A version of this paper is to be published in a special ICALP 83 version of the journal "Theoretical Computer Science".

The research reported in this paper was supported in part by funds from the Computer Science Department of Carnegie-Mellon University, and by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539. The views and conclusions contained in it are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

SYNCHRONISATION TREES

by
Glynn Winskel
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract.

Synchronisation trees are a concrete underlying model for much of the work on concurrency. They are trees with labelled arcs; the nodes represent states, the arcs occurrences of events and their labels how the events can synchronise with other events in the environment. The many different ways in which events are allowed to synchronise are captured abstractly by the concept of a synchronisation algebra. It says which pairs of labelled events can combine to form an event of synchronisation and what label the synchronisation event carries. Synchronisation trees are trees with arcs labelled by elements of a synchronisation algebra. Our approach is based on a natural definition of morphism of trees which essentially expresses how the occurrence of events in in one process imply the synchronised occurrence of events in another. Well-known operations on trees arise as categorical constructions. For example a sum construction is a coproduct on synchronisation trees while many familiar parallel compositions of synchronisation trees are restrictions of the product in the underlying category of trees. The constructions are continuous with respect to a natural complete partial order structure on trees so one obtains denotational semantics as synchronisation trees to a wide range of parallel programming languages, based on the constructions with recursion, in a routine manner by varying the synchronisation algebra. Isomorphism of synchronisation trees induces a basic congruence on terms of the language. We present a complete proof system for the congruence restricted to non-recursive terms. The categories of trees are generalised to categories of transition systems. The pleasant categorical set-up which exists between the categories of trees and transition systems makes possible a smooth translation between operational semantics expressed in terms of transition systems and denotational semantics expressed in terms of trees.

0. Introduction.

We present a collection of categories of labelled trees useful in giving denotational semantics to parallel programming languages such as Milner's "Calculus of communicating Systems" , CCS [M1], his synchronous CCS, called SCCS [M2], and languages derived from Hoare's CSP as presented in [HBR] and [B]. Enough results are given to provide denotational semantics to any of the languages in [M1, M2, HBR] though at the rather basic level of labelled trees---called synchronisation trees in [M1].

Synchronisation trees are a basic, very concrete, interleaving model of parallel computation in which processes communicate by mutual synchronisation. A synchronisation tree is a tree in which the nodes represent states and the arcs represent event occurrences, labelled to show how they synchronise with events in the environment. Tree semantics arise naturally once concurrency is simulated by nondeterministic interleaving and for this reason synchronisation-tree semantics underlie much of the work on the semantics of synchronising processes. For example in [M1] it is made clear how every equivalence on CCS programs presented there factors through a synchronisation-tree semantics while [B] shows a similar result for the failure-set semantics in [HBR].

In order to cover a wide range of synchronisation disciplines between synchronising processes we express synchronisation disciplines between processes as synchronisation algebras. They are algebras on sets of labels which specify how pairs of labelled events combine to form a synchronisation event and what labels such combinations carry. They also specify what labelled events can occur asynchronously. The parallel composition is derived from a product in a category of trees; essentially one restricts the product of trees to those synchronised events allowed by the synchronisation algebra. By varying the synchronisation algebra we obtain many forms of parallel composition in the literature. Other useful operations are defined on synchronisation trees. They are all continuous with respect to a natural complete partial order of trees and so can be used to give denotations to processes defined recursively in terms of them by using least-fixed points--the standard tool of Scott-Strachey semantics.

The denotational semantics is related to operational semantics expressed in terms of labelled transition systems used in most of the work on CCS. In this framework recursion is often handled by introducing loops into the chains of state-to-state transitions. We define a category of transition systems whose product unfolds to the product of trees. Consequently one can define a parallel composition of labelled transition systems which unfolds to parallel composition of trees. Again this is so for a wide variety of synchronisation disciplines obtained by varying the synchronisation algebra.

There is a natural notion of equivalence on processes; two processes are equivalent if they are represented by isomorphic synchronisation trees. A complete set of proof rules are provided for this equivalence on a language of finite processes. Of course these rules will still be valid for any more abstract equivalence based on synchronisation trees. Unfortunately we do not consider proof rules for infinite processes or the important phenomenon of divergence (see *e.g.* [HN],[HP]).

Many of the results below follow from the paper and report [W1, W2] which however concentrated on showing how to use a broader framework of event structures [NPW1,2, W,W1] to give denotational semantics languages of synchronising processes like CCS. Event structures which include trees are closely related to Petri nets, reflect concurrency naturally and are not committed to interleaving. In [W1,W2] it is proved that by interleaving (or serialising) the labelled event structure denotation of a process one obtains its synchronisation-tree denotation. The papers [W1,W2,W3] provide a precise sense in which event structure models and Petri net models of communicating processes specialise down to an interleaving model based on synchronisation trees. In the special case of purely synchronous processes (for which the synchronisation algebra satisfies the synchronous law below) the event structure and tree semantics agree.

1. A category of trees.

Assume in any finite history a process can perform a sequence of events. Because a process need not be deterministic, such a sequence need not be extended in a unique way, but rather form a tree of sequences.

1.1 Definition. A *tree* is subset $T \subseteq A^*$ of finite sequences of some set A which satisfies

(i) $<> \in T$ and,

(ii) $< a_0, a_1, \ldots a_n, \ldots > \in T \Rightarrow < a_0, a_1, \ldots a_n > \in T$.

Remark. Condition (i) says a tree must always contain the null sequence $<>$, the root node. Condition (ii) says a tree is closed under the initial subsequence relation. To make the ideas as familiar as possible I have taken a different definition of trees from that given in [W1,W2]. However importantly all the categories here will be equivalent to the categories of the same name introduced in [W1,W2]. (Two categories are equivalent if their skeletal categories of isomorphism classes are isomorphic-see [Mac].)

1.2 Notation. Let T be a tree with $T \subseteq A^*$. We say T *is over* A iff every element of A is in some sequence of T. We shall often call elements of A *events.*

The following convention is very useful to avoid treating the null sequence $<>$ as a special case. Often we shall write a typical sequence as $< a_0, a_1, \ldots, a_{n-1} >$ where n is an integer representing the length of the sequence. We shall allow the length n to be 0 when by convention we agree that the above sequence represents $<>$.

Let s be a sequence $< a_0, a_1, \ldots, a_{n-1} >$ and t be a sequence $< b_0, b_1, \ldots, b_{m-1} >$. Write their concatenation as

$$st = < a_0, a_1, \ldots, a_{n-1}, b_0, b_1, \ldots, b_{m-1} > .$$

Let T be a tree. Let b be an element. By bT we mean the tree

$$bT = \{ <> \} \cup \{ < b > t \mid t \in T \}.$$

Let T be a tree. For $t, t' \in T$ write

$$t \longrightarrow_T t' \Leftrightarrow_{def} \exists a. t' = t < a > .$$

When we wish to highlight that an arc is associated with a particular event we draw the event above the arrow so:

$$t \xrightarrow{a}_T t' \Leftrightarrow t' = t < a > .$$

Clearly the elements T correspond to the nodes of a tree T while arcs correspond to pairs (t, t') where $t \longrightarrow_T t'$. The nodes are thought of as states of a process and the arcs as occurrences of events. A *morphism* from a tree S to a tree T shows the way in which the occurrence of an event of the process S implies the synchronised occurrence of an event in the process T. Formally it is a map on nodes which preserves the root–node and either preserves or collapses arcs. A special kind of morphism are the *synchronous morphisms* which always preserve arcs.

1.3 Definition. A *morphism* of trees from S to T is a map $f : S \to T$ such that

(i) $f(<>) = <>$ and,

(ii) $s \longrightarrow_S s' \Rightarrow f(s) = f(s')$ or $f(s) \longrightarrow_T f(s')$.

A *synchronous morphism* of trees from S to T is a map $f : S \to T$ such that

(i) $f(<>) = <>$ and,

(ii) $s \longrightarrow_S s' \Rightarrow f(s) \longrightarrow_T f(s')$.

Let $f : S \to T$ be a morphism of trees. Assume $s \longrightarrow_S s'$ in S, representing the occurrence of an event a of S so that $s' = s < a >$. If $f(s) \longrightarrow_T f(s')$ there is an event b such that $f(s') = f(s) < b >$. Intuitively the occurrence of the event a implies the occurrence of the event b, synchronised with that of a. If instead $f(s) = f(s')$ then the occurrence of a is not synchronised with an event occurrence in T. The latter possibility is disallowed for synchronous morphisms. We shall see that morphisms and synchronous morphisms give rise to a product and synchronous product of trees. Events of the products will essentially be pairs of events of the two trees, representing events of synchronisation between two processes. Their occurrence will project via tree morphisms to occurrences of component events in the constituent processes.

1.4 Proposition. *Trees with tree morphisms form a category with composition and identities those usual for functions. Similarly trees with synchronous morphisms form a subcategory.*

1.5 Definition. Let $\mathbf{Tr}$ be the category of trees with tree morphisms. Let $\mathbf{Tr}_{syn}$ be the subcategory of trees with synchronous morphisms.

Remark. The above categories are equivalent but not equal to the categories of the same name in [W1,W2].

2. Categorical constructions on trees.

Some major categorical constructions on **Tr** and $\mathbf{Tr}_{syn}$ are presented. The basic category theory used can be found in [AM] or [Mac].

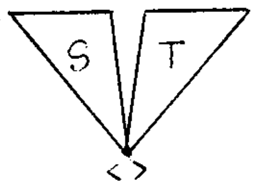
2.1 Definition. (Coproducts in Tr and $\mathbf{Tr}_{syn}$)
Let S and T be trees. Define

$$S+T = \{ < (0,a_0),\ldots,(0,a_{n-1}) > | < a_0,\ldots,a_{n-1} > \in S \} \cup \{ < (1,b_0),\ldots,(1,b_{n-1}) > | < b_0,\ldots,b_{n-1} > \in T \}.$$

Define the obvious injections $i_0 : S \to S+T$ and $i_1 : T \to S+T$ by

$$\begin{aligned} i_0 &: < a_0,\ldots,a_{n-1} > \mapsto < (0,a_0),\ldots,(0,a_{n-1}) > \\ i_1 &: < b_0,\ldots,b_{n-1} > \mapsto < (1,b_0),\ldots,(1,b_{n-1}) > \end{aligned}.$$

The coproduct construction just "glues" trees together at their roots, so:



2.2 Theorem. *The construction $S+T, i_0, i_1$ above is a coproduct of S and T in the categories* **Tr** *and* $\mathbf{Tr}_{syn}$.

Proof. Clearly $S+T$ is a tree and $i_0 : S \to S+T$ and $i_1 : T \to S+T$ are synchronous morphisms. In order for $S+T$, i_0, i_1 to be a coproduct in **Tr** we require that for arbitrary morphisms $j_0 : S \to U$ and $j_1 : T \to U$ to a tree U there is a unique morphism $j : S+T \to U$ such that the following diagram commutes:

$$\begin{tikzcd} & S+T \arrow[d, "j"] & \\ S \arrow[ur, "i_0"] \arrow[r, "j_0"'] & U & T \arrow[ul, "i_1"'] \arrow[l, "j_1"] \end{tikzcd}$$

This is clearly the case for j defined by:

$$j(v) = \begin{cases} j_0(< a_0,\ldots,a_{n-1} >) & \text{if } v = < (0,a_0),\cdots,(0,a_{n-1}) > \\ j_1(< b_0,\ldots,b_{n-1} >) & \text{if } v = < (1,b_0),\cdots,(1,b_{n-1}) >. \end{cases}$$

If j_0, j_1 are synchronous so is j. Consequently $S+T$, i_0, i_1 is a coproduct in $\mathbf{Tr}_{syn}$ too. ∎

2.3 Definition. (General coproducts)
Let $\{ T_i \mid i \in I \}$ be an indexed set of trees. Define their coproduct by

$$\sum_{i \in I} T_i = \bigcup_{i \in I} \{ < (i,a_0),\ldots,(i,a_{n-1}) > | < a_0,\ldots,a_{n-1} > \in T_i \}.$$

Define the obvious injections $in_i : T_i \to \sum_{i \in I} T_i$ by $in_i(< a_0,\ldots,a_{n-1} >) = < (i,a_0),\ldots,(i,a_{n-1}) >$ for $i \in I$.

When the indexed set I is null we understand $\sum_{i \in I} T_i = \sum \emptyset$ to be the null tree $\{ <> \}$.

2.4 Theorem. *The construction $\sum_{i \in I} T_i$, in_i for $i \in I$, above forms a coproduct of $\{ T_i \mid i \in I \}$ in the categories* $\mathbf{Tr}$ *and* $\mathbf{Tr}_{syn}$.

Proof. The proof is very similar to that of theorem 2.2. ∎

It is easier to define the product of trees in the category $\mathbf{Tr}_{syn}$ than the product in $\mathbf{Tr}$. We call the product in $\mathbf{Tr}_{syn}$ the *synchronous* product. The synchronous product of two trees basically "zips" their sequences together.

2.5 Definition. (Synchronous product in the category $\mathbf{Tr}_{syn}$)

Let S and T be trees. Define their *synchronous product* by

$$S \bigotimes T = \{ < (a_0, b_0), (a_1, b_1), \ldots, (a_{n-1}, b_{n-1}) > | < a_0, a_1, \ldots, a_{n-1} > \in S \;\&\, < b_0, b_1, \ldots, b_{n-1} > \in T \}.$$

Define projections $\pi_0 : S \bigotimes T \to S$ and $\pi_1 : S \bigotimes T \to T$ by

$$\pi_0 :< (a_0, b_0), \ldots, (a_{n-1}, b_{n-1}) > \mapsto < a_0, \ldots, a_{n-1} >,$$
$$\pi_1 :< (a_0, b_0), \ldots, (a_{n-1}, b_{n-1}) > \mapsto < b_0, \ldots, b_{n-1} > .$$

2.6 Theorem. *The construction $S \bigotimes T, \pi_0, \pi_1$ above is a product of S and T in the category* $\mathbf{Tr}_{syn}$.

Proof. Clearly $S \bigotimes T$ is a tree and $\pi_0 : S \bigotimes T \to S$ and $\pi_1 : S \bigotimes T \to T$ are synchronous morphisms. For $S \bigotimes T$, π_0, π_1 to be a product in $\mathbf{Tr}_{syn}$ we require the property: For arbitrary synchronous morphisms $f_0 : U \to S$ and $f_1 : U \to T$ from a tree U there is a unique synchronous morphism $f : U \to S \bigotimes T$ making the following diagram commute:

$$\begin{tikzcd}
& S \otimes T \arrow[ld, "\pi_0"'] \arrow[rd, "\pi_1"] & \\
S & & T \\
& U \arrow[lu, "f_0"] \arrow[ru, "f_1"'] \arrow[uu, "f"] &
\end{tikzcd}$$

Because f_0, f_1 are synchronous, for $u \in U$ the sequences $f_0(u)$ and $f_1(u)$ have the same length. Thus we can define

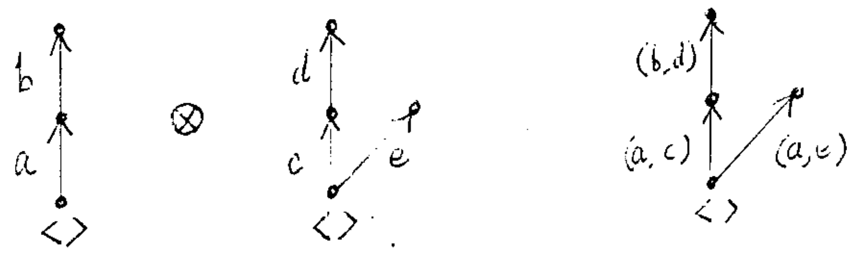
$$f(u) = < (a_0, b_0), \ldots, (a_{n-1}, b_{n-1}) >$$

where $f_0(u) = < a_0, \ldots, a_{n-1} >$ and $f_1(u) = < b_0, \ldots, b_{n-1} >$. Obviously $f : U \to S \bigotimes T$ is a synchronous morphism making the above diagram commute, and clearly it is the unique morphism doing so. ∎

2.7 Example.

$$\begin{tikzcd}
\langle a, b \rangle & & \langle c, d \rangle & & & \langle (a,c), (b,d) \rangle & \\
\langle a \rangle \arrow[u] & \otimes & \langle c \rangle \arrow[u] & \langle e \rangle & = & \langle (a,c) \rangle \arrow[u] & \langle (a,e) \rangle \\
\langle \, \rangle \arrow[u] & & \langle \, \rangle \arrow[u] \arrow[ru] & & & \langle \, \rangle \arrow[u] \arrow[ru] &
\end{tikzcd}$$

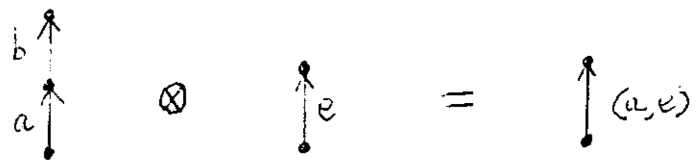
Or, labelling arcs by the events they are associated with we obtain:



For example

$$\pi_0(<(a,c),(b,d)>) = <a,b>$$
$$\pi_1(<(a,c),(b,d)>) = <c,d>$$

Notice how projections "unzip" sequences of pairs in the synchronous product. Clearly we have the following synchronous product



so projections need not be onto—consider the projection $\pi_0 : <(a,e)> \mapsto <a>$.

2.8 Notation. To give an explicit construction of a product in the category **Tr** we use partial functions. Represent undefined by the symbol $*$ and regard a partial function from A to B as a total function from A to $B \cup \{*\}$. Write a partial function, represented by $\theta : A \to B \cup \{*\}$, as $\theta : A \to_* B$—we shall always assume $* \notin B$ for such functions. Compose partial functions as follows: Let $\theta : A \to_* B$ and $\phi : B \to_* C$. Define their composition $\phi\theta : A \to_* C$ to be

$$\phi\theta(a) = \begin{cases} \phi(\theta(a)) & \text{if } \theta(a) \neq *, \\ * & \text{otherwise.} \end{cases}$$

Denote by **Set**$_*$ the category of sets (not containing $*$) with partial functions as morphisms. Now **Set**$_*$ itself has a useful product. The product in **Set**$_*$ of two sets A and B is given by

$$A \times_* B = \{(a,*) \mid a \in A\} \cup \{(a,b) \mid a \in A \;\&\; b \in B\} \cup \{(*,b) \mid b \in B\}$$

with projections $\rho_0 : A \times_* B \to A$ and $\rho_1 : A \times_* B \to B$ given by $\rho_i(x_0, x_1) = x_i$ for $i = 0, 1$.

We wish to extend a partial function $\theta : A \to_* B$ on sets to a function $\overline{\theta} : A^* \to B^*$ on sequences. So by induction on the length of sequences, we define

$$\overline{\theta}(<>) = <> \quad \text{and} \quad \overline{\theta}(<a>) = \begin{cases} <> & \text{if } \theta(a) = * \\ <\theta(a)> & \text{otherwise} \end{cases} \quad \text{for } a \in A,$$
$$\overline{\theta}(st) = (\overline{\theta}(s))(\overline{\theta}(t)) \quad \text{for } s, t \in A^*.$$

Now we define the product in **Tr** .

2.9 Definition. (Product in the category Tr)

Let S and T be trees. Assume S is over A and T is over B. Define $S \times T$ to consist of sequences over $A \times_* B$ which project via extensions of $\rho_0 : A \times_* B \rightarrow_* A$ and $\rho_1 : A \times_* B \rightarrow_* B$ to sequences in S and T as follows:

$$u \in S \times T \Leftrightarrow u \in (A \times_* B)^* \ \& \ \overline{\rho_0}(u) \in S \ \& \ \overline{\rho_1}(u) \in T.$$

Define *projections* $\pi_0 : S \times T \to S$ and $\pi_1 : S \times T \to T$ by taking $\pi_0(u) = \overline{\rho_0}(u)$ and $\pi_1(u) = \overline{\rho_1}(u)$ for $u \in S \times T$.

2.10 Theorem. *The construction $S \times T, \pi_0, \pi_1$ above is a product in the category* **Tr** .

Proof. Clearly $S \times T$ is a tree and $\pi_0 : S \times T \to S$ and $\pi_1 : S \times T \to T$ are morphisms. Assume $f_0 : U \to S$ and $f_1 : U \to T$ are morphisms from a tree U. We require that there is a unique morphism $f : U \to S \times T$ making the following diagram commute:

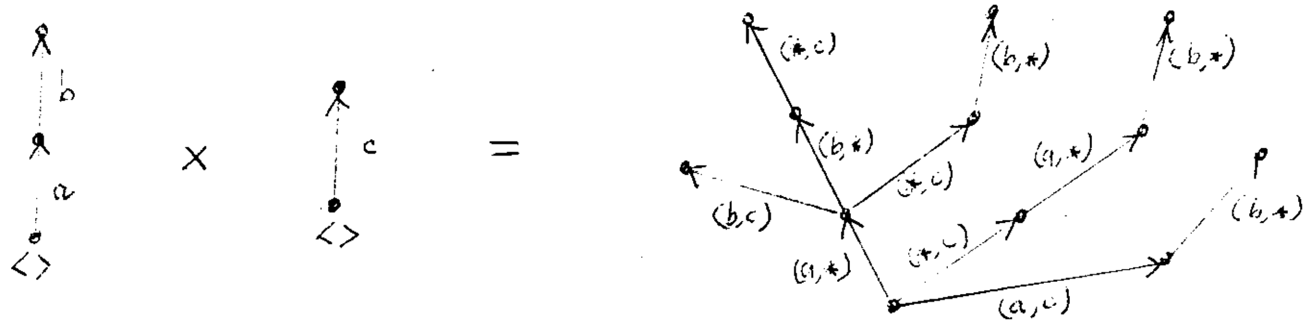
$$\begin{tikzcd} & S \times T \arrow[dl, "\pi_0"'] \arrow[dr, "\pi_1"] & \\ S & & T \\ & U \arrow[ul, "f_0"] \arrow[ur, "f_1"'] \arrow[uu, dashed, "f"] & \end{tikzcd}$$

Define $f(u)$ by induction on u:

$$f(u<e>) = \begin{cases} f(u) & \text{if } f_0(u<e>) = f_0(u) \text{ and } f_1(u<e>) = f_1(u) \\ f(u)<(a,*)> & \text{if } f_0(u<e>) = f_0(u)<a> \text{ and } f_1(u<e>) = f_1(u) \\ f(u)<(*,b)> & \text{if } f_0(u<e>) = f_0(u) \text{ and } f_1(u<e>) = f_1(u)<b> \\ f(u)<(a,b)> & \text{if } f_0(u<e>) = f_0(u)<a> \text{ and } f_1(u<e>) = f_1(u)<b>. \end{cases}$$

A simple induction on u shows that $\pi_j f(u) = f_j(u)$ for $j = 0, 1$. Obviously $f : U \to S \times T$ is a morphism. Assume $h : U \to S \times T$ is another morphism making the diagram commute. Another simple induction on sequences u shows $f(u) = h(u)$, establishing the uniqueness of f. Consequently $S \times T$, π_0, π_1 is a product in the category **Tr** . ∎

2.11 Example. We show the product of two simple trees. We label arcs by their associated events.



The projections π_0, π_1 act for example so that

$$\pi_0 :< (*,c),(a,*),(b,*) > \mapsto < a,b >$$
$$\pi_1 :< (*,c),(a,*),(b,*) > \mapsto < c >$$

Notice how the projections "unzip" sequences of pairs of events with $*$. By introducing $*$ we allow the possibility of asynchrony; events in the product of two trees are not forced to occur in step if they are to occur at all. Contrast the synchronous product.

In the categories **Tr** and $\mathbf{Tr}_{syn}$ there are pleasing relations between product and coproduct. This result indicates the relation between the parallel compositions of synchronisation trees (in *e.g.* [M1, B]) and the product of trees.

2.12 Theorem. *Let S and T be trees. Then*

$$S = \bigcup_{a \in A} aS_a \cong \sum_{a \in A} aS_a \quad \textit{and} \quad T = \bigcup_{b \in B} bT_b \cong \sum_{b \in B} bT_b$$

for some sets of events A and B and trees S_a and T_b indexed by $a \in A$ and $b \in B$ respectively. We have the following characterisation of the product of S and T:

$$\begin{aligned} S \times T &= \bigcup_{a \in A} (a,*)S_a \times T \cup \bigcup_{a \in A, b \in B} (a,b)S_a \times T_b \cup \bigcup_{b \in B} (*,b)S \times T_b \\ &\cong \sum_{a \in A} (a,*)S_a \times T + \sum_{a \in A, b \in B} (a,b)S_a \times T_b + \sum_{b \in B} (*,b)S \times T_b; \end{aligned}$$

and the following characterisation of their synchronous product:

$$S \bigotimes T = \bigcup_{a \in A, b \in B} (a,b)S_a \bigotimes T_b \cong \sum_{a \in A, b \in B} (a,b)S_a \bigotimes T_b.$$

Proof.

Clearly the tree $S = \bigcup_{a \in A} aS_a$, where $S_a = \{t \mid < a > t \in S\}$ for some subset A of events. As the sets aS_a are disjoint, $S \cong \sum_{a \in A} aS_a$. Similarly the tree $T = \bigcup_{b \in B} bT_b \cong \sum_{b \in B} bT_b$ for some subset B of events.

Let u be a sequence of events of the product which project via partial functions ρ_0, ρ_1 to events of S and T—we use the notation of definition 2.9. We have

$$\begin{aligned} u \in S \times T &\Leftrightarrow \overline{\rho_0}(u) \in S \;\&\; \overline{\rho_1}(u) \in T \\ &\Leftrightarrow u = \begin{cases} < (a,*) > u' & \text{for } a \in A \;\&\; \overline{\rho_0}(u') \in S_a \;\&\; \overline{\rho_1}(u') \in T \text{ or} \\ < (a,b) > u' & \text{for } a \in A \;\&\; b \in B \;\&\; \overline{\rho_0}(u') \in S_a \;\&\; \overline{\rho_1}(u') \in T_b \text{ or} \\ < (*,b) > u' & \text{for } b \in B \;\&\; \overline{\rho_0}(u') \in S \;\&\; \overline{\rho_1}(u') \in T_b \end{cases} \end{aligned}$$

This gives the above characterisation of the product. The characterisation of the synchronous product follows similarly. ∎

We define an operation of restriction in the next section. The synchronous product is a restriction of the product to those events with no undefined component (*i.e.* a component $*$). Parallel compositions will be defined as a restriction of the product. In fact the parallel composition of synchronisation trees appropriate to Milner's synchronous calculi will be a restriction of the synchronous product $\bigotimes$.

3. Complete partial orders of trees.

We consider two natural complete partial orderings on trees. One is based on the idea of restricting a tree to a subset of events—an operation natural in itself—and the other is just inclusion of trees. Our operations on trees will be continuous with respect to both orderings so we shall be able to define trees recursively following now standard lines—see *e.g.* [S]—by taking least fixed-points in either of the two cpo's.

3.1 Definition. (Restriction) Let T be a tree. Let B be a set. Define the *restriction* of T to B, written $T\lceil B$, by

$$t \in T\lceil B \Leftrightarrow t \in T \;\&\; t \in B^*.$$

In other words the restriction of a tree to a subset of events is just the subtree consisting of sequences in T for which all elements are in B. Restriction induces a partial order on trees; one tree is below another if it is a restriction of the other. This ordering makes a complete partial order (c.p.o.) of trees, apart from the fact that trees form a class and not a set. Of course there is another natural c.p.o. of trees induced by simple inclusion. All the above operations on trees are continuous with respect to the two c.p.o. structures.

3.2 Definition. Let S and T be trees over A and B respectively. Define

$$S \leq T \Leftrightarrow A \subseteq B \;\&\; S = T\lceil A.$$

3.3 Lemma. *Let S and T be trees over the same set of events. If $S \leq T$ then $S = T$.*

Proof. Assume S and T are both over the set of events A. Then $T = T \cap A^* = S$. ∎

3.4 Theorem.

(i) The relation $\leq$ is a partial order with least element the null tree, $\{ <> \}$. Let $T_0 \leq T_1 \leq \cdots \leq T_n \leq \cdots$ be an ω-chain of trees. Then it has a least upper bound $\bigcup_{n\in\omega} T_n$.

(ii) The null tree $\{ <> \}$ is the $\subseteq$-least tree i.e. for all trees T, $\{ <> \} \subseteq T$. Let $T_0 \subseteq T_1 \subseteq \cdots \subseteq T_n \subseteq \cdots$ be an ω-chain of trees. Then it has a least upper bound $\bigcup_{n\in\omega} T_n$.

Proof. (i) Obviously $S \leq S$ for any tree S so $\leq$ is reflexive. If $S \leq T \leq S$ then $S \subseteq T \subseteq S$ so $\leq$ is antisymmetric. If $S \leq T \leq U$, where S, T, U are trees over A, B, C respectively, then $S = T \cap A^* = U \cap B^* \cap A^* = U \cap A^*$ so $S \leq U$, making $\leq$ transitive. Thus $\leq$ is a partial order.

Clearly $\{ <> \} \leq T$, for all trees T.

Let $T_0 \leq T_1 \leq \cdots \leq T_n \leq \cdots$ be an ω-chain of trees T_n with T_n over events A_n. Then as $T_0 \subseteq T_1 \subseteq \cdots \subseteq T_n \subseteq \cdots$ we obtain that $T = \bigcup_{n\in\omega} T_n$ is a tree over $A = \bigcup_{n\in\omega} A_n$. By the following argument T is an upper bound of each T_n.

Suppose $t \in T \cap {A_n}^*$. Then $t \in T_m$ for some $m \geq n$. As $T_n \leq T_m$ we must have $t \in T_n$. Thus $T_n \leq T$ for every n, so T is an upper bound of $\{ T_n \mid n \in \omega \}$. Now we show that T is the least upper bound. Suppose $T_n \leq U$ for all n with U a tree. Clearly $T \subseteq U$. If $u \in U \cap A^*$ then $u \in {A_n}^*$ for some n. Hence as $T_n \leq U$ we have $u \in T_n$. So $u \in T$ too. This makes $T \leq U$ and so T is the least upper bound of $\{ T_n \mid n \in \omega \}$ with respect to $\leq$.

The remaining part, (ii), is obvious. ∎

3.5 Definition.

Say a unary operation operation op on trees is $\leq$-(respectively $\subseteq$-) *monotonic* iff $S \leq T \Rightarrow op(S) \leq op(T)$ (respectively $S \subseteq T \Rightarrow op(S) \subseteq op(T)$).

Say a unary operation operation op on trees is $\leq$-(respectively $\subseteq$-) *continuous* iff it is $\leq$-(respectively $\subseteq$-) monotonic and preserves least upper bounds of ω-chains of trees *i.e.* if $T_0 \leq T_1 \cdots T_n \leq \cdots$ (respectively $T_0 \subseteq T_1 \cdots T_n \subseteq \cdots$) is an ω-chain of trees then $op(\bigcup_{n\in\omega} T_n) = \bigcup_{n\in\omega} op(T_n)$.

If op is an n-ary operation on trees, say it is $\leq$-(respectively $\subseteq$-) *monotonic* iff it is monotonic in each argument separately. If op is an n-ary operation on trees, say it is $\leq$-(respectively $\subseteq$-) *continuous* iff it is continuous in each argument separately.

The next lemma provides useful necessary and sufficient conditions for an operation to be $\leq$-continuous; the operation should be $\leq$-monotonic and act continuously on the sets of events associated with trees, where the sets of events are ordered by inclusion.

3.6 Lemma. *Let op be a unary operation on trees. The operation op is $\leq$-continuous iff*

(i) the operation op is monotonic, and

(ii) if $T_0 \leq T_1 \cdots T_n \leq \cdots$ is a ω-chain of trees then the events of $op(\bigcup_{n\in\omega} T_n)$ are included in the events of $\bigcup_{n\in\omega} op(T_n)$.

Proof.

"$\Rightarrow$" Obvious.

"$\Leftarrow$" Suppose (i) and (ii) above. Let $T_0 \leq T_1 \leq \cdots \leq T_n \leq \cdots$ be a chain of trees such that each tree T_n is over events A_n. The chain has lub $\bigcup_n T_n$. By monotonicity $\bigcup_{n\in\omega} op(T_n)$ is a tree and $\bigcup_{n\in\omega} op(T_n) \leq op(\bigcup_{n\in\omega} T_n)$. From (ii) we know the trees $\bigcup_{n\in\omega} op(T_n)$ and $op(\bigcup_{n\in\omega} T_n)$ are over the same set of events. Thus by the above lemma they are equal. ∎

3.7 Theorem. *Each operation $T \mapsto bT$, $T \mapsto T\lceil B$, $+$, $\otimes$, $\times$, for an arbitrary element b and set B, is $\leq$-continuous and $\subseteq$-continuous. The operation of restriction is continuous on sets of events ordered by inclusion i.e. if T is a tree and if $B_0 \subseteq \cdots \subseteq B_n \subseteq \cdots$ is an ω-chain of sets then $T\lceil(\bigcup_{n\in\omega} B_n) = \bigcup_{n\in\omega}(T\lceil B_n)$.*

Proof. The continuity of these operations with respect to $\leq$ is best proved using lemma 3.6. Continuity with respect to $\subseteq$ is easier to show. We show only the continuity of $\times$ with respect to $\leq$. Assume S, S' and T are trees over A, A' and B respectively. Then $S \times T$, $S' \times T$ are over events $A \times_* B$ and $A' \times_* B$. Let $\rho_0 : A' \times B \rightarrow A'$ and $\rho_1 : A' \times B \rightarrow B$ be the partial functions projecting events in the product to their component events in S' and T respectively.

In showing monotonicity, by symmetry, it is sufficient to consider just one argument which we can assume to be the left. Suppose $S \leq S'$. We require $S \times T \leq S' \times T$. This follows by:

$$\begin{aligned} u \in S \times T &\Leftrightarrow u \in (A \times_* B)^* \,\&\, \overline{\rho_0}(u) \in S \,\&\, \overline{\rho_1}(u) \in T \\ &\Leftrightarrow u \in (A \times_* B)^* \,\&\, \overline{\rho_0}(u) \in S' \,\&\, \overline{\rho_1}(u) \in T \\ &\Leftrightarrow u \in (S' \times T)\lceil(A \times_* B). \end{aligned}$$

Now assume $S_0 \leq \cdots \leq S_n \leq \cdots$ is a chain of trees so that S_n is over events A_n. Let c be an event of $(\bigcup_{n\in\omega} S_n) \times T$. Then c has the form $(a, *)$, (a, b) or $(*, b)$. Thus c is an event of $\bigcup_{n\in\omega}(S_n \times T)$. Thus $\times$ is continuous in its first and, by symmetry, its second argument. Thus $\times$ is $\leq$-continuous.

The remaining proof is left to the reader. ∎

Consequently each of the above operations can be used in the recursive definition of trees.

4. Synchronisation algebras.

We shall label events of processes to specify how they interact with the environment. We shall obtain trees in which the arcs are labelled just like the synchronisation trees of CCS in [M1]. However our approach is more abstract. We shall label trees by elements of a *synchronisation algebra* which shows how labelled events synchronise with labelled events in the environment. Associated with any particular sychronisation algebra is a particular parallel composition of synchronisation trees. So, by specialising to particular synchronisation algebras we obtain Milner's parallel composition of synchronisation trees [M1], the parallel composition that underlies his synchronous calculi [M2], and the parallel compositions defined in [B] which underlie the parallel compositions on failure sets given in [HBR].

The intuitions behind synchronisation algebras are given in [W1,W2]. To recap, a synchronisation algebra is a binary, commutative, associative operation $\bullet$ on a set of labels which always includes two distinguished elements $*$ and 0. The binary operation $\bullet$ says how labelled events combine to form synchronisation events and what labels such combinations carry. No real events are ever labelled by $*$ or 0. However their introduction allows us to specify the way labelled events synchronise without recourse to partial operations on labels. (These two forms of undefined should not be confused with another "undefined" $\perp$ used in the theory of domains.)

The constant 0 is used to specify when sychronisations are disallowed. If two events labelled λ and λ' are not supposed to synchronise then their composition $\lambda \bullet \lambda'$ is 0. For this reason 0 does indeed behave like a zero with respect to the "multiplication" $\bullet$.

We have already seen the constant $*$ in the definition of product. Recall the partial functions ρ_0, ρ_1 which projected from the events in the product to events in one of the components. An event $(e_0, *)$ in the product $S \times T$ of trees S and T projected down to the event e_0 in S and the undefined "event" $* = \rho_1((e_0, *))$ in T. This meant the event e_0 of S occurred asynchronously, unsynchronised with any event of T. In a synchronisation algebra, the constant $*$ is used to specify when a labelled event can or cannot occur asynchronously. An event labelled λ can occur asynchronously iff $\lambda \bullet *$ is not 0. We insist that the only divisor of $*$ is $*$ itself, essentially because we do not want a synchronisation event to disappear. (The reader may find it helpful to glance ahead to the definition of parallel composition of synchronisation trees given in 6.8.)

4.1 Definition. A *synchronisation algebra* (S.A.) is an algebra $(L, \bullet, *, 0)$ where L is a set of *labels* so $L \setminus \{ *, 0 \} \neq 0$ and $\bullet$ is a binary commutative associative operation on L which satisfies

(i) $\forall \lambda \in L.\, \lambda \bullet 0 = 0$ and

(ii) $* \bullet * = *$ and $\forall \lambda, \lambda' \in L.\, \lambda \bullet \lambda' = * \Rightarrow \lambda = *$.

Synchronisation algebras have an obvious divisor relation which intuitively says when one labelled event can be a component of a synchronisation event.

4.2 Definition. Let $(L, \bullet, *, 0)$ be an S.A.. For $\lambda, \lambda' \in L$ define

$$\lambda \; div \; \lambda' \Leftrightarrow \lambda = \lambda' \text{ or } \exists \mu \in L.\, \lambda \bullet \mu = \lambda'.$$

When $\lambda \; div \; \lambda'$ we say "λ divides λ'".

4.3 Lemma. *Let $(L, \bullet, *, 0)$ be a synchronisation algebra. Then the following properties hold:*

(i) the constants $*$ and 0 are distinct,

(ii) the relation *div* is reflexive and transitive i.e. a preorder,

(iii) $\lambda \; div \; * \Rightarrow \lambda = *$,

(iv) $0 \; div \; \lambda \Rightarrow \lambda = 0$,

(v) $\alpha_0 \; div \; \beta_0 \; \& \; \alpha_1 \; div \; \beta_1 \Rightarrow (\alpha_0 \bullet \alpha_1) \; div \; (\beta_0 \bullet \beta_1)$.

Proof.

(i) We can take $\alpha \in L \setminus \{ *, 0 \}$. Then if $0 = *$ we would have $\alpha \bullet 0 = 0 = *$ which implies $\alpha = *$. This contradicts the choice of α making $0 \neq *$.

(ii) by associativity,

(iii) by property (ii) in the definition of synchronisation algebra,

(iv) as 0 is a zero,

(v) by commutativity and asscociativity. ∎

We might wish to specify that no event can occur asynchronously. An event will be labelled by a non-$*$, non-0 label so this can be specified by ensuring the composition of such labels with $*$ always gives 0. Milner's synchronous calculi [M2] fit into this scheme, as we shall see later in proposition 6.19. In 6.11, we shall make use of another law on synchronisation algebras. It expresses when $\bullet$ behaves like the least upper bound with respect to *div*, or, the same thing, when $\bullet$ is the operation of least common multiple (L.C.M.) for the "multiplication" $\bullet$.

4.4 Definition. Let $(L, \bullet, *, 0)$ be an S.A.. We say L is *synchronous* when it satisfies the law

$$\forall \lambda \in L \setminus \{ * \}. \lambda \bullet * = 0.$$

We say $(L, \bullet, *, 0)$ satisfies the L.C.M. law when

$$\forall \alpha, \beta, \gamma \in L. \alpha \; div \; \gamma \; \& \; \beta \; div \; \gamma \Rightarrow (\alpha \bullet \beta) \; div \; \gamma.$$

As examples and for future reference we now present some synchronisation algebras. We present the algebras in the form of multiplication tables. In fact the synchronisation algebras correspond to the parallel composition of CCS and the two forms of parallel composition in [HBR, B]. A full justification of these facts appears later. For the moment though, the reader can probably see what each synchronisation algebra is saying so we shall try to give the intuition. The tie-up with Milner's monoids and groups of actions for his synchronous calculi will be made later.

4.5 Example. (The synchronisation algebra for CCS [M1])

Pure CCS—no value passing: In CCS events are labelled by $\alpha, \beta, \cdots$ or by their complementary labels $\overline{\alpha}, \overline{\beta}, \cdots$ or by the label τ. The idea is that only two events bearing complementary labels may synchronise to form a synchronisation event labelled by τ. Events labelled by τ cannot synchronise further; in this sense they are invisible to processes in the environment, though their occurrence may lead to internal changes of state. All labelled events may occur asynchronously. Hence the synchronisation algebra for CCS takes the following form. We call the algebra L_1.

$\bullet$	$*$	α	$\overline{\alpha}$	β	$\overline{\beta}$	$\cdots$	τ	0
$*$	$*$	α	$\overline{\alpha}$	β	$\overline{\beta}$	$\cdots$	τ	0
α	α	0	τ	0	0	$\cdots$	0	0
$\overline{\alpha}$	$\overline{\alpha}$	τ	0	0	0	$\cdots$	0	0
β	β	0	0	τ	0	$\cdots$	0	0
.	.	.	.	.	.	$\cdots$	.	.

With value passing: Suppose values $v \in V$ are passed during synchronisation. Take labels of the form $*$, 0, αv (receiving a value v on line α) and $\overline{\alpha} v$ (sending a value v on line α) with a synchronisation algebra like that above but now with $\overline{\alpha} v$ the complement of αv. More precisely take $L_1(V)$ to be the synchronisation algebra $(L, \bullet, *, 0)$ where $L = (L_1 \setminus \{\tau, *, 0\} \times V) \cup \{\tau, *, 0\}$ with composition given by

$$\lambda \bullet \lambda' = \begin{cases} \tau & \text{if } \lambda = \alpha v \text{ and } \lambda' = \overline{\alpha} v, \\ \tau & \text{if } \lambda = \overline{\alpha} v \text{ and } \lambda' = \alpha v, \\ \lambda & \text{if } \lambda' = *, \\ \lambda' & \text{if } \lambda = *, \\ 0 & \text{otherwise.} \end{cases}$$

We shall see that $L_1(V)$ can be viewed as a simple quotient algebra of the (direct) product of two synchronisation algebras, one being L_1 and the other a straightforward extension of the set of values V to a synchronisation algebra.

4.6 Example. (The synchronisation algebra for || in [HBR, B]) In [HBR] and [B] events are labelled by $\alpha, \beta, \cdots$ or τ. For the parallel composition || in [HBR, B] events must "synchronise on" $\alpha, \beta, \cdots$. In other words non-τ-labelled events cannot occur asynchronously. Rather, an α-labelled event in one component of a parallel composition must synchronise with an α-labelled event from the other component in order to occur; the two events must synchronise to form a synchronisation event again labelled by α. The S.A. for this parallel composition takes the following form. We call the algebra L_2.

$\bullet$	$*$	α	β	$\cdots$	τ	0
$*$	$*$	0	0	$\cdots$	τ	0
α	0	α	0	$\cdots$	0	0
β	0	0	β	$\cdots$	0	0
.	.	.	.	$\cdots$	.	.

4.7 Example. (The synchronisation algebra for ||| in [HBR, B]) The parallel composition ||| in [HBR] and [B] is called the "interleaving" operation in [HBR, B]. The reason is that no synchronisations are allowed, but every event can occur asynchronously, so in the framework of [HBR, B] where processes are coerced so they perform only one event at a time the parallel composition ||| interleaves the sequences of events of the two component processes. Events are labelled exactly as they are for L_2 but the synchronisation algebra takes a different form, shown below. We call this algebra L_3.

$\bullet$	$*$	α	β	$\cdots$	τ	0
$*$	$*$	α	β	$\cdots$	τ	0
α	α	0	0	$\cdots$	0	0
β	β	0	0	$\cdots$	0	0
$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdots$	$\cdot$	$\cdot$

Of course synchronisation algebras can be viewed as standard algebras with an operation $\bullet$ and two constants $*$ and 0. Looked at in this way they come ready equipped with the usual definition of homomorphism (made to preserve the composition and the constants), and the attendant categorical constructions like (direct) product. But does this mathematical definition match the interpretation we put to the operation $\bullet$ and constants $*$ and 0? I think not, and tentatively suggest the following definitions are more suitable. They regard synchronisation algebras as *partial algebras* (see [Grä]) which have partial operations preserved by homomorphisms only when they are defined; think of composition as being undefined when it gives 0. Consequently 0 is preserved in rather a strict way. One class of homorphisms result if we impose a similar strict law for $*$—we call these *strict*—and another if we require simply that $*$ is preserved.

4.8 Definition. Let $A = (L_A, \bullet_A, *_A, 0_A)$ and $B = (L_B, \bullet_B, *_B, 0_B)$ be synchronisation algebras. A *homomorphism* of synchronisation algebras from A to B is a function $h : L_A \to L_B$ such that the following conditions hold:

(i) $\alpha \bullet \alpha' \neq 0 \Rightarrow h(\alpha \bullet_A \alpha') = h(\alpha) \bullet_B h(\alpha')$,

(ii) $h(\alpha) = 0_B \Leftrightarrow \alpha = 0_A$,

(iii) $h(*_A) = *_B$.

We say a homomorphism h is *strict* when $h(\lambda) = *_B \Leftrightarrow \lambda = *_A$.

4.9 Proposition. *Synchronisation algebras with homomorphisms form a category with composition the usual composition of functions and identity homomorphisms the identity functions. Synchronisation algebras with strict homomorphisms form a subcategory.*

Proof. We check the composition of homomorphisms is a homomorphism. Suppose $h : A \to B$ and $g : B \to C$ are homomorphisms. Assume $\alpha \bullet \alpha' \neq 0$ in A. Then $h(\alpha \bullet \alpha') = h(\alpha) \bullet h(\alpha') \neq 0$ in B. So $gh(\alpha \bullet \alpha') = g(h(\alpha) \bullet h(\alpha')) = gh(\alpha) \bullet gh(\alpha')$ in C. Clearly $gh(\alpha) = 0 \Leftrightarrow h(\alpha) = 0 \Leftrightarrow \alpha = 0$ and $gh(*) = g(*) = *$. Thus gh is a homomorphism. The remainder of the proof is left to the reader. ∎

4.10 Definition. Write SAfor the category of synchronisation algebras with homomorphisms.

We show the form of products in the category SAand its subcategory with strict homomorphisms. Products of synchronisation algebras provide one way to construct more complex algebras form more simple ones.

4.11 Definition. Let $A = (L_A, \bullet_A, *_A, 0_A)$ and $B = (L_B, \bullet_B, *_B, 0_B)$ be synchronisation algebras.

Define the *product of synchronisation algebras*, $A \times B$, to be $(L, \bullet, *, 0)$ given by

(i) $L = (L_A \setminus \{0_A\}) \times (L_B \setminus \{0_B\}) \cup \{(0_A, 0_B)\}$,

(ii) $(\alpha, \beta) \bullet (\alpha', \beta') = \begin{cases} (0_A, 0_B) & \text{if } \alpha \bullet \alpha' = 0_A \text{ or } \beta \bullet \beta' = 0_B, \\ (\alpha \bullet_A \alpha', \beta \bullet_B \beta') & \text{otherwise,} \end{cases}$

(iii) $* = (*_A, *_B)$ and $0 = (0_A, 0_B)$.

Define projection homomorphisms $h_A : A \times B \to A$, $h_B : A \times B \to B$, by $h_A(\alpha, \beta) = \alpha$ and $h_B(\alpha, \beta) = \beta$.

4.12 Definition. Let $A = (L_A, \bullet_A, *_A, 0_A)$ and $B = (L_B, \bullet_B, *_B, 0_B)$ be synchronisation algebras.
Define the *strict product of synchronisation algebras*, $A \otimes B$, to be $(L', \bullet, *, 0)$ given by

(i) $L' = (L_A \setminus \{ *_A, 0_A \}) \times (L_B \setminus \{ *_B, 0_B \}) \cup \{ (*_A, *_B), (0_A, 0_B) \}$,

(ii) $(\alpha, \beta) \bullet (\alpha', \beta') = \begin{cases} (0_A, 0_B) & \text{if } \alpha \bullet \alpha' = 0_A \text{ or } \beta \bullet \beta' = 0_B, \\ (\alpha \bullet_A \alpha', \beta \bullet_B \beta') & \text{otherwise,} \end{cases}$

(iii) $* = (*_A, *_B)$ and $0 = (0_A, 0_B)$.

Define projection homomorphisms $h'_A : A \times B \to A$, $h'_B : A \times B \to B$, by $h'_A(\alpha, \beta) = \alpha$ and $h'_B(\alpha, \beta) = \beta$.

Notice $A \otimes B$ has sort a subset of the sort of $A \times B$ and that it is closed under all the operation $\bullet$ of $A \times B$. It is subalgebra (of partial algebras) in the sense of [Grä]. It is also the restriction of the larger algebra to a subset, another way of constructing new synchronisation algebras from old.

4.13 Theorem. *Let A and B be synchronisation algebras. The construction $A \times B$, h_A, h_B is a categorical product of A and B in* **SA**. *The construction $A \otimes B$, h'_A, h'_B is a categorical product of A and B in the subcategory with strict homomorphisms.*

Proof. See the appendix. ∎

Another way to obtain new synchronisation algebras is to quotient by a congruence relation. A congruence relation on a synchronisation algebra is an equivalence relation $\equiv$ such that

$$\lambda \equiv \lambda' \;\&\; \mu \equiv \mu' \;\&\; \lambda \bullet \mu \neq 0 \;\&\; \lambda' \bullet \mu' \neq 0 \Rightarrow \lambda \bullet \mu \equiv \lambda' \bullet \mu'.$$

Given a synchronisation algebra and a congruence relation $\equiv$ the quotient consists of new labels the equivalence classes of $\equiv$ with $\bullet$–composition induced by the representatives. We illustrate how the synchronisation algebra for CCS with value passing arises as the quotient of a strict product. Firstly a non- null set of values V extends to a synchronisation algebra $V^\bullet$ with extra elements $*$ and 0 by taking $v \bullet v = v$ for $v \in V$ and $v \bullet * = * \bullet v = v$ for $v \in V \cup \{ * \}$ and $v \bullet 0 = 0 \bullet v = 0$ for $v \in V \cup \{ *, 0 \}$.

4.14 Proposition. *Let L_1 be the synchronisation algebra for CCS given in example 4.5. Let $h : L_1 \otimes V^\bullet \to L_1$ be the strict projection homomorphism from the strict product. Take the relation $\equiv$ on $L_1 \otimes V^\bullet$ to be given by:*

$$\lambda \equiv \lambda \Leftrightarrow h(\lambda) = h(\lambda') = \tau.$$

Then $\equiv$ is a congruence relation and the quotient $(L_1 \otimes V^\bullet)/\equiv$ is isomorphic to $L_1(V)$ the synchronisation algebra for CCS with value passing given in example 4.5.

Of course one can specify that more complicated operations are performed on values than just send and receive.

We stress that the definitions of homomorphisms on synchronisation algebras are tentative. Constructions like $\otimes$ on synchronisation algebras appear useful but may not be as general as one would like. The axioms on synchronisation algebras arose by considering an abstract way to formalise the range of synchronisation disciplines between labelled events. Possibly there is a class of algebras for specifying how processes are connected, or linked, together. That the physical linkage can be quite complicated and yet still be highly structured is demonstrated in [CP]. Typically processes may be linked by abstract channels or physical wires connected to linkage points or *ports* of the processes. To specify how they are linked by channels or wires the ports are assigned names or labels; perhaps ports to be linked carry the same label, as in [Mi], or complementary labels as in [M1]. An algebra on these labels might specify the geometric layout of the processes, how the processes are physically linked or wired together. But then along the channels or wires values may meet and interact; for example in hardware the values may be voltage contributions due to processes wired together. The interaction of these values might be specified by a synchronisation algebra.(The table giving this interaction in hardware is generally called the logic—it may be Boolean, have undefined values, floating values, strong and weak values etc..) Such processes interact through the synchronisation of events, where an event is a value at a port. Of course only events which are physically linked can interact. When they do the resultant value communicated will be determined by the component values. This suggests that the synchronisation algebra associated with processes should be a product of the "linkage algebra" and the synchonisation algebra of values. At present this is rather speculative but it does suggest we explore a wider class of algebras and, from our experience with synchronisation algebras, that the algebras should be partial.

5. Synchronisation trees.

A synchronisation tree is a tree with arcs labelled by elements of synchronisation algebra. It is convenient to label arcs via the underlying events from which the tree is built.

5.1 Definition. Let L be a synchronisation algebra. An L*-synchronisation tree* is a pair (T, l) where T is a tree over A and $l : A \to L \setminus \{*, 0\}$.

5.2 Notation. Let (T, l) be an L-synchronisation tree. Write $t \xrightarrow{\lambda} t'$ when $t \to t'$ and $l(a) = \lambda$ for the unique a such that $t' = t < a >$.

Frequently we shall omit the prefix "L–" when discussing synchronisation trees. When it is important the appropriate synchronisation algebra should be clear from the context.

We produce a category of synchronisation trees by restricting the tree-morphisms in accord with the synchronisation algebra. We insist the label of the image of an arc should divide the label of the arc because the image of an event is imagined to be a component of the event. Of course an arc may be collapsed in the image corresponding to the intuition that the event is not synchronised with any event of the image. But then we insist $*$ divides the original label.

5.3 Definition. Let L be a synchronisation algebra. Define an L*-morphism* of L-synchronisation trees from (S, l_S) to (T, l_T) to be a map $f : S \to T$ such that

$$f(<>) = <> \quad \text{and}$$

$$s \xrightarrow{\lambda} s' \Rightarrow (f(s) = f(s') \,\&\, * \; div\; \lambda) \text{ or } (f(s) \xrightarrow{\lambda'} f(s') \,\&\, \lambda' \; div\; \lambda).$$

5.4 Proposition. *Let L be a synchronisation algebra. Then L-synchronisation trees with L-morphisms form a category under the usual function composition and with the usual identity functions.*

Let (S, l_S) and (T, l_T) be two L-synchronisation trees. Then (S, l_S) and (T, l_T) are isomorphic in this category iff there is a bijection $f : S \to T$ such that

$$s \longrightarrow s' \Leftrightarrow f(s) \longrightarrow f(s')$$

and such that labels of corresponding arcs divide each other.

In particular, if div is an antisymmetric relation on L (i.e. $\lambda \; div\; \lambda' \; div\; \lambda \Rightarrow \lambda = \lambda'$) then (S, l_S) and (T, l_T) are isomorphic iff there is a bijection $f : S \to T$ such that

$$s \xrightarrow{\lambda} s' \Leftrightarrow f(s) \xrightarrow{\lambda} f(s').$$

Proof. That L-synchronisation trees with L-morphisms, for a synchronisation algebra L, form a category follows routinely from the facts that **Tr** is a category and div is a reflexive transitive relation on labels. The characterisations of isomorphism follow directly from the definition of L-morphism. ∎

5.5 Definition. Write $\mathbf{Tr}_L$ for the category of L-synchronisation trees with L-morphisms.

Remark. Note this category is equivalent but not equal to the category $\mathbf{Tr}_L$ in [W1, W2].

5.6 Proposition. *Let L be a synchronisation algebra. If $f : (S, l_S) \to (T, l_T)$ is an L-morphism of synchronisation trees then $f : S \to T$ is a morphism of trees. Assume that L is synchronous, so $\lambda \bullet * = 0$ for all $\lambda \in L \setminus \{*\}$. Then for any L-morphism $f : (S, l_S) \to (T, l_T)$ the map $f : S \to T$ is a synchronous morphism of trees.*

Proof. Clearly if L is synchronous $* \not{div}\ \lambda$ for any label $\lambda \in L \setminus \{*, 0\}$. Thus L-morphisms cannot collapse arcs. ∎

Thus we see how assumptions made on the synchronisation algebra influence the morphisms we allow. In fact, particular synchronisation algebras give us categories isomorphic to **Tr** and $\mathbf{Tr}_{syn}$.

5.7 Proposition. *Let A and S be the synchronisation algebras given by:*

A

$\bullet_A$	$*$	T	0
$*$	$*$	T	0
T	T	T	0
0	0	0	0

S

$\bullet_S$	$*$	T	0
$*$	$*$	0	0
T	0	T	0
0	0	0	0

Then $\mathbf{Tr}_A \cong \mathbf{Tr}$ *and* $\mathbf{Tr}_S \cong \mathbf{Tr}_{syn}$.

Proof. Because $*\ div\ T$ in A morphisms may collapse arcs while in S, because $* \not{div}\ T$, they must be preserved. ∎

6. Operations on synchronisation trees.

Assume $(L, \bullet, *, 0)$ is a synchronisation algebra. Define the following operations on (L-)synchronisation trees.

6.1 Definition. (Lifting)

Let $\lambda \in L \setminus \{*, 0\}$ and (T, l) be a synchronisation tree. Define $\lambda(T, l)$ to be the synchronisation tree (T', l') where

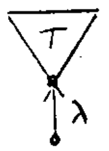
$$t \in T' \Leftrightarrow t = < > \text{or } t = < (0, \lambda), (1, a_0), \cdots, (1, a_{n-1}) >$$

for some $< a_0, \ldots, a_{n-1} > \in T$, and the new labelling function acts so $l'((0, \lambda)) = \lambda$ and $l'((1, a)) = l(a)$.

Extend lifting to morphisms as follows: Assume $f : (T, l_T) \to (T', l'_T)$ is a morphism of synchronisation trees and $\lambda \in L \setminus \{*, 0\}$. Define $\lambda f : \lambda(T, l_T) \to \lambda(T', l'_T)$ by

$$(\lambda f)(t) = \begin{cases} < > & \text{if } t = < > \\ < (0, \lambda), (1, b_0), \ldots, (1, b_{m-1}) > & \text{if } t = < (0, \lambda), (1, a_0), \ldots, (1, a_{n-1}) > \\ & \quad \& \ f(< a_0, \ldots, a_{n-1} >) = < b_0, \ldots, b_{m-1} >. \end{cases}$$

The process represented by λT must first do a λ-labelled event before becoming the process represented by a copy of T. In pictures we can draw lifting so:



6.2 Theorem. *Let $\lambda \in L \setminus \{*, 0\}$. The operation of lifting is a functor $\lambda : \mathbf{Tr}_L \to \mathbf{Tr}_L$.*

Proof. Obvious. ∎

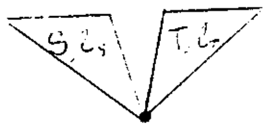
6.2 Definition. (Sum)
Let (S, l_S) and (T, l_T) be synchronisation trees. Define their *sum* by

$$(S, l_S) + (T, l_T) = (S + T, l)$$

where

$$l(c) = \begin{cases} l_S(a) & \text{if } c = (0, a), \\ l_T(b) & \text{if } c = (1, b). \end{cases}$$

The sum just sticks trees together at their roots. We can draw the sum $(S, l_S) + (T, l_T)$ so:



6.3 Definition. (Indexed Sum)
Let (T_i, l_i) be a set of synchronisation trees indexed by $i \in I$. Define their *sum* by

$$\sum_{i \in I}(T_i, l_i) = (\sum_{i \in I} T_i, l)$$

where $l(c) = l_i(a)$ if $c = (i, a)$ for $i \in I$.

Sum has obvious injection morphisms such that it is a coproduct in the category of synchronisation trees. Consequently the construction will extend naturally to a functor.

6.4 Theorem. *Let (S, l_S) and (T, l_T) be L-synchronisation trees. Let $i_0 : S \to S+T$ and $i_1 : T \to S+T$ be the injections—as given in the definition of coproduct. Then i_0, i_1 are L-morphisms and $(S, l_S)+(T, l_T), i_0, i_1$ is a coproduct in the category $\mathbf{Tr}_L$ of synchronisation trees.*

Similarly, $\sum_{i \in I}(T_i, l_i)$ with injections in_i for $i \in I$ is a coproduct where (T_i, l_i) is an I-indexed set of synchronisation trees with injections $in_i : (T_i, l_i) \to \sum_{i \in I}(T_i, l_i)$—as given in the definition of indexed coproduct. ∎

Proof. These properties follow from the corresponding properties in the underlying category of trees.

6.5 Definition. (Restriction)
Let $\Lambda \subseteq L \setminus \{*, 0\}$ satisfy the property: $\lambda \in \Lambda \;\&\; \lambda \; div \; \lambda' \;\&\; \lambda' \; div \; \lambda \Rightarrow \lambda' \in \Lambda$. Let (T, l) be a synchronisation tree over A. Define

$$(T, l)\lceil\Lambda = (T\lceil B, l')$$

where

$$B = \{ b \in A \mid l(b) \in \Lambda \} \quad \text{and} \quad l'(b) = l(b) \quad \text{for } b \in B.$$

The operation $(T,l)\lceil A$ restricts events to those which are labelled by elements of A. There are several alternative definitions of restriction in the literature [M1, M2, HBR, B]. Ours is chosen to be general and such that it still preserves isomorphism; it is like that in [M2]. I do not know how to extend restriction to a functor in a natural way. (At some cost in artificiality restriction can be presented as an equaliser.)

6.6 Definition. (Relabelling)

Let $\Xi : L \to L$ be a strict homomorphism of the synchronisation algebra L. Let (T,l) be a synchronisation tree. Define $(T,l)[\Xi] = (T, \Xi l)$.

For $\Xi : L \to L$ a strict homomorphism, extend relabelling to morphisms as follows: Assume $f : (S,l_S) \to (S', l'_S)$ is a morphism of synchronisation trees. Define $f[\Xi] : (S,l_S)[\Xi] \to (S',l'_S)[\Xi]$ by $(f[\Xi])(s) = f(s)$.

We have chosen this definition of relabelling because it extends to a functor on $\mathbf{Tr}_L$. (Of course there are other possible definitions which are also continuous with respect to $\leq_L$ given below. One example is the make-α-labels-into-τ-labels definition of hiding given in [HBR, B].)

6.7 Theorem. *Let $\Xi : L \to L$ be a strict homomorphism on the synchronisation algebra L. The operation $[\Xi] : \mathbf{Tr}_L \to \mathbf{Tr}_L$ is a functor on synchronisation trees.*

Proof. Recall what it means for Ξ to be a strict homomorphism on L: that $\Xi : L \to L$ and Ξ preserves $\bullet$, $*$, $\mathbf{0}$ and

$$\forall \lambda \in L.(\Xi(\lambda) = 0 \Rightarrow \lambda = 0) \;\&\; (\Xi(\lambda) = * \Rightarrow \lambda = *).$$

These properties ensure $T[\Xi]$ is a synchronisation tree for a synchronisation tree T. Because λ' *div* $\lambda \Rightarrow \Xi(\lambda')$ *div* $\Xi(\lambda)$ the map $[\Xi]$ produces morphisms from morphisms. Thus it is clearly a functor. ∎

6.8 Definition. (Parallel Composition)

Let (S,l_S) and (T,l_T) be synchronisation trees. Assume S is over A and T is over B. Then $S \times T$ is over $A \times_* B$, the product in $\mathbf{Set}_*$ with projections $\rho_0 : A \times_* B \to A$ and $\rho_1 : A \times_* B \to B$. Define the parallel composition of (S,l_S) and (T,l_T) by

$$(S,l_S) \textcircled{L} (T,l_T) = (S \times T\lceil C, l)$$

where

$$C = \{\, c \in A \times_* B \mid l_S\rho_0(c) \bullet l_T\rho_1(c) \neq 0 \,\} \quad \text{and} \quad l(c) = l_S\rho_0(c) \bullet l_T\rho_1(c).$$

Note we assume that the projection function compositions occur in $\mathbf{Set}_*$; so if, for example, $\rho_0(c) = *$ then $l_S\rho_0(c) = *$.

Extend $\textcircled{L}$ to morphisms as follows. Let $f : (S,l_S) \to (S', l_{T'})$ and $g : (T,l_T) \to (T', l_{T'})$ be two morphisms in $\mathbf{Tr}_L$. Define $f \textcircled{L} g = f \times g$, the image of f and g under the product functor $\times$ on $\mathbf{Tr}$.

In fact this definition makes $\textcircled{L}$ into a functor.

6.9 Theorem. *The operation $\textcircled{L}$ is a functor $\textcircled{L}$: $\mathbf{Tr}_L{}^2 \to \mathbf{Tr}_L$ on synchronisation trees.*

Proof. Let $f : S \to S'$ and $g : T \to T'$ be L-morphisms. We show by induction on the length of u' that if $u \xrightarrow{\lambda} u'$ in $S \textcircled{L} T$ then $f \times g(u') \in S' \textcircled{L} T'$ and $f \times g(u) = f \times g(u')$ & $*$ *div* λ or $f \times g(u) \xrightarrow{\lambda'} f \times g(u')$ & λ' *div* λ. It follows that $f \textcircled{L} g : S \textcircled{L} T \to S' \textcircled{L} T'$ is a morphism.

Either (a) $f \times g(u) = f \times g(u')$ or (b) $f \times g(u) \longrightarrow f \times g(u')$. If (a) then $f(u) = f(u')$ (and $g(u) = g(u')$) so $*$ *div* λ. Otherwise (b), in which case let c be the unique event such that $u < c >= u'$. Write its component-events in S and T as c_0 and c_1 respectively—one of c_0 and c_1 may be $*$. Let $l_S(c_0) = \lambda_0$ and $l_T(c_1) = \lambda_1$. Similarly let c' be the unique event of $S' \times T'$ such that $(f \times g(u)) < c' >= f \times g(u')$. Assume the component events of c' have labels λ'_0 and λ'_1 in S' and T' respectively. As f and g are L-morphisms λ'_0 *div* λ_0 and λ'_1 *div* λ_1. By lemma 4.2(v) we obtain $\lambda' = \lambda'_0 \bullet \lambda'_1$ *div* $\lambda_0 \bullet \lambda_1 = \lambda$. Thus $\lambda'_0 \bullet \lambda'_1 \neq 0$ by lemma 4.2(iv) so c' is an event of $S' \textcircled{L} T'$. Inductively this ensures that $f \times g(u') \in S' \textcircled{L} T'$ and clearly $f \times g(u) \xrightarrow{\lambda'} f \times g(u')$ & λ' *div* λ.

Thus $\textcircled{L}$ takes L-morphisms to L-morphisms. Its functorial properties follow from those of $\times$ in the underlying category of trees. ∎

Thus apart from restriction all the above operations extend to functors on $\mathbf{Tr}_L$ in an obvious way.

Generally the parallel composition of synchronisation trees is defined recursively—see *e.g.* [M1, B]. Instead we can give a recursive characterisation of our definition of parallel composition, which fortunately agrees with those in the literature when we specialise to particular synchronisation algebras. Because here we serialise all event occurrences, parallel composition, like product, can be expressed as an indexed sum.

6.10 Theorem. *Let S an T be L-synchronisation trees. Then*

$$S \cong \sum_{i \in I} \lambda_i S_i \quad \text{and} \quad T \cong \sum_{j \in J} \mu_j T_j$$

for some indexed sets of labels and synchronisation trees. Moreover, the parallel composition of S and T can be characterised as follows:

$$S \textcircled{L} T \cong \sum_{\lambda_i \bullet * \neq 0} (\lambda_i \bullet *)(S_i \textcircled{L} T) + \sum_{\lambda_i \bullet \mu_j \neq 0} (\lambda_i \bullet \mu_j)(S_i \textcircled{L} T_j) + \sum_{* \bullet \mu_j \neq 0} (* \bullet \mu_j)(S \textcircled{L} T_j).$$

Proof. This follows from theorem 2.12 and definition 6.8. ∎

The above result means we can show how by specialising to particular synchronisation algebras we obtain various parallel compositions of synchronisation trees present in the literature. Before this we pause to show how parallel composition relates to product in the categories of synchronisation trees. Although there are obvious projection functions, parallel composition does not always coincide with product. It does however when the operation $\bullet$ in the algebra behaves like the least common multiple (L.C.M.) operation, defined in 4.3.

6.11 Theorem. *Let (S, l_S) and (T, l_T) be two L-synchronisation trees over A and B respectively. Let $\pi'_0 = \pi_0 \lceil (S \textcircled{L} T)$ and $\pi'_1 = \pi_1 \lceil (S \textcircled{L} T)$ be the obvious restrictions of the projections $\pi_0 : S \times T \to S$ and $\pi_1 : S \times T \to T$ to the parallel composition. Then $S \textcircled{L} T, \pi'_0, \pi'_1$ is a product in the category $\mathbf{Tr}_L$ if*

$$\forall \gamma \in L \forall \alpha \in l_S A \forall \beta \in l_T B . \alpha \; div \; \gamma \; \& \; \beta \; div \; \gamma \Rightarrow (\alpha \bullet \beta) \; div \; \gamma.$$

It follows that parallel composition is always a categorical product in $\mathbf{Tr}_L$ iff the synchronisation algebra satisfies

$$\forall \alpha, \beta, \gamma \in L . \alpha \; div \; \gamma \; \& \; \beta \; div \; \gamma \Rightarrow (\alpha \bullet \beta) \; div \; \gamma.$$

Proof.

Let (S, l_S), (T, l_T) be two L-synchronisation trees and $\pi'_0 : (S, l_S) \textcircled{L} (T, l_T) \to (S, l_S)$ and $\pi'_1 : (S, l_S) \textcircled{L} (T, l_T) \to (T, l_T)$ be restrictions of the projections $\pi_0 : S \times T \to S$ and $\pi_1 : S \times T \to T$ in $\mathbf{Tr}$.

Suppose $\forall \gamma \in L \forall \alpha \in l_S A \forall \beta \in l_T B . \alpha \; div \; \gamma \; \& \; \beta \; div \; \gamma \Rightarrow (\alpha \bullet \beta) \; div \; \gamma$. Assume $f_0 : (U, l_U) \to (S, l_S)$ and $f_1 : (U, l_U) \to (T, l_T)$ are L-morphisms. Let $h : U \to S \times T$ be the unique morphism of trees such that

$\pi_0 h = f_0$ and $\pi_1 h = f_1$. We show h is an L-morphism, $h : (U, l_U) \to (S, l_S) \textcircled{L} (T, l_T)$. Then h is certainly the unique L-morphism such that $\pi'_0 h = f_0$ and $\pi'_1 h = f_1$.

Clearly $h(<>) = \{<>\}$. We show by induction on the length of $u' \in U$ that if $u \xrightarrow{\gamma} u'$ then $h(u') \in S \textcircled{L} T$ and $h(u) = h(u')$ & $* \; div \; \gamma$ or $(h(u) \xrightarrow{\delta} h(u')$ & $\delta \; div \; \gamma)$. It follows that h is an L-morphism.

Suppose $u \xrightarrow{\gamma} u'$. If $h(u) = h(u')$ then $f_0(u) = f_0(u')$ and $* \; div \; \gamma$. Otherwise $h(u) < c >= h(u')$ for some event $c = (a, b)$ of the product. As f_0 and f_1 are L-morphisms, $\alpha = l_S(a) \; div \; \gamma$ and $\beta = l_T(b) \; div \; \gamma$. (We allow a, b to be $*$ in which case the labelling is $*$.) By assumption $\alpha \bullet \beta \; div \; \gamma$ so $\alpha \bullet \beta \neq 0$. This makes c an event of the parallel composition. Thus $h(u') \in S \textcircled{L} T$, completing the induction.

Suppose L satisfies the LCM law. Then by the previous argument $S \textcircled{L} T$, π'_0, π'_1 is the product of synchronisation trees S, T in the category $\mathbf{Tr}_L$. Conversely suppose for arbitrary synchronisation trees S, T we have $S \textcircled{L} T$, π'_0, π'_1 is a product in $\mathbf{Tr}_L$. Suppose $\alpha \; div \; \gamma$ and $\beta \; div \; \gamma$ in L. Clearly if $\gamma = 0$ or $\alpha = \beta = *$ then $\alpha \bullet \beta \; div \; \gamma$ so assume $\gamma \neq 0$ and $\neg(\alpha = \beta = *)$. Suppose $\alpha = *$ (so $\beta \neq *$). Take S to be the null tree and T to be the synchronisation tree consisting of a single β-labelled arc. Let U be the synchronisation tree consisting of a single arc labelled by γ. Take $f_0 : U \to S$ to be the unique morphism to the null tree and $f_1 : U \to T$ to be the unique arc preserving morphism. A unique morphism $h : U \to S \textcircled{L} T$ exists such that $\pi'_0 h = f_0$ and $\pi'_1 = f_1$. Thus $\alpha \bullet \beta \; div \; \gamma$. If $\alpha \neq *$ and $\beta \neq *$ then taking S to consist of a single arc labelled by α and T to consist of a single arc labelled by β a similar argument shows $\alpha \bullet \beta \; div \; \gamma$. ∎

Let us run through, in a series of propositions, some parallel compositions in the literature. We refer to the synchronisation algebras L_1, L_2, L_3 of the earlier examples—4.5, 4.6, 4.7.

6.12 Proposition. **(Parallel composition in CCS)** *Let L_1 be the synchronisation algebra for CCS presented above. Write the parallel composition $\textcircled{L}$ as $|$, as in [M1]. Then two L_1-synchronisation trees*

$$S \cong \sum_{i \in I} \lambda_i S_i \quad \text{and} \quad T \cong \sum_{j \in J} \mu_j T_j$$

have a parallel composition given by

$$S \mid T \cong \sum_i \lambda_i (S_i \mid T) + \sum_{\lambda_i = \overline{\mu}_j \text{ or } \mu_j = \overline{\lambda}_i} \tau (S_i \mid T_j) + \sum_j \mu_j (S \mid T_j).$$

Because, for instance, $\alpha \; div \; \tau$ yet $0 = \alpha \bullet \alpha$ and $0 \not{div} \; \tau$ the parallel composition $|$ for CCS does not coincide with product in the category of synchronisation trees.

A similar proposition holds for the synchronisation algebra of CCS with value passing—recall the synchronisation algebra in example 4.5 two processes synchronise iff one sends and the other receives a common value on the same line.

Now we examine the parallel compositions $||$ and $|||$ given in [B] to support the failure set semantics in [HBR]. Here $||$ only coincides with product in the appropriate category of synchronisation trees if no events in the components are labelled by τ.

6.13 Proposition. **(Parallel composition $||$ in [B])** *Let L_2 be the synchronisation algebra presented above. Write the parallel composition $\textcircled{L}$ as $||$, as in [B]. Then two L_2-synchronisation trees*

$$S \cong \sum_i \lambda_i S_i + \sum_k \tau S_k \quad \text{and} \quad T \cong \sum_j \lambda_j T_j + \sum_l \tau T_l,$$

where λ_i, λ_j are non-τ labels, have a parallel composition given by

$$S \parallel T \cong \sum_{i,j:\lambda_i=\lambda_j} \lambda_i(S_i \parallel T_j) + \sum_k \tau(S_k \parallel T) + \sum_l \tau(S \parallel T_l).$$

The synchronisation algebra does not satisfy the L.C.M. law above because for instance τ div τ and yet $\tau \bullet \tau = 0 \not{div}\, \tau$. However for trees without τ-labels $\parallel$ coincides with product in the category of L_2-synchronisation trees.

6.14 Proposition. The parallel composition $|||$ in [B]: *Let L_3 be the synchronisation algebra presented above. Write the parallel composition $\textcircled{L}$ as $|||$, as in [B]. Then two L_3-synchronisation trees*

$$S \cong \sum_{i\in I} \lambda_i S_i \quad \text{and} \quad T \cong \sum_{j\in J} \mu_j T_j$$

have a parallel composition given by

$$S \,|||\, T \cong \sum_i \lambda_i(S_i \,|||\, T) + \sum_j \mu_j(S \,|||\, T_j).$$

For L_3 we have α div α and yet $\alpha \bullet \alpha = 0$ so ($\alpha \bullet \alpha \not{div}\, \alpha$). Therefore $|||$ does not coincide with product in the category of L_3-synchronisation trees.

The papers [HBR] and [B] contain another operation $\Box$ called "conditional composition" which can also be thought of as a parallel composition. The idea is that both components of a conditional composition can proceed independently performing τ-labelled events until one component makes a communication with the environment—performs a non-τ labelled event—when future communication must henceforth be with that component. There are two choices for the subsequent behaviour of the other component: one is that it may continue to perform τ-events (the idea in [HBR]) and another that even these invisible events are stopped (the idea in [B]). From the point of view of the failure-set equivalence in [HBR, B] these distinctions make no difference but they are detected by a synchronisation tree semantics. We present the first alternative and leave the second to the reader—or see [B]. We choose to obtain $\Box$ as a restriction of $|||$.

6.15 Definition. Let (S, l_S) and (T, l_T) be synchronisation trees labelled by elements of L_2 (or L_3). Define $(S, l_S)\Box(T, l_T)$ to be the synchronisation tree consisting of sequences $< c_0, \ldots, c_{n-1} >$ of $(S, l_S) \,|||\, (T, l_T)$ which satisfy

$$(\forall i.\, l_S\rho_0(c_i) = * \text{ or } l_S\rho_0(c_i) = \tau) \text{ or } (\forall i.\, l_T\rho_1(c_i) = * \text{ or } l_T\rho_1(c_i) = \tau)$$

with the labelling l given by $l((a,*)) = l_S(a)$ and $l((*,b)) = l_T(b)$.

6.16 Proposition. *Let S and T be L_2-synchronisation trees so*

$$S \cong \sum_i \lambda_i S_i + \sum_k \tau S_k \text{ and } T \cong \sum_j \mu_j T_i + \sum_l \tau T_l.$$

Then

$$S \Box T \cong \sum_i \lambda_i(S_i \Box (T\lceil\tau)) + \sum_j \mu_j((S\lceil\tau)\Box T_j) + \sum_k \tau(S_k \Box T) + \sum_l \tau(S \Box T_l),$$

where for instance $T\lceil\tau$ abbreviates $T\lceil\{\tau\}$.

As a final example we exhibit how Milner's synchronous calculi fit into the picture. In [M2] algebras of actions are presented. They are closely related to synchronisation algebras, though because the algebras do not contain $*$ they cannot express asynchrony in the direct way synchronisation algebras can. The most general algebras of actions described in [M2] are Abelian monoids of the form $(M, \bullet, 1)$. The identity element serves to label delay events. These are essential to the way asynchrony is handled in [M2]; there the asynchrony of an event is modelled by allowing the event to be preceded by an arbitrary number of delay events. Contrast the direct way asynchrony is modelled using synchronisation algebras to restrict the events in the product; the fact that an event is not synchronised with any events of a process is expressed by the event not having any component event from the process.

We show how Milner's monoids of actions determine synchronisation algebras which satisfy the synchronous law of definition 4.4.

6.17 Definition. Let $(M, \bullet_M, 1)$ be an Abelian monoid (assumed to not contain $*$ or 0).

Define $L[M]$ to be the algebra $(M \cup \{*, 0\}, \bullet, *, 0)$ where $\bullet$ extends the monoid operation $\bullet_M$ so $* \bullet * = *$, and $* \bullet \mu = \mu \bullet * = 0$ for $\mu \in M \cup \{0\}$, and $0 \bullet \mu = \mu \bullet 0 = 0$ for $\mu \in M \cup \{*, 0\}$ and $\mu \bullet \mu' = \mu \bullet_M \mu'$ for $\mu, \mu' \in M$.

Define a divisor relation on $(M, \bullet_M, 1)$ by

$$\mu \ div_M \ \mu' \Leftrightarrow \mu = \mu' \text{ or } \exists \nu. \ \mu \bullet_M \nu = \mu'.$$

6.18 Lemma. *The algebra $L[M]$ defined above is a synchronisation algebra which satifies the synchronous law $\forall \lambda \neq *. \ \lambda \bullet * = 0$. Further, the algebra $L[M]$ satisfies the L.C.M. law $\alpha \ div \ \gamma \ \& \ \beta \ div \ \gamma \Rightarrow \alpha \bullet \beta \ div \ \gamma$ iff M satisfies the L.C.M. law $\alpha \ div_M \ \gamma \ \& \ \beta \ div_M \ \gamma \Rightarrow \alpha \bullet \beta \ div_M \ \gamma$.*

Proof. These follow because the composition $\bullet$ of $L[M]$ is simply the extension of $\bullet_M$ to the extra elements $*$ and 0. ∎

6.19 Proposition. *Let L be a synchronisation algebra which satisfies the synchronous law. Then the parallel composition of L–synchronisation trees*

$$S \cong \sum_i \lambda_i S_i \quad \text{and} \quad T \cong \sum_j \mu_j T_j$$

has the form

$$S \textcircled{\scriptsize L} T \cong \sum_{\lambda_i \bullet \mu_j \neq 0} (\lambda_i \bullet \mu_j)(S_i \textcircled{\scriptsize L} T_j).$$

So then parallel composition $\textcircled{\scriptsize L}$ is obtained by restricting $\otimes$ the synchronous product.

Let $(M, \bullet_M, 1)$ be an Abelian monoid. Write $\times_M$ for the parallel composition with respect to the synchronisation algebra $L[M]$. Then for two M-labelled synchronisation trees

$$S \cong \sum_i \lambda_i S_i \quad \text{and} \quad T \cong \sum_j \mu_j T_j$$

we have

$$S \times_M T \cong \sum_{i,j} (\lambda_i \bullet_M \mu_j)(S_i \times_M T_j).$$

The operation $\times_M$ coincides with product in the category of synchronisation trees iff the operation $\bullet_M$ in $(M, \bullet_M, 1)$ behaves like an L.C.M.. If $(M, \bullet_M, 1)$ is an Abelian group $\times_M$ coincides with product.

Proof. When the synchronisation algebra satisfies the synchronous law parallel composition takes the above form by 6.10. By 2.12 this is a restriction of the synchronous product. The remaining facts follow directly from the definition of $L[M]$, 6.11 and 6.18. (Clearly an Abelian group satisfies the LCM law.) ∎

7. Denotational semantics.

We present a denotational semantics to a simple parallel programming language which involves the constructs we have defined earlier. The class of languages is parameterised by the synchronisation algebra L.

7.1 Definition. Let L be a synchronisation algebra. The language $\mathbf{Proc}_L$is given by the following grammar:

$$t ::= NIL \mid x \mid \lambda t \mid t + t \mid t\lceil\Lambda \mid t[\Xi] \mid t \; \textcircled{\scriptsize L} \; t \mid \mathbf{rec}x.t$$

where x is in some set of variables X over processes, $\lambda \in L\backslash\{*,0\}$, $\Lambda \subseteq L\backslash\{*,0\}$ is closed under $div \cap div^{-1}$, and $\Xi : L \to L$ is a strict homomorphism.

In order to give a meaning to the recursively defined processes of the form $\mathbf{rec}x.t$ we use the fact that the operations are continuous with respect to a c.p.o. of synchronisation trees. Fortunately the two c.p.o.'s of trees $\leq$ and $\subseteq$ extend naturally to synchronisation trees in such a way that the operations of the previous section are continuous.

7.2 Definition. Let L be a synchronisation algebra. Define the orderings $\leq_L$ and $\subseteq_L$ on synchronisation trees by:

$$(S, l_S) \leq_L (T, l_T) \Leftrightarrow S \leq T \;\&\; l_S = l_T\lceil A,$$
$$(S, l_S) \subseteq_L (T, l_T) \Leftrightarrow S \subseteq T \;\&\; l_S = l_T\lceil A.$$

7.3 Theorem. *The null synchronisation tree $(\{<>\}, \emptyset)$ is the least L-synchronisation tree with respect to both orderings $\leq_L$ and $\subseteq_L$. Both orderings $\leq_L$ and $\subseteq_L$ possess least upper bounds of ω-chains; the lub of a chain $(T_0, l_0), (T_1, l_1), \ldots, (T_n, l_n), \ldots$ with respect to either order takes the form $(\bigcup_n T_n, \bigcup_n l_n)$.*

All the operations lifting $T \mapsto \lambda T$, sum $+$, restriction $T \mapsto T\lceil\Lambda$, relabelling $T \mapsto T[\Xi]$ and parallel composition $\textcircled{\scriptsize L}$, of section 6, are continuous with respect to $\leq_L$ and $\subseteq_L$ i.e. they preserve lubs of ω-chains.

Proof. The cpo properties of of $\leq_L$ and $\subseteq_L$ follow directly from the cpo properties of $\leq$ and $\subseteq$.

The continuity of the operations on synchronisation trees follows from the continuity of the operations on trees from which they are derived *e.g.* parallel composition is a restriction of the product so its continuity with respect to $\leq_L$ is proved as follows.

Let $T_0 \leq_L \cdots T_n \leq_L \cdots$ be an ω-chain of synchronisation trees such that T_n is over events A_n labelled by l_n. We write its lub as $\bigcup_n T_n$ over events $A = \bigcup_n A_n$ with labelling $l = \bigcup_n l_n$. Let S be a synchronisation tree with events B labelled by l_S. We use $\rho_0 : A \times_* B \to_* A$ and $\rho_1 : A \times_* B \to_* B$ to represent the obvious projection functions on events. The parallel composition of $\bigcup_n T_n$ and S is the restriction of their product to events $C = \{e \in A \times_* B \mid l\rho_0(e) \bullet l_S\rho_1(e) \neq 0\}$ so we obtain

$$\begin{aligned}(\bigcup_n T_n) \; \textcircled{\scriptsize L} \; S = (\bigcup_n T_n) \times S\lceil C &= (\bigcup_n (T_n \times S))\lceil C \quad \text{by the continuity of } \times \\ &= \bigcup_n (T_n \times S\lceil C) \quad \text{by the continuity of restriction} \\ &= \bigcup_n (T_n \times S\lceil C_n) = \bigcup_n T_n \; \textcircled{\scriptsize L} \; S.\end{aligned}$$

as required. ∎

Thus we can give a denotational semantics to $\mathbf{Proc}_L$by representing recursively defined processes as the least fixed points of continuous functionals.

7.4 Definition. Denotational semantics for $\mathbf{Proc}_L$.

Let L be a synchronisation algebra. Define an *environment* for process variables to be a function $\rho : X \to \mathbf{Tr}_L$. For a term t and an environment ρ, define the denotation of t with respect to ρ written $[\![t]\!]\rho$ by the following structural induction. Note syntactic operators appear on the left and their semantic counterparts on the right.

$$\begin{array}{llll}
[\![NIL]\!]\rho & =(\{<>\},\emptyset) & [\![t\lceil\Lambda]\!]\rho & =[\![t]\!]\rho\lceil\Lambda \\
[\![x]\!]\rho & =\rho(x) & [\![t[\Xi]]\!]\rho & =[\![t]\!]\rho[\Xi] \\
[\![\lambda t]\!]\rho & =\lambda([\![t]\!]\rho) & [\![t_1 \textcircled{L} t_2]\!]\rho & =[\![t_1]\!]\rho \textcircled{L} [\![t_2]\!]\rho \\
[\![t_1+t_2]\!]\rho & =[\![t_1]\!]\rho+[\![t_2]\!]\rho & [\![\mathbf{rec}x.t]\!]\rho & =\mathit{fix}\,\Gamma
\end{array}$$

where $\Gamma : \mathbf{Tr}_L \to \mathbf{Tr}_L$ is given by $\Gamma(T) = [\![t]\!]\rho[T/x]$ and *fix* is the least-fixed-point operator so that $\mathit{fix}\,\Gamma = (\bigcup_n T_n, \bigcup_n l_n)$ where $(T_0, l_0) = (\{<>\}, \emptyset)$ and $(T_{n+1}, l_{n-1}) = \Gamma(T_n, l_n)$ inductively.

Remark. A straightforward structural induction shows that Γ above is indeed continuous with respect to either order $\leq_L$ or $\subseteq_L$ so the denotation of a recursively defined process is really the least fixed point of the associated functional Γ.

Choosing L to be the appropriate synchronisation algebra we immediately obtain denotational semantics for CCS and SCCS.

Of course we cannot expect all languages to fit into the simple scheme $\mathbf{Proc}_L$; for instance the CSP–language of [HBR, B] does not quite because it has two parallel compositions corresponding to two synchronisation algebras on the same set of labels. However the semantics for this language and that for CCS with value–passing follow similar lines to that for $\mathbf{Proc}_L$.

We point out how to extend the language $\mathbf{Proc}_L$to value–passing. We assume the synchronisation algebra is that of CCS with value passing, as given in example 4.5. Include terms of the form $\overline{\alpha}v.t$ with denotation $\overline{\alpha}v[\![t]\!]\rho$ to represent the sending of a value v. Include terms of the form $\alpha\vartheta.t$, where ϑ is a variable over the set of values V, with denotation $\sum_{v\in V}\alpha v[\![t]\!]\rho$ to represent the receipt of a value. Terms can be taken to include constants from V, value-variables like ϑ, conditional expressions *etc.* so the language can be quite rich—see [M1] for the full language of CCS and examples.

Some languages like those in [H, Mi] have a parallel composition which depends on *sorts* being associated with processes. They need a slightly more intricate definition of parallel composition which uses combinations of our parallel composition, with respect to some synchronisation algebra, together with restriction and relabelling.

8. Labelled transition systems.

We show how categories of synchronisation trees fit into the broader categories of labelled transition systems. Transition systems have often been used to give operational semantics to programming languages. For example, semantics for Robin Milner's CCS are often based on them and Gordon Plotkin shows in [P1,2] how widely they can be applied in giving semantics to languages. This section provides a bridge between operational semantics in terms of transition systems and denotational semantics expressed in terms of trees.

8.1 Definition. A *transition system* is a 4-tuple (S, i, A, Tran) where S is a set of *states* with *initial state* i, A is a set of *events*, $\mathit{Tran} \subseteq S \times A \times S$ is the *transition relation*, elements of which are called *transitions*, which satisfy

(i) $\forall a \in A \exists s, s' \in S.\, (s, a, s') \in \mathit{Tran}$,

(ii) $(s, a, s') \in \mathit{Tran} \;\&\; (s, a, s'') \in \mathit{Tran} \Rightarrow s' = s''$.

Intuitively a transition system represents a process which can make transitions between states starting from an initial state. Here we assume, as with trees, it can only perform one event at a time. The first axiom we impose says every event is associated with some transition and the second axiom says that from a state the occurrence of an event is associated with a unique transition and so, of course, leads to a unique state. Thus transitions from a state correspond to occurrences of events from that state. (Note however that this will not be the case for "idle" transitions which we shall introduce soon.) Of course transition systems are more general than trees because the transitive closure of the transition relation may contain loops. In fact this is often the way recursion is handled when using transition systems.

8.2 Notation. Let (S, i, A, Tran) be a transition system. We draw the transitions between two states as arrows—there may be more than one. For example the transition system $(\{0,1\}, 0, \{a, b\}, \{(0,a,1), (0,b,0)\})$ would be drawn as:

$$\begin{tikzcd} 0 \arrow[r, bend left, "a"] \arrow[r, bend right, "b"'] & 1 \end{tikzcd}$$

And we can write the transition $(0, a, 1)$ as $0 \xrightarrow{a} 1$, so events serve to index transitions between pairs of states.

It is convenient to extend the set of transitions in a formal way so that they include the possiblity of inaction at any state. We already have a symbol for such inaction, the symbol $*$. Of course inaction does not take a state to another state so we extend the set *Tran* just by elements of the form $(s, *, s)$. We call such transitions *idle* transitions because they are not associated with any event occurrence. For a transition system as above write the idle transitions as

$$\mathit{Tran}_* = \mathit{Tran} \cup \{(s, *, s) \mid s \in S\}.$$

Idle transitions are *not* to be thought of as events of inaction performed by a process; they are not associated with any event of the process at all.

Morphisms on transition systems are defined analogously to those on trees. The intuition is the same. A morphism from a transition system T to a transition system U specifies how the occurrence of an event in T implies the synchronised occurrence of an event in U. States of T image to states of U. However, there may well be occurrences of events in T which are not represented by any event occurrences in U. The transitions associated with such event occurrences image to idle transitions introduced above. Hence we define a morphism as consisting of two parts one a function on states and the other a partial function on events which induces a function on transitions, including the idle ones. Following the definition on trees, we say a morphism is synchronous if it is a total function on events and so never sends a non-idle transition to an idle one.

8.3 Definition. A *morphism* from a transition system $(S_0, i_0, A_0, \mathit{Tran}_0)$ to a transition system $(S_1, i_1, A_1, \mathit{Tran}_1)$ is a pair (f_S, f_E) where $f_S : S_0 \to S_1$ is a function on states such that

$$f_S(i_0) = i_1$$

and where $f_E : A_0 \to_* A_1$ is a partial function on events which satisfies

$$(s, a, s') \in \mathit{Tran}_0 \Rightarrow (f_S(s), f_E(a), f_S(s')) \in \mathit{Tran}_{1*}.$$

Say the morphism (f_S, f_E) is *synchronous* if f_E is a total function.

8.4 Proposition. *Transition systems with morphisms as defined above form a category under the pairwise composition of functions* $(f_S, f_E)\circ(g_S, g_E) =_{def} (f_S g_S, f_E g_E)$ *where composition on the state functions is the usual composition on total functions and the composition on the event functions is that for partial functions and identity morphisms are pairs of identity functions. Transition systems with synchronous morphisms form a subcategory.*

Proof. Routine. ∎

8.5 Definition. Let **Tran** denote the category of transition systems with the above definition of morphism and $\mathbf{Tran}_{syn}$ the subcategory with synchronous morphisms.

Let us see the form products take in the category **Tran** . The projection functions will provide examples of typical morphisms.

8.7 Definition. (The product of transition systems) Let $(S_0, i_0, A_0, \mathit{Tran}_0)$ and $(S_1, i_1, A_1, \mathit{Tran}_1)$ be transition systems. Define their *product*, $(S_0, i_0, A_0, \mathit{Tran}_0) \times (S_1, i_1, A_1, \mathit{Tran}_1) = (S, i, A, \mathit{Tran})$ by taking:

(i) States $S = S_0 \times S_1$, the product in **Set** with projections $\pi_j : S \to S_j$ for $j = 0, 1$,
(ii) initial state $i = (i_0, i_1)$,
(iii) events $A = A_0 \times_* A_1$, the product in $\mathbf{Set}_*$ with projections $\rho_j : A \to_* A_j$ for $j = 0, 1$, and,
(iv) transitions $(s, c, s') \in \mathit{Tran} \Leftrightarrow \begin{cases} (\pi_0(s), \rho_0(c), \pi_0(s')) \in \mathit{Tran}_{0*} \ \& \\ (\pi_1(s), \rho_1(c), \pi_1(s')) \in \mathit{Tran}_{1*}. \end{cases}$

Define the *projections* $\Pi_j : (S, i, A, \mathit{Tran}) \to (S_j, i_j, A_j, \mathit{Tran}_j)$ by taking $\Pi_j = (\pi_j, \rho_j)$ for $j = 0, 1$.

A similar construction has been used to build the "product machine" of [CES]. It really is a product.

8.8 Theorem. *The construction* $(S_0, i_0, A_0, \mathit{Tran}_0) \times (S_1, i_1, A_1, \mathit{Tran}_1), \Pi_0, \Pi_1$ *above is a categorical product in the category* **Tran** *of transition systems.*

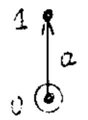
Proof. It follows immediately from the definition of product that it is a transition system—the axioms follow from their truth in the components—and that the projections are morphisms of transition systems.

Let T_j abbreviate $(S_j, i_j, A_j, \mathit{Tran}_j)$ for $j = 0, 1$. Suppose that U is a transition system and that $f = (f_S, f_E) : U \to T_0$ and $g = (g_S, g_E) : U \to T_1$ are morphisms. In order for $T_0 \times T_1$ to be a product we require that there is a unique morphism $h = (h_S, h_E) : U \to T_0 \times T_1$ such that $\Pi_0 h = f$ and $\Pi_1 h = g$. This is so when we define h as follows:

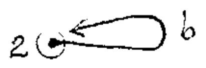
$$h_S(u) = (f_S(u), g_S(u)) \quad \text{and} \quad h_E(c) = (f_E(c), g_E(c))$$

for u a state of U and c an event of U. ∎

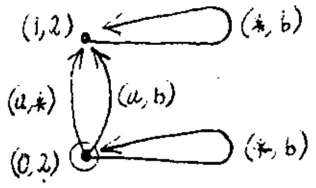
8.9 Example. The product of the transition systems



and



takes the form:



We shall see how the definition of product of transition systems in **Tran** generalises that of trees in **Tr** . In fact a product of transition systems will unfold to a product of trees. Transition systems also have a coproduct, perhaps not quite what is expected as its unfolding will turn out to not coincide with the coproduct of trees.

8.10 Definition. (The coproduct of transition systems) Let $(S_0, i_0, A_0, Tran_0)$ and $(S_1, i_1, A_1, Tran_1)$ be transition systems. Define their *coproduct* $(S_0, i_0, A_0, Tran_0) + (S_1, i_1, A_1, Tran_1) = (S, i, A, Tran)$ by taking:

(i) $S = (S_0 \times \{ i_1 \}) \cup (\{ i_0 \} \times S_1)$ with injections $in_j : S_j \to S$, for $j = 0, 1$, given by $in_0(s) = (s, i_1)$ and $in_1(s) = (i_0, s)$,

(ii) $i = (i_0, i_1)$,

(iii) $A = (\{ 0 \} \times A_0) \cup (\{ 1 \} \times A_1)$ the disjoint union of the sets of events with injections $\alpha_j : A_j \to A$ given by $\alpha_j(a) = (j, a)$ for $j = 0, 1$ and,

(iv) $t \in Tran \Leftrightarrow \begin{cases} \exists (s, a, s') \in Tran_0 . t = (in_0(s), \alpha_0(a), in_0(s')) \text{ or} \\ \exists (s, a, s') \in Tran_1 . t = (in_1(s), \alpha_1(a), in_1(s')). \end{cases}$

Define the *injections* $I_j : (S_j, i_j, A_j, Tran_j) \to (S, i, A, Tran)$ by $I_j = (in_j, \alpha_j)$ for $j = 0, 1$.

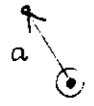
8.11 Theorem. *The construction above is a coproduct in the categories* **Tran** *and* $\mathbf{Tran}_{syn}$ *of transition systems.*

Proof. It is easy to see that the coproduct construction gives a transition system and that the injections are indeed (synchronous) morphisms. Suppose $f : T_0 \to U$ and $g : T_1 \to U$ are morphisms from transition systems T_j abbreviating $(S_j, i_j, A_j, Tran_j)$, for $j = 0, 1$, to a transition system U. Define a morphism $h = (h_S, h_E) : T_0 + T_1 \to U$ by taking:

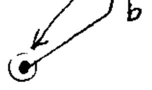
$$h_S(s) = \begin{cases} f_S(s_0) & \text{if } s = in_0(s_0) \\ g_S(s_1) & \text{if } s = in_1(s_1) \end{cases} \quad \text{and} \quad h_E(c) = \begin{cases} f_E(a) & \text{if } c = \alpha_0(a) \\ g_E(a) & \text{if } c = \alpha_1(a). \end{cases}$$

It is easily seen that h is the unique morphism of transition systems so that $hI_0 = f$ and $hI_1 = g$. Moreover if f and g are synchronous then so is h. Therefore the construction is a coproduct in **Tran** and $\mathbf{Tran}_{syn}$.
∎

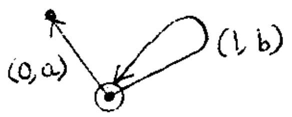
8.12 Example. The coproduct of the transition systems



and



takes the form:



Clearly a tree can be viewed as a transition system:

8.13 Definition. Let S be a tree over the set A. Define $\mathcal{TS}\,S$ to be $(S, <>, E, \mathit{Tran})$ where

$$E = \{ (s,a) \in S \times A \mid s < a > \in S \} \text{ and}$$
$$\mathit{Tran} = \{ (s,(s,a),s') \mid s < a > = s' \in S \}.$$

Extend $\mathcal{TS}$ to a functor by defining it to act on morphisms of trees as follows: Let $f : S \to U$ be a tree morphism. Define $(\mathcal{TS}\,f) : \mathcal{TS}\,S \to \mathcal{TS}\,U$ by taking

$$(\mathcal{TS}\,f)_S(s) = f(s) \quad \text{and} \quad (\mathcal{TS}\,f)_E(s,a) = \begin{cases} (f(s),b) & \text{if } f(s < a >) = f(s) < b > \\ * & \text{otherwise.} \end{cases}$$

Remark. Notice that $\mathcal{TS}$ would not have extended to a functor, in the above definition, if we had taken A instead of E as the events. The reason: In the category **Tr** morphisms respect only the node-arc structure, and not the event sets, which are respected by the more discriminating morphisms in **Tran** .

Not only can trees be viewed as transition systems, but also transition systems can be unfolded to trees. This is well known. The unfolding is determined by the categorical set-up. It is characterised to within isomorphism as the right adjoint to the obvious functor $\mathcal{TS}$ taking trees to transition systems—see [AM] or [Mac]. In other words, given a transition system its unfolding is cofree over it with respect to $\mathcal{TS}$ the natural identification of trees with a form of transition system.

8.14 Definition. Let (S, i, A, Tran) be a transition system. Define its *unfolding* $\mathcal{U}(S, i, A, \mathit{Tran})$ to be the tree

$$\{ < a_0, a_1, \ldots, a_{n-1} > \mid \exists s_0, s_1, \ldots, s_n \in S.\, s_0 = i \;\&\; \forall j < n.\,(s_j, a_j, s_{j+1}) \in \mathit{Tran} \}.$$

Define the *folding* morphism $\phi = (\phi_S, \phi_E) : \mathcal{TS}\,\mathcal{U}((S, i, A, \mathit{Tran})) \to (S, i, A, \mathit{Tran})$ by taking $\phi_E(u,a) = a$ on events and defining ϕ_S by induction as follows:

$$\phi_S(<>) = i \quad \text{and} \quad \phi_S(u < a >) = s$$

where s is the unique state such that $(\phi_S(u), a, s) \in \mathit{Tran}$.

Thus ϕ folds a state $< a_0, a_1, \ldots, a_{n-1} >$ in the unfolding to the state s_n where $s_0, s_1, \ldots, s_n \in S$ is the unique sequence of states such that $s_0 = i \;\&\; \forall j < n.\,(s_j, a_j, s_{j+1}) \in \mathit{Tran}$.

8.15 Theorem. *Let (S, i, A, Tran) be a transition system. Then $\mathcal{U}(S, i, A, \mathit{Tran})$ is a synchronisation tree and ϕ defined above is a morphism of transition systems. In fact, $\mathcal{U}(S, i, A, \mathit{Tran}), \phi$ is cofree over (S, i, A, Tran) with respect to the functor $\mathcal{TS}$ i.e. for any morphism $f : \mathcal{TS}\,V \to (S, i, A, \mathit{Tran})$ with V a tree, there is a unique morphism $g : V \to \mathcal{U}(S, i, A, \mathit{Tran})$ in* **Tr** *such that $f = \phi(\mathcal{TS}\,g)$:*

$$\begin{tikzcd} \mathcal{U}(S,i,A,Tran) \\ V \arrow[u, "g"'] \end{tikzcd} \qquad \begin{tikzcd} (S,i,A,Tran) & \mathcal{TS}\,\mathcal{U}(S,i,A,Tran) \arrow[l, "\phi"'] \\ & \mathcal{TS}\,V \arrow[u, "\mathcal{TS}\,g"'] \arrow[ul, "f"] \end{tikzcd}$$

Consequently, $\mathcal{U}$ extends to a right adjoint of $\mathcal{TS}$.

Proof. Let V be a tree and $f : \mathcal{TS}\,V \to (S, i, A, \mathit{Tran})$ be a morphism of transition systems.

Define $g : V \to \mathcal{U}(S, i, A, \mathit{Tran})$ by induction as follows:

$$g(<>) = <>,$$
$$g(v < b >) = \begin{cases} g(v) < f_E(v,b) > & \text{if } f_E(v,b) \neq *, \\ g(v) & \text{otherwise.} \end{cases}$$

Clearly g is a morphism of trees. We require $\phi \circ (\mathcal{T}\mathcal{S}g) = f$ and of course this follows if we can show $(\phi \circ (\mathcal{T}\mathcal{S}g))_S = f_S$ and $(\phi \circ (\mathcal{T}\mathcal{S}g))_E = f_E$.

We first show $(\phi \circ (\mathcal{T}\mathcal{S}g))_S = f_S$. We show $\phi_S \circ g(v) = f_S(v)$ by induction on $v \in V$. Obviously $\phi_S \circ g(<>) = f_S(<>)$ establishing the basis of the induction.

Now we show the inductive step, that $\phi_S \circ g(v < b >) = f_S(v < b >)$ if the induction hypothesis $\phi_S \circ g(v) = f_S(v)$ holds. From the definition of g there are two cases to consider, when $f_E(v,b) \neq *$ and when $f_E(v,b) = *$.

Assume $f_E(v,b) \neq *$. From the definition of g we get $g(v < b >) = g(v) < f_E(v,b) >$. From the definition of ϕ_S we obtain $\phi_S \circ g(v < b >) = s$ where s is the *unique* state such that $(\phi_S(g(v)), f_E(v,b), s) \in \mathit{Tran}$. As f is a morphism of transition systems we must have $(f_S(v), f_E(v,b), f_S(v < b >)) \in \mathit{Tran}$ too. The induction hypothesis provides $\phi_S(g(v)) = f_S(v)$. Thus $\phi_S(g(v < b >)) = s = f_S(v < b >)$.

Now assume $f_E(v,b) = *$. The definition of g gives $g(v < b >) = g(v)$. So $\phi_S(g(v < b >)) = \phi_S(g(v)) = f_S(v)$ by induction. As f is a morphism $f_S(v) = f_S(v < b >)$. Thus $\phi_S \circ g(v < b >) = f_S(v < b >)$.

This shows that $(\phi \circ (\mathcal{T}\mathcal{S}g))_S = f_S$. We now show $(\phi \circ (\mathcal{T}\mathcal{S}g))_E = f_E$. This is part of a more general fact which also establishes the uniqueness of g; a morphism of trees $h : V \to \mathcal{U}(S, i, A, \mathit{Tran})$ satisfies g's recursive definition iff $(\phi \circ (\mathcal{T}\mathcal{S}h))_E = f_E$. More precisely we show:

Let $h : V \to \mathcal{U}(S, i, A, \mathit{Tran})$ be a morphism of trees. Then $(\phi \circ (\mathcal{T}\mathcal{S}h))_E = f_E$ iff

$$h(<>) = <>,$$
$$h(v < b >) = \begin{cases} g(v) < f_E(v,b) > & \text{if } f_E(v,b) \neq *, \\ h(v) & \text{otherwise} \end{cases}$$

for $v \in V$ and b an event of the tree V such that $v < b > \in V$.

"if": Let $v < b > \in V$. From the assumption and the definition of $(\mathcal{T}\mathcal{S}h)_E$ we obtain

$$(\mathcal{T}\mathcal{S}h)_E(v,b) = \begin{cases} (g(v), f_E(v,b)) & \text{if } f_E(v,b) \neq * \\ * & \text{otherwise.} \end{cases}$$

From the definition of ϕ_E we immediately have $\phi_E \circ (\mathcal{T}\mathcal{S}h)_E(v,b) = f_E(v,b)$.

"only if": By the definition of ϕ_E and $(\mathcal{T}\mathcal{S}h)_E$ we have

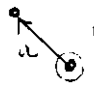
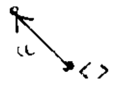
$$\phi_E \circ (\mathcal{T}\mathcal{S}h)_E(v,b) = \begin{cases} a & \text{if } h(v < b >) = h(v) < a > \\ * & \text{otherwise.} \end{cases}$$

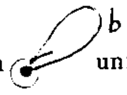
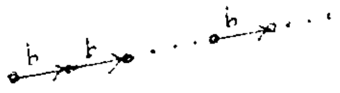
But by assumption $\phi_E \circ (\mathcal{T}\mathcal{S}h)_E(v,b) = f_E(v,b)$ which implies the result.

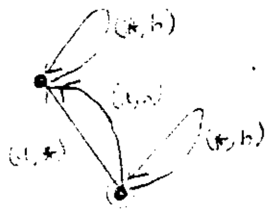
Clearly it now follows that $(\phi \circ (\mathcal{T}\mathcal{S}g))_E = f_E$. So $\phi \circ g = f$. The uniqueness of g follows too. Assume $h : V \to \mathcal{U}(S, i, A, \mathit{Tran})$ is a morphism such that $\phi \circ h = f$. Then $(\phi \circ (\mathcal{T}\mathcal{S}h))_E = f_E$. By a simple induction using the above result with $h(<>) = <>$ we obtain $h(v) = g(v)$ for all $v \in V$.

The theorem is proved. ∎

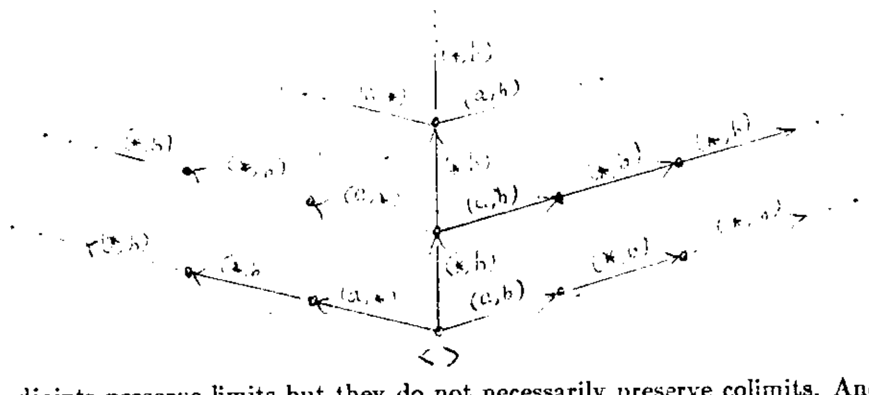
Right adjoints have the pleasant property that they preserve limits, so in particular they preserve products—see [AM] or [Mac]. This means that if we take the product of two transition systems and then unfold them we obtain the same tree, to within isomorphism, as if we unfold them first and then take their product in the category of trees. This is significant for us because we derive parallel compositions from products by restricting the events. It will mean that we can define a parallel composition directly on labelled transition systems and know that it unfolds to the parallel composition of the synchronisation trees which are the unfoldings. In view of these facts the following example is not surprising.

8.16 Example. The transition system  unfolds to the tree: 

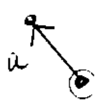
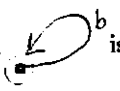
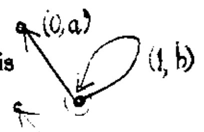
The transition system  unfolds to the tree: 

Their product 

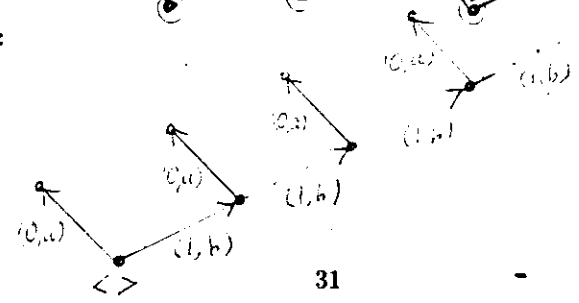
unfolds to the following tree which is isomorphic to the product (in **Tr**) of the two tree unfoldings:



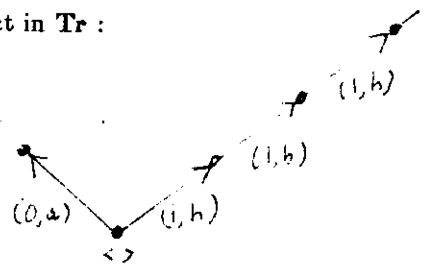
Right adjoints preserve limits but they do not necessarily preserve colimits. And in fact the unfolding functor $\mathcal{U}$ does not preserve coproducts as the following example shows.

8.17 Example. The coproduct of the transition systems  and  is 

which unfolds to this tree:



But their two unfoldings have this coproduct in **Tr** :



Now we label events by elements of a synchronisation algebra to specify how they interact with the environment.

8.18 Definition. Let L be a synchronisation algebra. An *L-labelled transition system* is a 5-tuple $(S, i, A, \mathit{Tran}, l)$ where (S, i, A, Tran) is a transition system and l is a *labelling function* $l : A \to L \setminus \{*, 0\}$.

Just as with trees we can restrict morphisms on transition systems in accord with labellings of the transitions by elements of a synchronisation algebra.

8.19 Definition. Let L be a synchronisation algebra. Let $(S_0, i_0, A_0, \mathit{Tran}_0, l_0)$ and $(S_1, i_1, A_1, \mathit{Tran}_1, l_1)$ be L-labelled transition systems. An L-morphism from $(S_0, i_0, A_0, \mathit{Tran}_0, l_0)$ to $(S_1, i_1, A_1, \mathit{Tran}_1, l_1)$ is a morphism of transition systems $f : (S_0, i_0, A_0, \mathit{Tran}_0) \to (S_1, i_1, A_1, \mathit{Tran}_1)$ such that $l_1 f_E(a)\ \mathit{div}\ l_0(a)$ for all $a \in A_0$.

The condition satisfied by L-morphisms of transition systems simply expresses that the label of the image of an event must divide the label of the event.

8.20 Proposition. *Let L be a synchronisation algebra. Then L-labelled transition systems with L-morphisms form a category with composition the pairwise composition of functions and identities pairs of identity functions.*

8.21 Definition. Let L be a synchronisation algebra. Let $\mathbf{TRAN}_L$ be the category of labelled transition systems.

Not surprisingly labelled transition systems unfold to labelled trees or synchronisation trees simply by extending the unfolding operation to cope with labels. Similarly synchronisation trees can be viewed as sorts of labelled transition systems by extending the operation $\mathcal{TS}$.

8.22 Definition. Let L be a synchronisation algebra. Define the operation $\mathcal{TS}_L : \mathbf{Tr}_L \to \mathbf{TRAN}_L$ by $\mathcal{TS}_L : (T, l) \mapsto (\mathcal{TS}\,T, l)$.

Define the *unfolding* operation on labelled transition systems by taking $\mathcal{U}_L : (S, i, A, \mathit{Tran}, l) \mapsto (T, l')$ where $T = \mathcal{U}(S, i, A, \mathit{Tran})$, and l' is l restricted to the events of T. (Not all events A necessarily appear in branches of T.)

8.23 Proposition. *In fact $\mathcal{TS}_L$ extends to a functor with respect to which $\mathcal{U}_L$ gives the cofree object; thus $\mathcal{U}_L$ extends to a right adjoint of $\mathcal{TS}_L$.*

Proof. This follows from theorem 8.15. ∎

Just as with synchronisation trees one can define operations on labelled transition systems and use these to give a semantics to to a variety of parallel programming languages. The most interesting operation is parallel composition which we obtain by restricting the transitions of the product of transition systems in accord with their labelling.

8.24 Definition. Parallel composition of labelled transition systems: Let L be a synchronisation algebra. Let $(S_0, i_0, A_0, Tran_0, l_0)$ and $(S_1, i_1, A_1, Tran_1, l_1)$ be L-labelled transition systems. Define their *parallel composition* $(S_0, i_0, A_0, Tran_0, l_0) \, \textcircled{\scriptsize L} \, (S_1, i_1, A_1, Tran_1, l_1)$ to be $(S, i, A', Tran', l)$ formed from the product of transition systems as follows—we use the notation of definition 8.7:

(i) S is the states of their product with the same initial state i,
(ii) $A' = \{ c \in A_0 \times_* A_1 \mid l_0\rho_0(c) \bullet l_1\rho_1(c) \neq 0 \}$ is a subset of events of the product,
(iii) labelled by $l : A' \to L \setminus \{ *, 0 \}$; $a \mapsto l_0\rho_0(c) \bullet l_1\rho_1(c)$,
(iv) with transitions $Tran' = S \times A' \times S \cap Tran$ which are a subset of the transitions $Tran$ of the product.

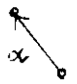
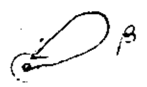
Because the operation of unfolding preserves products and the parallel compositions of synchronisation trees and labelled transition systems are restrictions determined in the same way from the labelling we obtain the following reassuring fact:

8.25 Proposition. *Let L be a synchronisation algebra. The parallel composition of labelled transition systems T_0 and T_1 unfolds to the parallel composition of the unfoldings:*

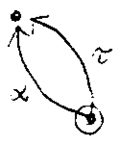
$$\mathcal{U}_L(T_0 \, \textcircled{\scriptsize L} \, T_1) \cong \mathcal{U}_L(T_0) \, \textcircled{\scriptsize L} \, \mathcal{U}_L(T_1).$$

8.26 Example. Let L be a synchronisation algebra with the following multiplication table:

$\bullet$	$*$	α	β	τ	0
$*$	$*$	α	0	0	0
α	α	0	τ	0	0
β	0	τ	0	0	0
.	.	.	.	.	.

The parallel composition of the labelled transition systems  and 

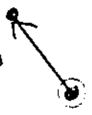
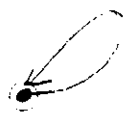
is the appropriate restriction of the product in example 8.9 and takes the form:

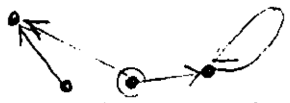


By example 8.17 we know that the unfolding of a coproduct of transition systems is not necessarily the coproduct of their unfoldings; we must look elsewhere for a definition of the sum of labelled transition systems if we wish it to unfold correctly to the sum of the synchronisation-tree unfoldings. We can define the *Milner sum* of two transition systems as follows:

8.27 Definition. Let $(S_0, i_0, A_0, Tran_0)$ and $(S_1, i_1, A_1, Tran_1)$ be transition systems. Define their *Milner sum* $(S_0, i_0, A_0, Tran_0) +_M (S_1, i_1, A_1, Tran_1) = (S, i, A, Tran)$ by taking:

(i) $S = \{ 0 \} \times S_0 \cup \{ 1 \} \times S_1 \cup \{ (2, (i_0, i_1)) \}$,
(ii) $i = (2, (i_0, i_1))$,
(iii) $A = (\{ 0 \} \times A_0) \cup (\{ 1 \} \times A_1)$ and,
(iv) $$t \in Tran \Leftrightarrow \begin{cases} \exists (s, a, s') \in Tran_0 . t = ((0, s), (0, a), (0, s')) \text{ or} \\ \exists (s, a, s') \in Tran_1 . t = ((1, s), (1, a), (1, s')) \text{ or} \\ \exists (i_0, a, s) \in Tran_0 . t = (i, (0, a), (0, s)) \text{ or} \\ \exists (i_1, a, s) \in Tran_1 . t = (i, (1, a), (1, s)). \end{cases}$$

8.28 Example. The Milner sum of the transition systems  and 

is the transition system: 

The Milner sum of two transition systems does unfold to the sum of the two unfoldings. Note too that provided the transition systems have no loops back to the initial state their coproduct does unfold nicely. Of course the same construction works if the transition systems are labelled. It is easy to define operations on labelled transition systems which unfold to the remaining operations on synchronisation trees given in section 6.

As presented, transition systems are still an interleaving model of concurrency because they allow the occurrence of only one event at a time. One can however generalise transition systems to reflect concurrency. For example one can view Petri nets as kinds of transition systems in which transitions are sets of concurrently firing events—see *e.g.* [Bra]. The definition of morphism can be generalised to reflect this extra information about concurrency while maintaining, in essence, the results of this section—see [W3].

9. Proof rules.

Naturally one wishes to use semantics to prove properties of programs. This can often be reduced to the problem of whether or not two programs have equivalent behaviour with respect to some natural notion of equivalence. Thus much work is involved with inventing natural equivalences and proof rules for them—see *e.g.* [M1], [B], [HN].

Consider the programming language $\mathbf{Proc}_L$for some synchronisation algebra L. There is an obvious equivalence on closed terms of the language: Say two closed terms are equivalent iff they have isomorphic denotations. (The idea extends to open terms; say two terms are equivalent if the closed terms obtained by an arbitrary assignment of closed terms to free variables are always equivalent.)

9.1 Definition. Let L be a synchronisation algebra. Let t and t' be closed terms of $\mathbf{Proc}_L$. Write

$$t \sim t' \Leftrightarrow [\![t]\!]\rho \cong [\![t']\!]\rho$$

for some arbitrary environment ρ.

We immediately know some properties of the equivalence. Firstly it really is an equivalence—is reflexive, symmetric and transitive—because these properties hold for isomorphism, and then the commutativity and associativity of sum $+$ with respect to $\sim$ follows directly from the properties of coproduct. Less immediate are the commutativity and associativity of parallel composition $\textcircled{\scriptsize L}$, but these facts follow easily from the corresponding properties of product $\times$ of trees and $\bullet$ in the synchronisation algebra L. Because all our operations preserve isomorphism—all but restriction are functors anyhow and functors must preserve isomorphism—we know that the equivalence $\sim$ is also a congruence with respect to the operations of $\mathbf{Proc}_L$.

9.2 Proposition. *The equivalence $\sim$ on closed terms of $\mathbf{Proc}_L$is a congruence with respect to the operations lifting $T \mapsto \lambda T$, sum $+$, restriction $T \mapsto T\lceil\Lambda$, relabelling $T \mapsto T[\Xi]$ and parallel composition of $\mathbf{Proc}_L$.*

Particular laws follow from particular properties of the synchronisation algebra L. One useful property, when it is valid, is that of the distributivity of parallel composition over sum. This property holds for the equivalence $\sim$ precisely when the synchronisation algebra satisfies the synchronous law.

9.3 Proposition. *Let L be a synchronisation algebra. The following conditions are equivalent:*

(i) *L satisfies the synchronous law i.e. $\lambda \bullet * = 0$ for λ an element of $L \setminus \{*\}$,*

(ii) *NIL* is a $\textcircled{L}$-zero *i.e.* $NIL \textcircled{L} t \sim NIL$ for t an arbitrary closed term of $\mathbf{Proc}_L$,

(iii) Parallel composition distributes over sum *i.e.* $t \textcircled{L} (u+v) \sim (t \textcircled{L} u) + (t \textcircled{L} v)$, for arbitrary closed terms t, u, v of $\mathbf{Proc}_L$.

Proof.

(i)⇔ (ii): If L is synchronous no events of the form $(*, e)$ are allowed in the parallel composition so $NIL \textcircled{L} t \sim NIL$ for any closed term t. Conversely if $NIL \textcircled{L} t \sim NIL$ for any closed term t then in particular $NIL \textcircled{L} \lambda NIL \sim NIL$ and this isomorphism ensures $* \bullet \lambda = 0$.

(i)⇒ (iii): The distribution of $\textcircled{L}$ over $+$ follows directly from the expansion rule of proposition 6.19.

(iii)⇒ (i): Suppose (iii) and that $\alpha \bullet * = \beta \neq 0$ for some $\alpha \in L \setminus \{*\}$. Then $\beta NIL \sim \alpha NIL \textcircled{L} NIL \sim \alpha NIL \textcircled{L} (NIL + NIL) \sim (\alpha NIL \textcircled{L} NIL) + (\alpha \textcircled{L} NIL) \sim \beta NIL + \beta NIL$. But this is impossible so $\alpha \bullet * = 0$ for $\alpha \in L \setminus \{*\}$, making L synchronous. ∎

Of course a semantics for a language of synchronising processes may well ensure that parallel composition distributes over sum without the synchronisation algebra being synchronous. The above result only implies that any abstract semantics which factors through our synchronisation tree semantics will satisfy the distributivity. For example the synchronous calculi SCCS do because the equivalences in [M2] could be based on synchronisation trees and the synchronisation algebras associated with monoids of actions are synchronous—see lemma 6.18.

Now we present a sound and complete proof system for the non-recursive processes of $\mathbf{Proc}_L$.

9.4 Definition. Let L be a synchronisation algebra. Let the language $\mathbf{Simp}_L$ consist of the following subset of $\mathbf{Proc}_L$:

$$t ::= NIL \mid \lambda t \mid t + t \mid t\lceil\Lambda \mid t[\Xi] \mid t \textcircled{L} t$$

where $\lambda \in L \setminus \{*, 0\}$, $\Lambda \subseteq L \setminus \{*, 0\}$ is closed under $div \cap div^{-1}$ and $\Xi : L \to L$ is a strict homomorphism.

9.5 Notation. We use the convention that

$$\sum_{i<n} \lambda_i s_i = \lambda_0 s_0 + \cdots + \lambda_{n-1} s_{n-1}$$

where $n > 0$ with the understanding that the sum represents *NIL* when $n = 0$. Our notation assumes the associativity of $+$, one of the rules below.

9.6 Definition. (Proof rules for $\mathbf{Simp}_L$)

Let s, t, u, v range over terms of $\mathbf{Simp}_L$.

1. Rules of equivalence.

$$s \sim s, \quad \frac{s \sim t}{t \sim s}, \quad \frac{s \sim t, t \sim u}{s \sim u}$$

2. Substitutivity.

$$\frac{s \sim s'}{op(s) \sim op(s')}$$

where *op* is an operation of lifting, restriction or relabelling.

$$\frac{s \sim s', t \sim t'}{op(s,t) \sim op(s',t')}$$

where *op* is the operation sum or parallel composition.

3. Divisor rules.

$$\lambda t \sim \lambda' t$$

when $\lambda, \lambda' \in L \setminus \{*, 0\}$ and λ *div* λ' and λ' *div* λ.

4. Rules for restriction.

$$NIL\lceil\Lambda \sim NIL, \quad (s+t)\lceil\Lambda \sim s\lceil\Lambda + t\lceil\Lambda$$

$$(\lambda t)\lceil\Lambda \sim \begin{cases} \lambda(t\lceil\Lambda) & \text{if } \lambda \in \Lambda \\ NIL & \text{if } \lambda \notin \Lambda \end{cases}$$

where $\Lambda \subseteq L \setminus \{*, 0\}$ is closed under the relation $div \cap div^{-1}$ and $\lambda \in L \setminus \{*, 0\}$.

5. Rules for relabelling.

$$NIL[\Xi] \sim NIL, \quad (\lambda t)[\Xi] \sim \Xi(\lambda)t, \quad (s+t)[\Xi] \sim s[\Xi] + t[\Xi]$$

where $\Xi : L \to L$ is a strict homomorphism of L and $\lambda \in L \setminus \{*, 0\}$.

6. Rules for sum.

$$s + NIL \sim s, \quad s + t \sim t + s, \quad s + (t+u) \sim (s+t) + u.$$

7. Expansion rules for parallel composition.

$$\frac{s \sim \sum_{i<n} \lambda_i s_i, \quad t \sim \sum_{j<m} \mu_j t_j}{s \textcircled{L} t \sim \sum_{\lambda_i \bullet * \neq 0} (\lambda_i \bullet *)(s_i \textcircled{L} t) + \sum_{\lambda_i \bullet \mu_j \neq 0} (\lambda_i \bullet \mu_j)(s_i \textcircled{L} t_j) + \sum_{* \bullet \mu_j \neq 0} (* \bullet \mu_j)(s \textcircled{L} t_j)}.$$

9.7 Theorem. *Let L be a synchronisation algebra. Let s and t be terms of* $\mathbf{Simp}_L$. *They have isomorphic denotations as synchronisation trees in* $\mathbf{Tr}_L$ *iff they are provably equivalent according to the proof system above.*

Proof. Previous results ensure that the rules are sound. The above rules are sufficient to convert any term of $\mathbf{Simp}_L$to one of the normal form $\sum_{i<n} \lambda_i s_i$ in which each s_i is itself of normal form. The normal form corresponds in an obvious way to a synchronisation tree. The isomorphism of two denotations is then provable by inductively using the divisor rule. ▮

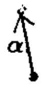
In the case where the synchronisation algebra is synchronous the expansion rules above can be replaced by simpler rules expressing the commutativity and associativity of parallel composition and rules as in proposition 9.3 which say *NIL* is a $\textcircled{L}$-zero and that parallel composition distributes over sum. More precisely the expansion rules can be replaced by the rules:

$$NIL \textcircled{L} t \sim NIL, \quad s \textcircled{L} t \sim t \textcircled{L} s; \quad (s \textcircled{L} t) \textcircled{L} u \sim s \textcircled{L} (t \textcircled{L} u),$$

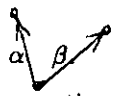
$$t \textcircled{L} (u+v) = (t \textcircled{L} u) + (t \textcircled{L} v), \quad (\lambda s) \textcircled{L} (\mu t) \sim \begin{cases} (\lambda \bullet \mu) s \textcircled{L} t & \text{if } \lambda \bullet \mu \neq 0, \\ NIL & \text{otherwise.} \end{cases}$$

Of course the above proof rules are rather limited; they only work for finite processes and for a somewhat primitive notion of equivalence. Still many more abstract ideas of equivalence are or could be based on synchronisation trees. Proof rules for the more abstract equivalences would have to imply the rules above. It is even arguable that synchronisation trees give the basic interleaving semantics making indentifications of processes which all other interleaving semantics should also make. The argument does not quite push home, however, because of the phenomena of divergence. We explain the problem.

One technique for making a synchronisation-tree semantics more abstract is to identify a process with the set of assertions it satisfies. The assertions may be in some fragment of modal logic and express the possible or inevitable behaviour of a process. A recursively defined process is denoted by the least upper bound of a chain of iterates obtained by repeated application of a continuous functional to the $\perp$-process. One would like that the set assertions satisfied by the recursively defined process is the union of those sets of assertions true for the iterates. Unfortunately this is not the case for synchronisation trees when taking modal assertions which express the inevitable behaviour of a process—see *e.g.* [HM] and [HN]. Suppose one iterate was the synchronisation tree

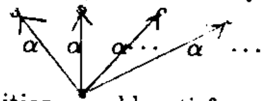


We cannot say of the process that it is inevitably prepared to make an α-communication because some later iterate could be



A satisfaction relation defined between trees and assertions does not respect any approximation ordering on trees. The problem is that trees alone do not carry enough structure to reflect where their growth is complete and incomplete and without such extra knowledge we cannot be sure of any non-trivial assertions about the inevitable behaviour of the process. Of course one can extend trees or transition systems by extra structure to express those states which are incompletely defined, generally called "divergent"—see [HP],[HN] for example. I am not certain how the work above generalises to trees or transition systems which take account of divergence.

Although our approach ignores divergence there is a defence. Each closed program of $\mathbf{Proc}_L$ is given a denotation as a synchronisation tree. This tree faithfully represents the completed program and we can consider those assertions which it satisfies and then take this set of assertions as its more abstract denotation. As an example, the process $P = \mathbf{rec}x.(\alpha NIL + x)$ is denoted by the infinitely branching tree



which according to one reasonable definition would satisfy an assertion saying that the process would inevitably be prepared to make an α-communication. Contrast the situation in [HN] where, essentially, they denote a process by the set of assertions it satisfies. Because in [HN] it is ensured that all the functions in the denotational semantics are continuous they cannot attribute this inevitable behaviour to P. This is not to say the equivalence in [HN] is wrong, just different.

Finally, I hope that the relation between parallel composition and product will be useful in proving properties of processes with synchronised communication. It is certainly useful in proving relations between semantics in the different categories of Petri nets [W3], event structures [W1,2], trees and transition systems. But also, I hope that the projection functions will be useful in formalising the practice of proving properties of a parallel composition by projecting-down to the component processes, proving properties there and then combining the properties to yield the required proof.

10. Related work.

The paper and report [W1, W2] show how the above results for trees hold in the more general framework of event structures. Event structures are related to Petri nets in [NPW1, 2]. They exhibit the causal independence and dependence of events and provide a basic model of parallel processes which does not rely on interleaving. In [W1,2] it is shown that they bear a smooth relation with trees; there is a natural interleaving , or serialising, operation on event structures which essentially imposes an extra causal constraint on the occurrence of events by ensuring events occur synchronised, in-step, with the ticks of a clock—it is a synchronous product on event structures. Then one can for example prove easily that a noninterleaving semantics for $\mathbf{Proc}_L$ in terms of labelled event structures interleaves to the synchronisation tree semantics we provide here. The recent paper [W3], on a new category of Petri nets, extends the work of [W1,2] and the work here. All the different categories are related by adjunctions so we can go quite far in translating between the different modes of expression.

The categories here and those mentioned above might be criticised for being too concrete because they distinguish too many processes. For example $S + S$ is not generally isomorphic to S even though it is hard to see a programming context in which they could be distinguished. Hopefully there are categories with objects which reflect a more abstract notion of behaviour with pleasant relations to the ones here. In [HN] it is shown how equivalence classes with respect to three natural equivalences on behaviour can be represented by a form of labelled tree. In [LP] morphisms very like those here are defined on equivalence classes of trees with respect to Milner's observational equivalence, which essentially treats τ-labelled events as invisible.

And then there are relations with path expressions and trace languages ([CH], [LTS]). Obviously a synchronisation tree determines a set of sequences of labels showing the possible communications. Only recently I noticed that ideas very similar to that of the morphisms presented here are found in the literature on languages of traces used to model concurrent processes—see [KGR].

In [M3] the finite delay property is considered for a synchronous calculus with an Abelian group of actions. The basic idea is to prune away disallowed infinite derivations from the labelled-transition-system semantics. One can generalise synchronisation trees to reflect this in the unfolding. Take a generalised tree to consist of finite and infinite sequences. Infinite sequences hang as limit points at the ends of ω-chains of nodes. By not insisting that every ω-chain of nodes have a limit one specifies by their absence those infinite derivations which are not permitted. In a way exactly analogous to the above one obtains a category of generalised trees whose product, when labelled appropriately, is the parallel composition; it coincides with the unfolding of the transition system given in [M3] with the correct infinite derivations removed. A transition system semantics similar to Milner's is presented by Plotkin in [P3] to give an operational semantics to constructs like a fair parallel operation. Interestingly in proving that the operational and denotational semantics are equivalent Plotkin uses projection functions from the parallel composition to the component processes.

As indicated in the previous section one can obtain more abstract semantics by "filtering-out" those properties of interest for a specific problem. (See the work [HN] for a good example of this idea. Think of a property as an assertion one might make about the behaviour of a program.) This begs two questions: Is there a class of basic models from which all interesting properties can be extracted? What are the interesting properties of concurrent programs? Petri nets and event structures are more basic models than trees because they express much more about the causal relations between events. It is not yet clear however what interesting class of assertions force one to use event structures or nets instead of trees.

Unfortunately trees, event structures and Petri nets are indifferent, as they stand, to notions of divergence as presented for example in [HP] and [HN]. This means that a satisfaction relation defined between trees or event structures and assertions about their inevitable behaviour cannot respect an approximation ordering on trees or event structures. In order to capture divergence in event structures one needs somehow to extend their structure to include local places of growth, just as how, with trees sometimes $\perp$ is put at the leaf-nodes to show how they may extend in the approximation ordering. At first glance this idea is very like that of places in concrete data structures—see [KP], [BC], [W].

Appendix The proof of theorem 4.13.

Theorem. *Let A and B be synchronisation algebras. The construction $A \times B$, h_A, h_B is a categorical product of A and B in* **SA**. *The construction $A \otimes B$, h'_A, h'_B is a categorical product of A and B in the subcategory with strict homomorphisms.*

Proof. Let $A = (L_A, \bullet_A, *_A, 0_A)$ and $B = (L_B, \bullet_B, *_B, 0_B)$ be synchronisation algebras.

We first show $A \times B$, h_A, h_B is a product in the category **SA**. We use the notation of 4.11.

We make sure $A \times B$ is an S.A.: It is obvious that $\bullet$ is commutative. The following steps show $\bullet$ is associative:

$$\begin{aligned}
(\alpha,\beta)\bullet((\alpha',\beta')\bullet(\alpha'',\beta'')) &= \begin{cases} (\alpha,\beta)\bullet 0 & \text{if } \alpha'\bullet\alpha'' = 0_A \text{ or } \beta'\bullet\beta'' = 0_B, \\ (\alpha,\beta)\bullet(\alpha'\bullet\alpha'', \beta'\bullet\beta'') & \text{otherwise} \end{cases} \\
&= \begin{cases} 0 & \text{if } \alpha'\bullet\alpha'' = 0_A \text{ or } \beta'\bullet\beta'' = 0_B, \\ 0 & \text{if } \alpha\bullet\alpha'\bullet\alpha'' = 0_A \text{ or } \beta\bullet\beta'\bullet\beta'' = 0_B, \\ (\alpha\bullet\alpha'\bullet\alpha'', \beta\bullet\beta'\bullet\beta'') & \text{otherwise} \end{cases} \\
&= \begin{cases} 0 & \text{if } \alpha\bullet\alpha'\bullet\alpha'' = 0_A \text{ or } \beta\bullet\beta'\bullet\beta'' = 0_B, \\ (\alpha\bullet\alpha'\bullet\alpha'', \beta\bullet\beta'\bullet\beta'') & \text{otherwise} \end{cases} \\
&= ((\alpha,\beta)\bullet(\alpha',\beta'))\bullet(\alpha'',\beta'').
\end{aligned}$$

Clearly $L \setminus \{*,0\} \neq 0$; by definition $(0_A, 0_B)\bullet(\alpha,\beta) = (0_A, 0_B)$; by definition $*\bullet* = (*_A\bullet*_A, *_B\bullet*_B) = *$ while $(\alpha,\beta)\bullet(\alpha',\beta') \Rightarrow \alpha\bullet\alpha' = *_A \;\&\; \beta\bullet\beta' = *_B \Rightarrow \alpha = \alpha' = *_A \;\&\; \beta = \beta' = *_B$ so $*$ is the unique divisor of $*$.

We check that the projections h_A and h_B are homomorphisms. Suppose $(\alpha,\beta)\bullet(\alpha',\beta') \neq 0$. Then $(\alpha,\beta)\bullet(\alpha',\beta') = (\alpha\bullet\alpha', \beta\bullet\beta')$ where $\alpha\bullet\alpha' \neq 0_A$ and $\beta\bullet\beta' \neq 0_B$. Thus $h_A((\alpha,\beta)\bullet(\alpha',\beta')) = h_A(\alpha\bullet\alpha', \beta\bullet\beta') = \alpha\bullet\alpha' = h_A((\alpha,\beta))\bullet h_A((\alpha',\beta'))$. Also $h_A(\alpha,\beta) = 0_A \Leftrightarrow \alpha = 0_A \Leftrightarrow \alpha = 0_A \;\&\; \beta = 0_B$ for $(\alpha,\beta) \in L$. And $h_A(*) = h_A(*_A, *_B)$, which shows that h_A is a homomorphism. Similarly h_B is a homomorphism.

Assume in there are homomorphisms $f_A : C \to A$ and $f_B : C \to B$ for a synchronisation algebra $C = (L_C, \bullet_C, *_C, 0_C)$. In order order to show $A \times B$, h_A, h_B is a product we require there exists a unique $f : C \to A \times B$ making the following diagram commute:

$$\begin{tikzcd}
 & A\times B \arrow[dl, "h_A"'] \arrow[dr, "h_B"] & \\
A & & B \\
 & C \arrow[ul, "f_A"] \arrow[ur, "f_B"'] \arrow[uu, "f"] &
\end{tikzcd}$$

Define $f(c) = (f_A(c), f_B(c))$. Clearly provided f is a homomorphism it is the unique one such that the above diagram commutes. If $c \in L_C$ then either $c = 0_C$ & $f(c) = (f_A(c), f_B(c)) = 0$ or $c \neq 0$ & $f_A(c) \neq 0_A$ & $f_B(c) \neq 0_B$ so $f(c) \in L$, making f a function $L_A \to L_B$. We now argue that f is also a homomorphism. Suppose $c\bullet c' \neq 0$. Then $f(c\bullet c') = (f_A(c\bullet c'), (f_B(c\bullet c')) = (f_A(c)\bullet f_A(c'), (f_B(c)\bullet f_B(c'))$ where $f_A(c)\bullet f_A(c') \neq 0_A$ and $f_B(c)\bullet f_B(c') \neq 0_B$ as f_A and f_B are homomorphisms. Therefore $f(c\bullet c') = (f_A(c), f_B(c))\bullet(f_A(c'), f_B(c')) = f(c)\bullet f(c')$. We have $f(c) = 0 \Leftrightarrow f_A(c) = 0_A \;\&\; f_B(c) = 0_B \Leftrightarrow c = 0_C$. Also $f(*_C) = (f_A(*_A), f_B(*_B)) = (*_A, *_B)$. And so f is a homomorphism, as required for $A \times B$, h_A, h_B to be a product in **SA**.

The verification that $A \otimes B$, h'_A, h'_B is a product in the subcategory is so similar that we omit it; one simply checks that the constructions stay inside the subcategory. ∎

Acknowledgements

Thanks are due to Mogens Nielsen of the Computer Science Department, Aarhus University, Denmark for encouragement and many helpful discussions. This work was supported in part by a postdoctoral fellowship from the Royal Society of Great Britain, to work at Aarhus, and in part by funds from the Computer Science Department of Carnegie-Mellon University, and by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539. The views and conclusions contained in it are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

References

[AM] Arbib, M.A.,and Manes,E.G., Arrows, Structures and Functors, The categorical imperative. Academic Press (1975).

[B] Brookes, S.D., On the relationship of CCS and CSP, ICALP 1983.

[BC] Berry, G. and Curien, P.L., Sequential algorithms on concrete data types, Report of Ecole Nationale Supérieure des Mines de Paris, Centre de Mathématiques Appliquées, Sophia Antipolis (1981).

[Bra] Brauer, W.(Ed.), Net Theory and Applications, Springer-Verlag Lecture Notes in Comp. Sci., vol.84 (1980).

[CES] Clarke, E.M., Emerson, E.A., Sistla, A.P., Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications: A Practical Approach, to appear in Proceedings, 10^{th} ACM Conference on Principles of Programming Languages (1983)

[CH] Campbell, R. H.,and Habermann, A. N.,, The Specification of Process Synchronisation by Path Expressions. Springer-Verlag Lecture Notes in Comp. Sc. Vol.16 (1974).

[CP] Cardelli, L., and Plotkin, G., An Algebraic Approach to VLSI design. In VLSI 81, Academic Press (1981).

[Grä] Gratzer, G., Universal Algebra. Van Nostrand University series in Higher Mathematics (1968).

[HBR] Hoare, C.A.R., Brookes, S.D., and Roscoe, A.W., A Theory of Communicating Processes, Technical Report PRG-16, Programming Research Group, University of Oxford (1981); to appear also in JACM.

[HM] Hennessy, M.C.B. and Milner, R., On observing nondeterminism and concurrency, Springer LNCS Vol. 85. (1979).

[HN] Hennessy, M.C.B., and de Nicola, R., Testing Equivalences for Processes, Internal Report, University of Edinburgh, (July 1982).

[HP1] Hennessy, M.C.B. and Plotkin, G., A term model for CCS, Proceedings of the 9^{th} Conference on Mathematical Foundations of Computer Science, Springer-Verlag LNCS Vol. 88. (1980)

[KP] Kahn, G., and Plotkin, G., Domaines Concrètes, Rapport IRIA-LABORIA, No.336 (1978).

[KGR] Knuth, E., Györy, Gy.,and Ronyai, L., A Study of the Projection Operation. Proc. of workshop on Petri nets, Springer-Verlag Informatik-Fachberichte Vol. 52 (1982).

[LP] Labella, A., and Peterossi, A., Towards a Categorical Understanding of Parallelism. Report of Istituto di Analisi dei Sistemi ed Informatica del C.N.R., Rome (1983).

[LTS] Lauer,P., Torrigiani, P.,and Shields, M., COSY, a system specification language based on paths and processes. Acta Informatica 12 (1979).

[Mac] Maclane, S., Categories for the Working Mathematician. Graduate Texts in Mathematics,Springer-Verlag (1972).

[M1] Milner, R., A Calculus of Communicating Systems. Springer-Verlag Lecture Notes in Comp. Sc. vol. 92 (1980).

[M2] Milner, R., On relating Synchrony and Asynchrony, Dept. of Comp. Sci. report, University of Edinburgh (1980).

[M3] Milner, R., A finite delay operator in Synchronous CCS, Internal Report CSR-116-82, University of Edinburgh (1982)

[Mi] Milne, G., Synchronised Behaviour Algebras; a model for interacting systems. Report of Comp. Sc. Dept., University of Southern California (1979).

[NPW1] Nielsen, M., Plotkin, G., Winskel, G., Petri nets, Event structures and Domains. Proc. Conf. on Semantics of Concurrent Computation, Evian, Springer-Verlag Lecture Notes in Comp. Sc. 70 (1979).

[NPW2] Nielsen, M., Plotkin, G., Winskel, G., Petri nets, Event structures and Domains, part 1 . Theoretical Computer Science, vol. 13 (1981) pp.85–108.

[P1] Plotkin, G., A structural approach to operational semantics. DAIMI FN-19 Comp. Sc. Dept., Aarhus University (1981).

[P2] Plotkin, G., A Powerdomain for countable non-determinism, Springer-Verlag Lecture Notes in Comp. Sc. 140 (1982).

[S] Scott, D., Domains for Denotational Semantics, Springer-Verlag Lecture Notes in Comp. Sc. 140 (1982).

[W] Winskel, G., Events in Computation. Ph.D. thesis, University of Edinburgh (1980).

[W1] Winskel, G., Event structure semantics of CCS and related languages, Springer-Verlag Lecture Notes in Comp. Sc. 140 (1982).

[W2] Winskel, G., Event structure semantics of CCS and related languages, Report of the Computer Sc. Dept., University of Aarhus, Denmark (1982).

[W3] Winskel, G., A New Definition of Morphism on Petri Nets. To be submitted (1983).