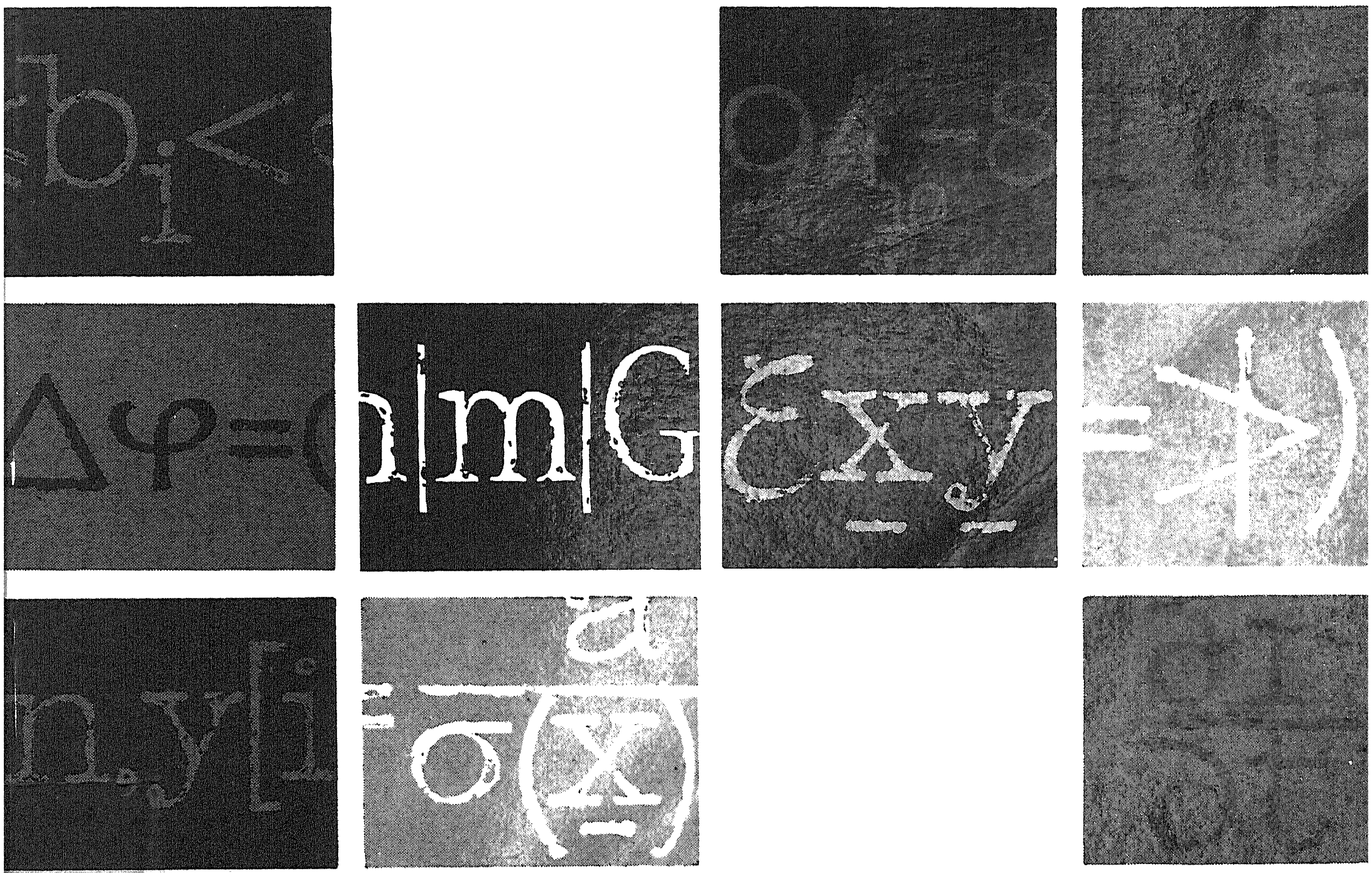
FOUNDATIONS OF COMPUTER SCIENCE IV

DISTRIBUTED SYSTEMS : PART 2, SEMANTICS AND LOGIC

J.W. DE BAKKER (ed.)
J. VAN LEEUWEN (ed.)



MATHEMATICAL CENTRE TRACTS 159

Printed at the Mathematical Centre, Kruislaan 413, Amsterdam, The Netherlands.

The Mathematical Centre, founded 11th February 1946, is a non-profit institution for the promotion of pure and applied mathematics and computer science. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

MATHEMATICAL CENTRE TRACTS 159

FOUNDATIONS OF COMPUTER SCIENCE IV

DISTRIBUTED SYSTEMS:
PART 2, SEMANTICS AND LOGIC

J.W. DE BAKKER (ed.)
J. VAN LEEUWEN (ed.)

MATHEMATISCH CENTRUM AMSTERDAM 1983

1980 Mathematics subject classificatie: 68B10, 68C05

1982 CR. Categories: B.7.1, D.1.3, D.3.1, D.3.3, F.3.1, F.3.2, F.3.3
ISBN 90 6196 255 2

CONTENTS

AUTHORS' CURRENT ADDRESSES

K.R. Apt	LITP, Université Paris 7, 2, Place Jussieu, 75251 Paris, France
J.W. de Bakker	Mathematisch Centrum, Kruislaan 413, 1098 SJ Amsterdam, the Netherlands
Z. Manna	Computer Science Department, Stanford University, Stanford, CA, USA, and applied Mathematics Department, The Weizmann Institute of Science, Rehovot, Israel
D. Park	Department of Computer Science, University of Warwick, Coventry CV4 7AL, England
A. Pnueli	Applied Mathematics Department, The Weizmann Institute of Science, Rehovot, Israel
M. Rem	Department of Mathematics and Computing Science, Eindhoven Technological University, P.O. Box 513, the Netherlands
J.I. Zucker	Department of Computer Science, SUNY at Buffalo, 4226 Ridge Lea Road, Amherst, NY 14226, USA

PARTIALLY ORDERED COMPUTATIONS, WITH APPLICATIONS TO VLSI DESIGN

M. Rem
Eindhoven Technological University, Eindhoven, the Netherlands

1. INTRODUCTION

VLSI (Very Large Scale Integration) is a medium for the execution of computations. Like all integrated circuits, a VLSI circuit consists of transistors, pads, and connections between them. The pads are points via which the communication with the circuit's environment takes place. A transistor can best be compared with an on-off switch. Over the years the transistors in integrated circuits have become smaller, thus allowing more transistors to be put on the same silicon chip. Making a transistor smaller shortens its switching delay. In VLSI chips the switching delays are so short compared to the delays in connections that the latter can no longer be ignored. The properties of the VLSI medium that are relevant to our exposition will be discussed in Section 3.

A general-purpose computer is also a medium for the execution of computations. In what respects is the VLSI medium different from traditional implementations? What makes VLSI an interesting subject? There are four differences we would like to point out:

(1) VLSI is a concurrent medium: it offers a surface of thousands, and in the future possibly millions, of simultaneously active computing elements.

(2) VLSI is a two-dimensional medium. The computing elements that together constitute a component have to be laid out in the plane. In a traditional implementation the computation has to be mapped onto a one-dimensional store. We have good techniques of achieving the latter, but the difference between one and two dimensions is the difference between lining up and solving a jigsaw puzzle.

(3) VLSI allows only limited connectivity. If a value computed in one place has to be used in a different place, a connection between the two places

must be present. But the VLSI medium does not allow an arbitrary placement of connections.

(4) The VLSI medium is not order-preserving. The connections in a circuit exhibit delays. When signals are sent via different connections from one place to another they can, of course, not be received before they are sent. But they may be received in an order that is different from the order in which they were sent.

There is a fifth problem, but that one is not unique to VLSI design. It is a problem that permeates all of computing science: complexity bridling. Uncontrolled concurrency causes uncontrolled complexity, even for moderately sized components. The well-known technique for avoiding complexity consists in partitioning - also known as modularizing - the components into subcomponents. Subcomponents are again components. The whole component thus exhibits a tree structure. Such tree-structured components are known as hierarchical components. Given the specifications of the subcomponents and the way in which the subcomponents constitute the component, we must be able to deduce the specification of the whole component. In view of the central role of complexity bridling, hierarchical composition will be used all through these notes.

A hierarchical design method not only helps to avoid complexity, it also alleviates some of the problems mentioned earlier. A component may be looked upon as a graph: the transistors (and the pads) are the vertices, and the connections the edges. Hierarchical components give rise to tree-like graphs. Balanced binary trees can be positioned in the plane very well (REM 79). The fact that we have tree-like graphs may, consequently, alleviate the layout problem. A tree is the graph that has the fewest edges while still being connected. Our confinement to tree-like graphs seems thus to result in circuits that satisfy VLSI's property of limited connectivity.

A component consists of subcomponents and connections between them. A connection equates the output of one component with the input of another. Therefore, we shall have to take input and output events into account. Actually, we shall phrase the meaning (the semantics) of a component as a relation between input and output events. Input and output events are partially ordered. Events that are not ordered are called concurrent. We do not postulate that certain occurrences of events must overlap in time. We take the complementary view: certain occurrences must be ordered. The more partial the order - i.e., the more concurrency - the more freedom we leave to the implementation to have the occurrences of events overlap in time. For the formulation of the meaning of a component we shall, rather than the partial order itself, use the set of all sequences of input and output events that comply with the partial order. Such a sequence of events is called a trace. The meaning of a component is a set of traces, each trace being a finite-length sequence of events. This approach bears resemblance to path expressions (CAMPBELL & HABERMANN 74). It differs from COSY (LAUER 81) in that we use traces rather than vectors of traces. The reason for our preference is that vectors of traces exhibit the internal structure of components and thus do not lend themselves well to hierarchical composition. Trace theory is the subject of Section 2.

How do we cope with delays in connections? The traditional method is to ignore them. But, as we said earlier, that cannot be done in VLSI (SEITZ 79). We could try to estimate the delay times, but that would make the correct functioning of our components dependent on the way in which the connections are placed in the silicon chip. We take the position that delay times are unbounded (STUCKI & COX 79), and that the components should function correctly irrespective of the delay times. Such components are called delay-insensitive (or self-timed). They are discussed in Section 4. Making our components delay-insensitive will also alleviate the layout problem, since it does not impose upper bounds on the lengths of connections.

2. TRACE THEORY

In this section we discuss the concept of a trace structure. We introduce composition rules and we show how trace structures can be used to define the semantics of partially ordered computations.

2.1. Trace structures and composition rules

Let Σ be an infinite set of symbols. A *trace structure* is a pair $<T,A>$, in which A is a finite subset of Σ, and $T \subset A^*$. A^* denotes, as usual, the set of all finite-length sequences of elements of A, including the empty sequence which is denoted by ε. A is called the *alphabet* of the trace structure, and T its *trace set*. The elements of T are called *traces*.

Let $S = <T,A>$ and $S' = <T',A'>$ be trace structures, and let h be a function $h: A^* \to (A')^*$ such that

(i) $h(\varepsilon) = \varepsilon$;
(ii) $h(ab) = h(a)h(b)$;

then h is a homomorphism from S to S'. (Concatenation is denoted by juxtaposition.) The homomorphism π_B given by $\pi_B(a) = a$ for $a \in B$ and $\pi_B(a) = \varepsilon$ for $a \notin B$ is called the *projection* on alphabet B. In words, $\pi_B(t)$ is trace t from which all symbols not in B have been deleted. The projection of a trace set T on alphabet B is the trace set

$$\{\pi_B(t) \mid t \in T\}.$$

It will be denoted by $\pi_B(T)$. The projection of a trace structure $S = \langle T,A\rangle$ on alphabet B is the trace structure

$$\langle \pi_B(T),\ A \cap B\rangle.$$

We denote it by $\pi_B(S)$.

PROPERTY 2.1. $\pi_A \circ \pi_B = \pi_{A\cap B}$

PROPERTY 2.2. Let $s \in A^*$ and $t \in B^*$ such that $\pi_B(s) = \pi_A(t)$. Then

$$(\exists\, u \in (A\cup B)^*:\ \pi_A(u) = s \wedge \pi_B(u) = t).$$

PROOF. A trace u can be constructed as follows. If s or t is ε, take as u the other one. If s and t both start with a symbol in $A \cap B$ they start with the same symbol. Take that as the first symbol of u. Otherwise, take a first symbol of s or t that is not in $A \cap B$. Construct the remainder of u in the same way out of the remainders of s and t. □

The *p-composition* of two trace structures $S = \langle T,A\rangle$ and $S' = \langle U,B\rangle$, denoted $S\ \underline{p}\ S'$, is the trace structure

$$\langle\{t \in (A\cup B)^* \mid \pi_A(t) \in T \wedge \pi_B(t) \in U\},\ A \cup B\rangle.$$

Whenever obvious from the context, the alphabets may be deleted from the trace structures and we simply talk of the p-composition of trace sets.

EXAMPLE 2.1. (The alphabets associated with the following trace sets are assumed to be chosen as small as possible.)

$$\{ab, cd\} \underline{p} \{be, df\} = \{abe, cdf\}$$

$$(\{ab\} \underline{p} \{ac\}) \underline{p} \{ac\} = \{abc, acb\} \underline{p} \{ac\} = \{abc, acb\}$$

$$\{ab\} \underline{p} (\{ac\} \underline{p} \{ac\}) = \{ab\} \underline{p} \{ac\} = \{abc, acb\}$$

<u>PROPERTY 2.3.</u> P-composition is idempotent, symmetric, and associative.

As an aside, we mention that p-composition can also be defined with inverse homomorphisms. The inverse of π_B, π_B^{-1}, is defined as follows. Let T be a trace set. Then

$$\pi_B^{-1}(T) = \{u \in \Sigma^* \mid \pi_B(u) \in T\}.$$

It satisfies

$$\pi_B(\pi_B^{-1}(T)) = T \text{ for } T \subset B^*.$$

The p-composition of S = <T,A> and S' = <U,B> is the trace structure

$$<\pi_{A \cup B}(\pi_A^{-1}(T) \cap \pi_B^{-1}(U)), A \cup B>.$$

Since regular sets are closed under (inverse) homomorphism and intersection (HOPCROFT & ULLMAN 69), we have the following property.

<u>PROPERTY 2.4.</u> If T and U are regular sets then $T \underline{p} U$ is a regular set.

<u>PROPERTY 2.5.</u> Let <T,A> be a trace structure. Then

$$\pi_A(T \underline{p} T') \subset T \quad \text{for all } T'.$$

<u>PROOF.</u> Let $u \in \pi_A(T \underline{p} T')$. Let $t \in T \underline{p} T'$ such that $u = \pi_A(t)$. Since $t \in T \underline{p} T'$, we have $\pi_A(t) \in T$ and, hence, $u \in T$. □

Property 2.5 with the inclusion replaced by equality will hold only for special choices of T and T'. This is reflected in the following definition. Two trace structures S = <T,A> and S' = <U,B> are said to *match* when $\pi_B(T) = \pi_A(U)$.

PROPERTY 2.6. Two trace structures $S = <T,A>$ and $S' = <U,B>$ match if and only if

(2.1) $$\pi_A(T \,\underline{p}\, U) = T \wedge \pi_B(T \,\underline{p}\, U) = U.$$

PROOF. (i) Assume S and S' match. We prove $T \subset \pi_A(T \,\underline{p}\, U)$. Equality then follows from Property 2.5. For reasons of symmetry, we then also have $\pi_B(T \,\underline{p}\, U) = U$.
Let $t \in T$. Let $u \in U$ be such that $\pi_B(t) = \pi_A(u)$. The fact that S and S' match implies that such a u exists. Let $s \in (A \cup B)^*$ be such that $\pi_A(s) = t \wedge \pi_B(s) = u$. From Property 2.2 we know that such an s exists. Then $s \in T \,\underline{p}\, U$, which, combined with $t = \pi_A(s)$, yields $t \in \pi_A(T \,\underline{p}\, U)$.
(ii) Assume (2.1).

$$\begin{aligned}\pi_A(U) &= \pi_A(\pi_B(T \,\underline{p}\, U)) \\ &= \pi_{A\cap B}(T \,\underline{p}\, U) \\ &= \pi_B(\pi_A(T \,\underline{p}\, U)) \\ &= \pi_B(T) \quad . \qquad \Box\end{aligned}$$

PROPERTY 2.7. The property of matching is reflexive and symmetric, but not transitive.

EXAMPLE 2.2.

$S_0 = <\{abc, de\}, \{a,b,c,d,e\}>$
$S_1 = <\{bcf, dg\}, \{b,c,d,f,g\}>$
$S_2 = <\{cbf, dg\}, \{b,c,d,f,g\}>$
$S_3 = <\{fg \ , \{f,g\}>$
$S_4 = <\{bcf,\varepsilon\}, \{b,c,d,f,g\}>$
$S_5 = <\{bcf,\varepsilon\}, \{b,c,f\}>$
$S_6 = <\{bcf\}, \{b,c,f\}> .$

Trace structure S_0 matches S_1, S_3, and S_5. It does not match the other three trace structures.

Let $S = \langle T,A\rangle$ be a trace structure. We call $s \in A^*$ a *prefix* of trace $t \in T$ if $(\exists u \in A^*: t = su)$. The set of all prefixes of traces in T is denoted by PREF(T). A trace set T satisfying PREF(T) = T is called *prefix-closed*.

Let a relation $\sim$ be defined on PREF(T) by $s \sim t$ if and only if

$$\{u \in A^* \mid su \in T\} = \{u \in A^* \mid tu \in T\}.$$

Relation $\sim$ is a right congruence and, hence, an equivalence relation. We call the equivalence classes of $\sim$ the *states* of S. We denote the equivalence class (state) of which s is a member by $[s]_S$. Whenever S is obvious from the context, it is omitted. For regular T the relation $\sim$ is of finite index (HOPCROFT & ULLMAN 69), or, phrased differently, if T is a regular set the number of states is finite.

Trace structures are used for defining the semantics of partially ordered computations. The trace structure of a computation is the composition of the trace structures of the subcomputations. The trace structure of the composite should not reflect the internal relations of the subcomputations. Therefore, we introduce a second composition operation, q-composition, which is the p-composition followed by the elimination of all common symbols.

The *q-composition* of two trace structures $S = \langle T,A\rangle$ and $S' = \langle U,B\rangle$, notation $S \underline{q} S'$, is the trace structure

$$\langle \pi_{A \div B}(T \underline{p} U),\ A \div B\rangle$$

($\div$ denotes symmetric set difference, i.e. $A \div B = (A \cup B) \setminus (A \cap B)$.) This composition operator differs from the one in MILNER 80 in that the latter one replaces common symbols by "silent moves" τ.

PROPERTY 2.8. Q-composition is symmetric.

EXAMPLE 2.3. (cf. Example 2.1)

$$\{ab, cd\} \underline{q} \{be, de\} = \{ae, cf\}$$

$$(\{ab\} \underline{q} \{ac\}) \underline{q} \{ac\} = \{bc, cb\} \underline{q} \{ac\} = \{ab, ba\}$$

$$\{ab\} \underline{q} (\{ac\} \underline{q} \{ac\}) = \{ab\} \underline{q} \{\varepsilon\} \quad = \{ab\}.$$

The example above shows that q-composition is not associative. In order to achieve associativity, we must have, for trace structures $\langle T,A \rangle$, $\langle U,B \rangle$, and $\langle V,C \rangle$,

$$\pi_{A \div B \div C}(\pi_{A \div B}(T \, \underline{p} \, U) \, \underline{p} \, V) = \pi_{A \div B \div C}(T \, \underline{p} \, U \, \underline{p} \, V).$$

This is true if the symbols deleted by $\pi_{A \div B}$ do not occur in the traces in V, i.e. if $A \cap B \cap C = \emptyset$. This is expressed by the following Property.

PROPERTY 2.9. For trace structures $S = \langle T,A \rangle$, $S' = \langle U,B \rangle$, and $S'' = \langle V,C \rangle$ such that $A \cap B \cap C = \emptyset$ we have

$$(S \, \underline{q} \, S') \, \underline{q} \, S'' = S \, \underline{q} (S' \, \underline{q} \, S'').$$

PROPERTY 2.10. The p-composition and the q-composition of prefix-closed trace sets are prefix-closed.

PROPERTY 2.11. Let $S = \langle T,A \rangle$ and $S' = \langle U,B \rangle$ be two matching trace structures. Then

$$\mathrm{PREF}(T \, \underline{q} \, U) = \mathrm{PREF}(T) \, \underline{q} \, \mathrm{PREF}(U).$$

There is a special trace structure called SYNC. Let k be a natural number and let a and b be two distinct symbols. $\mathrm{SYNC}_k(a,b)$ is the trace structure with $\{a,b\}$ as its alphabet and

$$\{t \in \{a,b\}^* \mid 0 \le \#_a t' - \#_b t' \le k \text{ for every prefix } t' \text{ of } t\}$$

as its trace set. ($\#_a t'$ denotes the number of occurrences of a in t'.) $\mathrm{SYNC}_k(a,b)$ is, consequently, prefix-closed. It has k + 1 states, viz. the elements of the set

$$\{[a^i] \mid 0 \le i \le k\}$$

($a^0 = \varepsilon$; $a^{i+1} = a^i a$).

PROPERTY 2.12. Let a, b, and c be three distinct symbols. Then

$$\mathrm{SYNC}_k(a,b) \, \underline{q} \, \mathrm{SYNC}_m(b,c) = \mathrm{SYNC}_{k+m}(a,c).$$

A number of proofs that have been omitted in these notes may be found in VAN DE SNEPSCHEUT 82.

2.2. Programs denoting partially ordered computations

We introduce a program notation. Every program denotes a partially ordered computation and thus defines a trace structure. A program is a hierarchy of commands: a program is a component, every component consists of a command and a number of subcomponents with relations between their alphabets. First we define the trace set TR(S) defined by a command S. Since a command has one of five possible forms, we define TR inductively by five cases. (b denotes a symbol; S_0, S_1, and S denote commands.)

(i) A symbol is a command

$$TR(b) = \{b\}$$

(ii) $S_0 \mid S_1$ is a command

$$TR(S_0|S_1) = TR(S_0) \cup TR(S_1)$$

(iii) S_0, S_1 is a command

$$TR(S_0,S_1) = TR(S_0) \ \underline{q} \ TR(S_1)$$

(iv) $S_0 ; S_1$ is a command

$$TR(S_0;S_1) = \{t_0t_1 \mid t_0 \in TR(S_0) \wedge t_1 \in TR(S_1)\}$$

(v) S^* is a command

$$TR(S^*) = TR(S)^*.$$

Except for (iii), the trace structure of a command has as its alphabet the union of the alphabets of its parts. Where appropriate, TR(S) denotes the trace structure of S. Notice that trace sets of commands are not necessarily profix-closed.

If $TR(S_0)$ and $TR(S_1)$ are trace sets with disjoint aphabets we have

$$\begin{aligned} TR(S_0,S_1) &= TR(S_0)\ \underline{q}\ TR(S_1) \\ &= TR(S_0)\ \underline{p}\ TR(S_1). \end{aligned}$$

In this case, both compositions amount to the shuffle (GINSBURG 66) operation. Since regular sets are closed under q-composition, TR(S), for S a command, is a regular set.

In order to save on parentheses, we introduce the rule that the comma has the highest priority, followed by the semicolon, and then the vertical bar:

$$\begin{aligned} S_0,S_1 \mid S_2 &= (S_0,S_1) \mid S_2 \\ S_0,S_1;S_2 &= (S_0,S_1);S_2 \\ S_0;S_1 \mid S_2 &= (S_0;S_1) \mid S_2 \end{aligned}$$

PROPERTY 2.13.

(i) $TR(S_0|S_1) = TR(S_1|S_0)$

(ii) $TR(S_0,S_1) = TR(S_1,S_0)$

(iii) $TR((S_0|S_1)|S_2) = TR(S_0|(S_1|S_2))$

(iv) $TR((S_0;S_1);S_2) = TR(S_0;(S_1;S_2))$

(v) $TR((S_0,S_1),S_2) = TR(S_0,(S_1,S_2))$ provided $\bigcap_{i=0}^{2}(\text{alphabet of } S_i) = \emptyset$

(vi) $TR(S_0;(S_1|S_2)) = TR(S_0;S_1|S_0;S_2)$

(vii) $TR((S_0|S_1);S_2) = TR(S_0;S_2|S_1;S_2)$

(viii) $TR(S_0,(S_1|S_2)) = TR(S_0,S_1|S_0,S_2)$

(ix) $TR(S^{**}) = TR(S^*)$

(x) $TR(S^*;S^*) = TR(S^*)$.

Property 2.13(vi) does not hold in MILNER 80; there the commands $S_0;(S_1|S_2)$ and $S_0;S_1|S_0;S_2$ are considered to be different.

A program is a component. The simplest form a component can have is a single command. Syntactically, such a component is of the form

$$\underline{\text{com}}\ C(\text{"alphabet of S"}):\ S\ \underline{\text{moc}}$$

C is the name of the component. Its trace structure is

<PREF(TR(S)), alphabet of S> .

Thus, components have a prefix-closed trace set. In HOARE 78 PREF is applied to trace sets of commands. This requires the introduction of a termination symbol "√" to cater for sequential composition (the semicolon). A termination symbol at the end of a trace indicates that it is not a trace brought about by PREF. Since we do not have sequential composition for components, there is no need to introduce a termination symbol.

The following is an example of a component.

com binsem(v,p): $(v;p)^*$ moc.

Its trace structure is $\mathrm{SYNC}_1(v,p)$.

We now turn to components that have subcomponents as well as a command. Each subcomponent has a name and a type. The type of a subcomponent is a component. To differentiate between the alphabets of the subcomponents we introduce composite symbols. A composite symbol is a symbol s.a, in which s is a name of a subcomponent and a a symbol in the alphabet of that subcomponent's type. Symbols that are not composite are called simple. The alphabet of a component contains simple symbols only.

Let s be the name of a subcomponent of type C. Let C have <T,A> as its trace structure. The trace structure of s is then

<s.T, s.A>

in which s.T is the trace set obtained from T by replacing in each trace of T every symbol a by the composite symbol s.a. The alphabet s.A is likewise obtained from A by changing each symbol a into s.a.

Syntactically, a component with subcomponents is of the form

com C(A):

    sub s_0: C_0, ..., s_{n-1}: C_{n-1}

    S

moc

A is an alphabet of simple symbols. Component C has n subcomponents s_i $(0\leq i<n)$. The n names s_i must be distinct. The type of s_i is C_i. Let C_i have alphabet A_i, then S is a command with

$$A \cup \bigcup_{i=0}^{n-1} s_i.A_i$$

as its alphabet. The trace set of C is given by

$$\text{(2.2)} \qquad TR(C) = PREF(TR(S))\ \underline{q}\ s_0.TR(C_0)\ \underline{q}\ \ldots\ \underline{q}\ s_{n-1}.TR(C_{n-1}).$$

Since the alphabets of the subcomponents are subsets of the alphabet of S, the alphabet of C is A and thus contains simple symbols only. Notice also that TR(C) is prefix-closed. Due to the prefixing with the subcomponent's name, the alphabets of the subcomponents are disjoint. Property 2.9 then ensures the associativity of the q-composition in (2.2).

In these notes we want to restrict ourselves to regular trace sets. Therefore, we do not allow components to be recursive. More precisely: we say that component C has component D as a composing part if C has a subcomponent that either is of type D or has D as a composing part. We do not allow components that have themselves as a composing part. This restriction makes (2.2) a nonrecursive equation.

The following is an example of a component with subcomponents.

```
com sexsem(v,p):
    sub b0,b1: binsem
    (v; b0.v)*, (b0.p; b1.v)*, (b1.p; p)*
moc
```

The second line is short for "sub b0: binsem, b1: binsem". According to Properties 2.11 and 2.12, the trace structure of sexsem is $SYNC_5(v,p)$.

By extending the alphabet of binsem with a third symbol we are able to achieve a stronger synchronization between the two subcomponents:

```
com binsem'(v,p,q): (v;p;q)* moc
com quinsem(v,p):
    sub b0,b1: binsem'
    (v; b0.v)*, (b0.p; b1.v; b0.q; b1.q)*, (b1.p; p)*
moc
```

Consider the p-composition of the trace sets of b0, b1, and the middle part of the command of quinsem:

(2.3) $(b0.v;\ b0.p;\ b0.q)^*$

(2.4) $(b1.v;\ b1.p;\ b1.q)^*$

(2.5) $(b0.p;\ b1.v;\ b0.q;\ b1.q)^*$

In each trace t of that p-composition the (i+2)nd b0.v follows, because of (2.3), the (i+1)st b0.q and hence, because of (2.5), the (i+1)st b1.v and hence, because of (2.4), the i-th b1.p. Hence

$$\#_{b0.v}\ t - \#_{b1.p}\ t \le 2 \ .$$

The q-composition of the three trace sets is SYNC_2(b0.v, b1.p). By dropping in (2.3) through (2.5) the symbols b0.q and b1.q, the q-composition would, according to Property 2.11, yield SYNC_3(b0.v, b1.p). Thus we have achieved a stronger synchronization between the two subcomponents. By q-composing SYNC_2(b0.v, b1.p) with the trace sets of the other two parts of the command of quinsem we get, again using Properties 2.11 and 2.12, SYNC_4(v,p) as the trace structure of quinsem.

We introduce a simpler mechanism to achieve stronger synchronization between (sub)components. Symbols a and b in different alphabets may be equated by adding an equation "a = b" to the component definition. The following component contains an example of this.

```
com quinsem'(v,p):
    sub b0,b1: binsem
    b0.p = b1.v
    (v; b0.v)*, (b1.p; p)*
moc
```

In quinsem' the alphabets of b0 and b1 are not disjoint. Their intersection contains one symbol.

We have thus arrived at the most general form a component can have

```
com C(A):
    sub $s_0$: $C_0$, ..., $s_{n-1}$: $C_{n-1}$
    $a_0 = b_0$, ..., $a_{m-1} = b_{m-1}$
    S
moc
```

Let C_i have alphabet A_i. The symbols occurring in the equations must be symbols of the alphabets A, $s_0.A_0$, ..., $s_{n-1}.A_{n-1}$. Two symbols occurring in the same equation must be chosen from two different alphabets. The trace set of C is again given by (2.2). Let B be the set of all symbols that occur in an equation. Each symbol in B must occur exactly once in the equations. The alphabet of S is $(A \cup \bigcup_{i=0}^{n-1} s_i.A_i)\backslash B$. Thus we achieve again that the q-composition in (2.2) is associative. We allow the command S to be empty. In that case, TR(S) in equation (2.2) is by definition equal to $\{\varepsilon\}$. Having defined the trace structure of a component with equations, we can use Properties 2.11 and 2.12 again to show that quinsem' has $SYNC_4(v,p)$ as its trace structure.

We now have two ways of expressing communication between (sub)components: by introducing an equation between symbols in their alphabets or by having the symbols occur in the command of the component. The former way is equivalent to the way communication is expressed in HOARE 78a. In MARTIN 81 the concept of "synchronization slack" is introduced. The slack is the number of steps two synchronized processes are allowed to be out of step. A slack = 0 between symbols a and b corresponds to an equation a = b in our formalism. A slack = k for $k > 0$ between a and b corresponds to a q-composition with the trace structure $\mathrm{SYNC}_k(a,b)$.

The following is an example of a component in which all symbols in the alphabets of the component and the subcomponents occur in the equations.

```
com trisem(v,p):
    sub b0,b1: binsem
    v = b0.v, b0.p = b1.v, b1.p = p
moc
```

Its trace structure is, again according to Properties 2.11 and 2.12, $\mathrm{SYNC}_2(v,p)$.

2.3. Examples of components

EXAMPLE 2.4.

$$\underline{\mathrm{com}}\ \mathrm{buf1}(x0,x1,y0,y1):\ (x0;\ y0 \mid x1;\ y1)^*\ \underline{\mathrm{moc}}$$

This component has three states: [ε], [x0], and [x1]. It may be interpreted as a one-bit buffer. The symbols x0 and x1 then stand for "receive a 0" and "deliver a 1", respectively. Using this interpretation, the three states stand for "buffer empty", "buffer contains a 0", and "buffer contains a 1".

EXAMPLE 2.5.

$$\underline{\mathrm{com}}\ \mathrm{boolvar}(x0,x1,y0,y1):\ (x0;\ y0^* \mid x1;\ y1^*)^*\ \underline{\mathrm{moc}}$$

This component has the same three states as buf1. However, in this case the state [ε] contains the trace ε only, and thus stands for "component uninitialized". The component may be interpreted as a boolean variable. Notice that the component is constructed in such a way that is must be initialized before it can inspected.

EXAMPLE 2.6.

```
com queue$_1$(x0,x1,y0,y1) : (x0; y0 | x1; y1)$^*$ moc
```

for $i > 1$:

```
com queue$_i$(x0,x1,y0,y1):
    sub q: queue$_{i-1}$
    q.y0 = y0, q.y1 = y1
    (x0; q.x0 | x1; q.x1)$^*$
moc
```

Component queue$_i$ ($i \geq 1$) is an i-bit buffer. For $i = 1$ it is equal to component buf1. For $i > 1$ it consists of a one-bit buffer between its inputs and the inputs to its subcomponent, as expressed by its command, and an (i-1)-bit buffer as a subcomponent. Notice that the component does not violate our restriction on recursion: queue$_i$ does not have queue$_i$ as a composing part.

We can replace the command of queue$_1$ by a suitable subcomponent and add the appropriate equations

```
com queue$_1$(x0,x1,y0,y1):
    sub b: buf1
    x0 = b.x0, x1 = b.x1, b.y0 = y0, b.y1 = y1
moc
```

We can change queue$_i$ in a similar fashion ($i>1$)

```
com queue$_i$(x0,x1,y0,y1):
    sub b: buf1, q: queue$_{i-1}$
    x0 = b.x0, x1 = b.x1, b.y0 = q.x0, b.y1 = q.x1,
    q.y0 = y0, q.y1 = y1
moc
```

Thus we can, if we wish, change any component into one that has either no subcomponents (and, consequently, no equations) or no command.

EXAMPLE 2.7. This example is a binary stack of depth i. A binary stack has an alphabet of four symbols: two opening parentheses, $(_0$ and $(_1$, and two closing parentheses, $)_0$ and $)_1$. Consider all well-nested sequences of these parenthese. For example,

$$(_0\ (_1\ )_1\ (_0\ (_1\ )_1\ )_0\ )_0\ (_1\ )_1$$

is well-nested, but

$$(_0\ )_1\ (_1\ )_0 \quad \text{and} \quad (_0\ (_1\ )_0\ )_1$$

are not. The trace set of a stack is the set of all prefixes of well-nested sequences. Our example will not be a general stack but a stack of depth i. Consider a prefix of a well-nested sequence. Its nesting level is defined as the number of opening parentheses minus the number of closing parentheses, and its nesting depth as the maximum of the nesting levels of its prefixes. The trace set of a stack of depth i is the set of all prefixes of well-nested sequences with nesting depth ≤ i.

In the component the opening parentheses are denoted by x0 and x1, and the closing parentheses by y0 and y1.

```
com stack$_1$(x0,x1,y0,y1): (x0, y0 | x1; y1)$^*$ moc
```

for i > 1:

```
com stack$_i$(x0,x1,y0,y1):
    sub s: stack$_{i-1}$
    ( (x0; s.x0 | x1; s.x1)$^*$;
      (x0; y0 | x1; y1);
      (s.y0; y0 | s.y1; y1)$^*$ )$^*$
moc
```

Component stack$_1$ is obviously a binary stack of depth 1. Now consider stack$_i$ for $i > 1$. Assume that subcomponent s is a binary stack of depth $i - 1$. Leave out the middle line of the command of stack$_i$. Then component stack$_i$ is, because of the exact matching of its symbols with the symbols of s, also a binary stack of depth $i - 1$. The effect of adding the middle line is that at every position in a trace where an opening parenthesis is immediately followed by a closing parenthesis one pair of matching parentheses is inserted between them. Hence, component stack$_i$ is a binary stack of depth i.

The program above defines a stack, but it also exhibits the usage of a stack, viz. the usage of subcomponent s. It is a complicated program. Its intricacy becomes apparent if one wonders how the component is "executed", i.e. how a trace is selected that matches the component's environment. Or more specifically: how is the number of steps for each repetition determined? Consider the repetition in the first line of the command of stack$_i$. Let S denote

```
(x0; s.x0 | x1; s.x1) ;
(x0; y0 | x1; y1) .
```

Then S has the same trace set as

```
 x0; (y0 | s.x0; S)
|x1; (y1 | s.x1; S).
```

We see that every trace in TR(S) starts with x0 or x1. The traces in the environment contain the symbols x0, x1, y0, and y1. (Neglect any other symbols for this discussion.) So the environment determines whether the first or the second line above is chosen. Assume the first line is chosen. Then next a choice must be made between y0 and "s.x0; S". If the environment has a trace with y0 as its next symbol y0 may be chosen and the repetition terminates. If there is a trace with x0 or x1 as its next symbol, "s.x0; S" may be chosen and the repetition continues. We are stuck when all traces have y1 as the next symbol: the trace sets of the component and the environment do not match. In the program of stack$_i$ than cannot occur: if s.y1 is a possible next symbol then so is s.y0.

EXAMPLE 2.8.

```
com fulladder(a0,a1,b0,b1,c0,c1,d0,d1,s0,s1):
    (a0,b0; d0,(c0; s0 | c1; s1)
    |a1,b1; d1,(c0; s0 | c1; s1)
    |(a0,b1 | a1,b0); (c0; d0,s1 | c1; d1,s0) )*
moc
```

This component may be interpreted as a full-adder element (cf. p. 250 of SEITZ 80). The pairs (a0,a1) and (b0,b1) represent the two bits to be added, (c0,c1) represents the carry-in, (d0,d1) the carry-out, and (s0,s1) the sum. The first line of the command is known as "carry-kill" and the second line as "carry-generate". In both cases the carry-out may precede the carry-in. The third line is known as "carry-propagate". In that case, of course, the carry-out has to follow the carry-in.

3. THE VLSI MEDIUM

It is our intention to realize the components defined in Section 2 as VLSI circuits. To gain insight in the problems associated with the translation of components into VLSI circuits we discuss the relevant properties of the VLSI medium. We consider in particular the realization of restoring logic circuitry in CMOS. An elementary introduction to integrated circuits may be found in CLARK 80.

3.1. CMOS

A Metal-Oxide-Semiconductor chip (MOS chip) is a thin layer of "substrate" with on top of it a network of conducting paths. A chip measures about 5 by 5 mm. The conducting paths are situated in a few layers. The different layers are separated from each other by an insulating material (siliconoxide). There are cuts through the oxide for the connection of paths in different layers. Proceeding from top to bottom we encounter one or two layers of metal (usually aluminum), a layer of poly-crystallized silicon, and one or two layers of doped silicon. The doped silicon is often called diffusion, after the way in which it is fabricated. Although in CMOS these layers are not made by diffusion, we shall adhere to that name.

In CMOS (Complementary CMOS) there are two types of diffusion, depending on the valence of the ions with which the silicon is doped. By doping silicon with ions we get a material in which either negative or positive charge carriers are abundant ("floating around"). The former material is called N-type diffusion, the latter P-type. They are both conductors. In N-type diffusion the charge carriers are electrons, in P-type holes (absent electrons). The substrate can be monolithic crystalline silicon or an insulator, such as sapphire. The latter choice has become known as CMOS/SOS (Silicon-On-Sapphire). SOS is a promising technology, since its insulating behaviour makes its physical properties very simple. Our discussion is mainly based on CMOS on an insulating substrate.

Wherever a polysilicon path crosses a diffusion path, a transistor is created: the voltage on the polysilicon path controls the flow of current through the diffusion path. These transistors are known as Field-Effect-Transistors or FETs. The polysilicon path is called the gate of the transistor. We shall call the diffusion path simply the path of the transistor. The part of the path that is underneath the gate is called the channel. Depending on the type of the path we distinguish N-type and P-type transistors. The voltage on the gate (V_g) determines the number of charge carriers in the channel. Increasing V_g attracts negative charge carriers, a decrease attracts positive charge carriers. Suppose there is a voltage difference between the two ends of the path. (Otherwise no current will flow through the path.) Phrased differently, suppose there are more charge carriers at one end of the path than at the other. We call the end with the excess of charge carriers the source and the other end the drain. (Notice that the distinction between source and drain is a dynamic one.) A transistor is on, i.e. its path is conveying charge carriers, when its gate attracts sufficiently many charge carriers from the source into the channel from where they flow to the drain. It is thus the voltage difference between V_g and V_s (source voltage) that determines whether a transistor is on. There is a threshold voltage V_t that determines the value of $V_g - V_s$ for which the transistor switches between on and off.

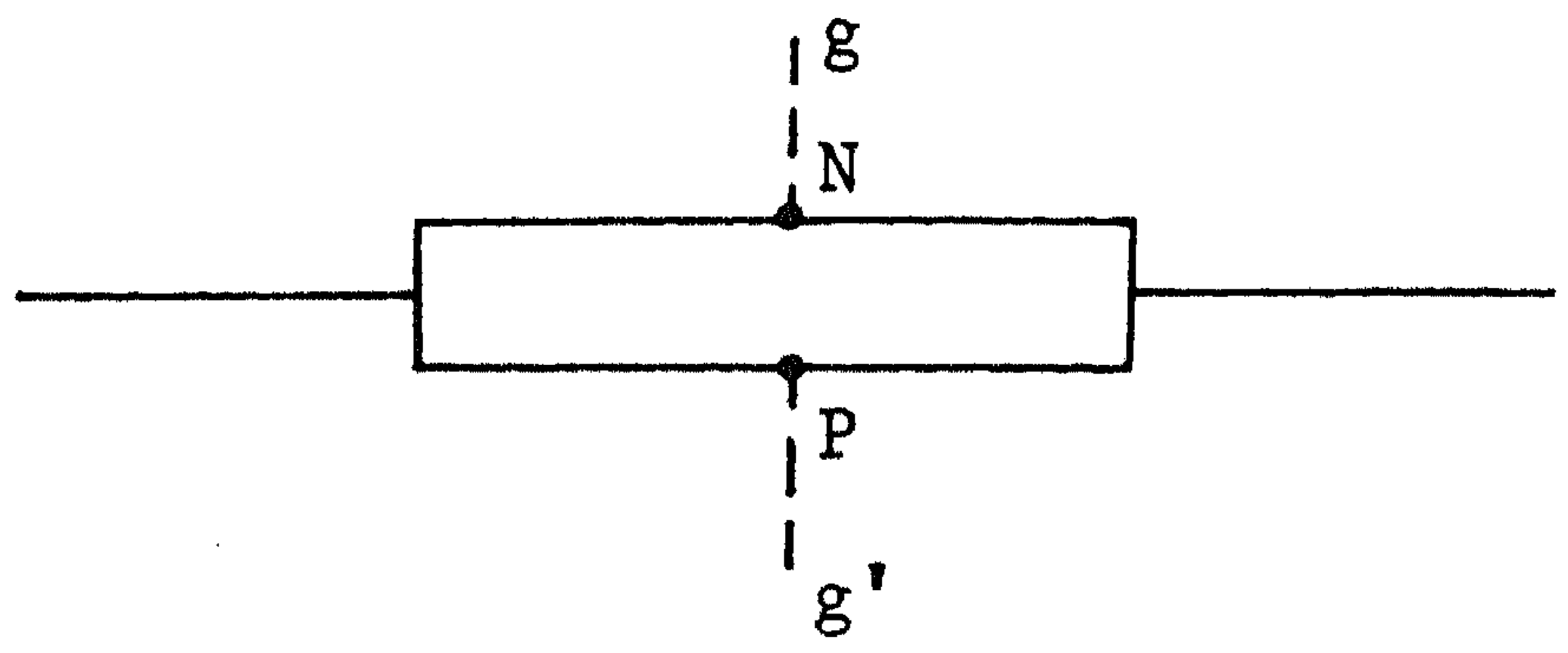
In an N-type transistor the charge carriers are electrons. Electrons are attracted by a high value of $V_g - V_s$ and repelled by a low value. The opposite is true for a P-type transistor:

		on when	off when
(3.1)	N-type	$V_g - V_s > V_t$	$V_g - V_s < V_t$
	P-type	$V_g - V_s < V_t$	$V_g - V_s > V_t$

By doping the channel with N-type or P-type impurities we can influence the V_t of the transistor. This technique is known as ion implantation. We call a transistor an enhancement transistor when it is "normally off", i.e. when it is off for $V_g = V_s$. Otherwise it is called a depletion transistor. In CMOS we do not have depletion transistors. According to (3.1), we could also have defined an enhancement transistor as a transistor of which the threshold voltage and the charge of the charge carriers have opposite signs.

Let 1 denote the positive voltage of the power supply, usually 3 to 5 V, and 0 the ground voltage. Now consider an N-type transistor. It is on for voltages V_s satisfying $0 \leq V_s < V_g - V_t$. For $V_s > V_g - V_t$ it is off and the drain voltage V_d also satisfies $V_d > V_g - V_t$, since otherwise the drain would be the source. For V_g maximal, i.e. $V_g = 1$, we find that an N-type transistor conveys only values V_s from source to drain that satisfy $0 \leq V_s < 1 - V_t$. It is a good conveyor for zeroes, but it corrupts ones. If such a corrupted one, say $1 - V_t$, is applied to the gate of a next transistor that one will convey only values V_s satisfying $0 \leq V_s < 1 - 2V_t$. An N-type transistor is a good switch only for zeroes, and that is the reason why we need a second type of transistor: the P-type transistor. In an analogous fashion we find that the P-type transistor is a good switch for ones but that it corrupts zeroes.

When designing VLSI circuits we want to abstract from the imperfection of the transistors. We want to compose VLSI circuits out of ideal switches and connections between them. This we can do in the following way. If a switch has to convey only ones or only zeroes we choose the appropriate transistor. If a switch has to convey both ones and zeroes we construct the switch out of an N-type and a P-type transistor



We use this combination only with complementary values, 0 and 1, on their gates. For g = 0 (and, hence, g'=1) both transistors are off. For g = 1 they are both on. If in the latter case a 0 is applied at one of the two sides the N-type transistor will convey it: the 0 is not corrupted. Similarly, a value 1 is conveyed perfectly by the P-type transistor.

In Section 3.2 we give some examples of VLSI circuits expressed in terms of switches and connections.

Another well-known MOS technology is NMOS. This is the technology discussed in the excellent introduction to VLSI MEAD & CONWAY 80. NMOS has only N-type transistors: enhancement and depletion transistors. The depletion transistors are used as resistors. Since a resistor is always somewhat on, NMOS has a higher power consumption and dissipation than CMOS. The asymmetry between the two types of transistors in NMOS has a number of annoying consequences, such as the need for pre-charging and ratio logic. In CMOS the only asymmetry between the types of transistors lies in the relative speeds of the charge carriers: the speed of an electron is between two and three times that of a hole. If the threshold voltage V_t is chosen too small we get a phenomenon known as subthreshold leakage. This is more serious in NMOS than in CMOS, since the way in which transistors are used in CMOS allows V_t to be chosen larger than would be desirable for NMOS.

Over the years integrated circuits have become smaller in size. It is interesting to observe how a circuit's behaviour is affected when all its dimensions are scaled down. In integrated circuits time is usually measured in multiples of τ, the transition time of transistor. It satisfies

$$\tau = \frac{L^2}{\mu(V_d - V_s)}$$

in which L is the distance from source to drain (the channel length) and μ the mobility of the charge carriers in the channel. We reduce the spatial dimensions, including those vertical to the substrate, by multiplying them by a factor α $(0<\alpha<1)$. Thus $L' = \alpha L$. In order to keep the electric field in the channel constant, we multiply the voltage by α as well. This results in $\tau' = \alpha\tau$: the transit time is reduced by the same factor. Another consequence of the fact that we scale the voltage down is that the power dissipation per unit area remains the same.

The time required for a signal to travel through a path from one transistor to another is proportional to the product of the resistance and the capacitance of the path. The resistance R of a path is proportional to its length and inversely proportional to its cross section. Hence $R' = R/\alpha$. The capacitance of a path is inversely proportional to the distance to its neighbouring paths and layers, and it is proportional to the area facing that neighbouring path or layer. Hence, $C' = \alpha C$, and, consequently, $R'C' = RC$. The time required for a signal to go from one transistor to another measured in seconds is, consequently, not affected. But since $\tau' = \alpha\tau$, the time measured in multiples of τ has changed. In τ-relative terms, if it took a signal time t to go from one transistor to another then after scaling it will take time t/α. Thus the scaled circuit may not function anymore. The delays in its paths may have become too long.

Matters are even worse if we look at the time required for a signal to go a fixed distance, say from one end of the chip to the other. If in τ-relative terms it first took time t, then after scaling it will take time t/α^2. This clearly demonstrates, firstly why propagation delays cannot be ignored in VLSI, and secondly that the distribution of a clock signal over the entire chip is not viable in VLSI, since such a cross-chip propagation has a quadratic scaling factor. The reader will now appreciate why we are looking for delay-insensitive circuits.

3.2. Restoring logic circuitry

We have demonstrated how perfect switches can be realized in CMOS. We now use these switches to construct restoring logic circuitry. A restoring logic circuit is a circuit in which the outputs are permanently driven by the power supply. The inputs determine, by controlling the switch settings, which outputs are connected to the high voltage of the power supply, denoted by 1, and which ones to the ground voltage, denoted by 0. We do, consequently, not rely on the fact that values may be stored temporarily on disconnected paths. (Due to the subthreshold leakage, values on disconnected paths deteriorate with time.) Restoring logic seems the natural choice for circuits that are realized in a submicron technology i.e. with path widths smaller than 10^{-6} m.

A *logic component* is a graph. Its vertex set is the union of the disjoint sets X, Y, Z, and $\{0,1\}$. The elements of X and Y are called *input ports* and *output ports*, respectively. The elements of Z are called *interior nodes*. Each edge of the graph is either labeled or unlabeled. (Labeled edges represent switches.) There are two types of labels: i and i', with $i \in X$. A label i represents a switch that is on for $i = 1$, i' a switch that is on for $i = 0$. Logic components will often simply be called components. (In hierarchically composed components, to be discussed below, we allow labels that are chosen from a larger set than just X.)

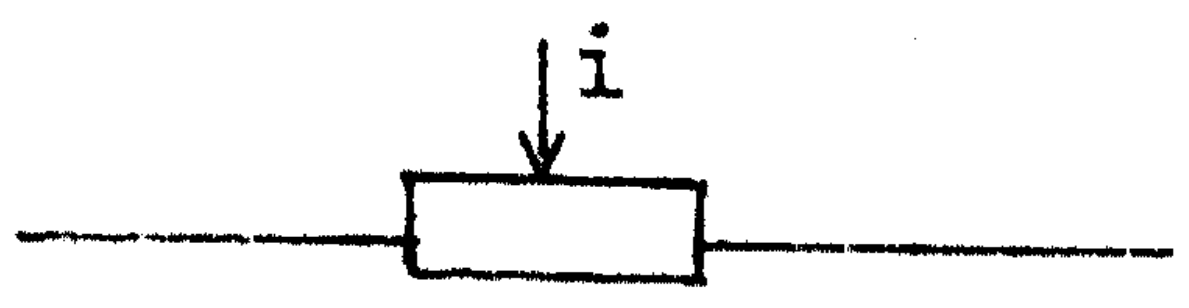
P(X) denotes the power set of X. Vertices j and k are called *separated* if for every $A \in P(X)$ every path between j and k has an edge labeled i with $i \notin A$ or an edge labeled i' with $i \in A$. Let $V \subset X \cup \{0,1\}$. Vertex j is called *driven by* V if for every $A \in P(X)$ there exists a $v \in V$ and a path between j and v for which $i \in A$ for every edge labeled i and $i \notin A$ for every edge labeled i'. A logic component is called *nonfighting* if every two distinct vertices in $X \cup \{0,1\}$ are separated. A logic component is called *well-behaved* if it is nonfighting and every vertex $j \in Y$ is driven by $X \cup \{0,1\}$. A logic component is called *restoring* if it is nonfighting and every vertex $j \in Y$ is driven by $\{0,1\}$. Notice that a restoring component is well-behaved.

We introduce a notation for the description of logic ocmponents that is very similar to the one we introduced for partially ordered computations. The difference is that the alphabet is replaced by a list of ports and the command by an enumeration of the edges of the graph. The following is an example of a description of a component.

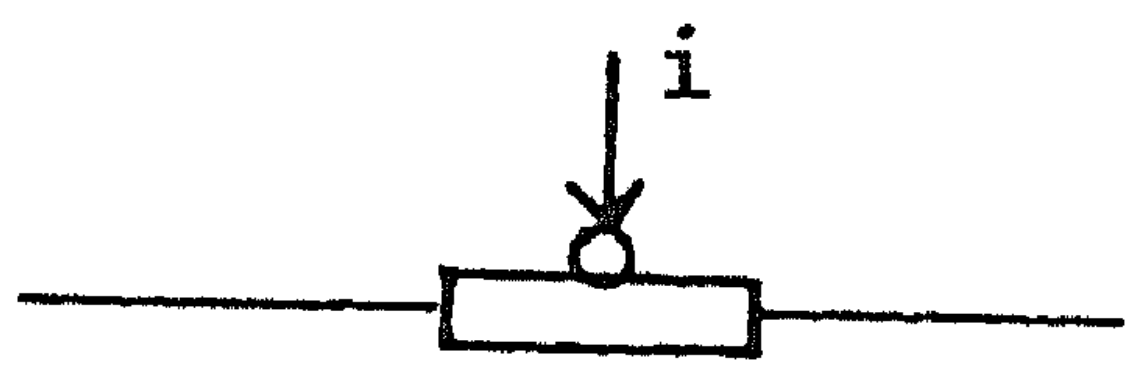
```
com inverter(in?,out!):
    in' → out = 1
    in → out = 0
moc
```

In the list of ports each input port is postfixed by a question mark and each output port by an exclamation point. The graph of this component has two edges: an edge labeled in', connecting out and 1, and an edge labeled in, connecting out and 0. It has no interior nodes. It is an example of a restoring logic component.

For every logic component we can draw a diagram. This is essentially a picture of the graph extended with connections between input ports and the switches they control. An edge labeled i is drawn as



and an edge labeled i' as



They represent the switches. Their gates are connected to input port i. Arrows with the same label may be drawn connected in the diagram. Fig. 1 shows a diagram of the inverter.

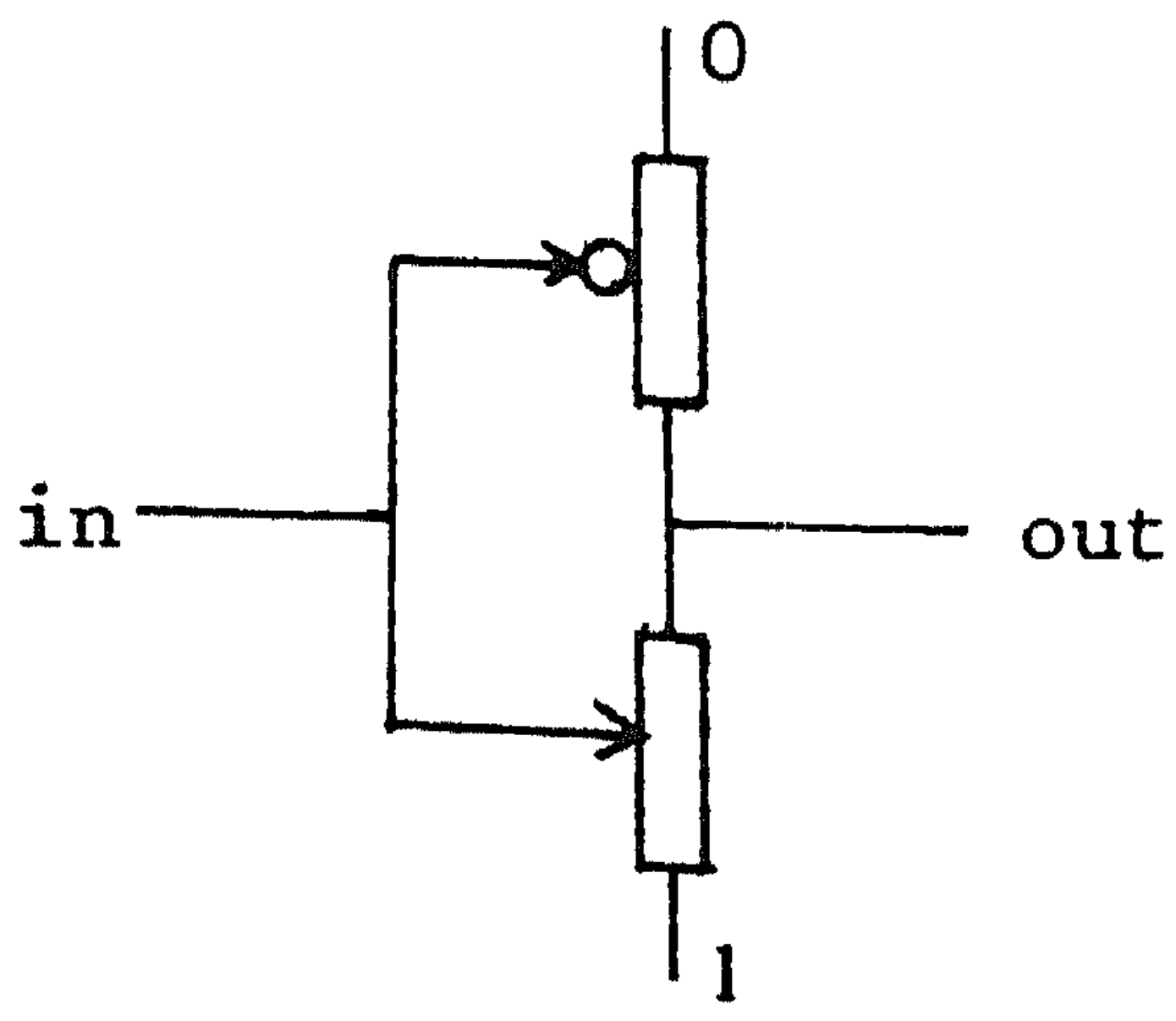


Fig. 1 Diagram of an inverter

When specifying labeled edges we allow more general Boolean expressions to the left of the arrow than just the labels i and i' ($i \in X$). Such labels may be connected by the Boolean operators $\wedge$ and $\vee$. The formula "$b_0 \vee b_1 \rightarrow x = y$" specifies two connections between x and y: $b_0 \rightarrow x = y$" and "$b_1 \rightarrow x = y$". This is known as parallel composition. The formula "$b_0 \wedge b_1 \rightarrow x = y$" introduces an interior node. Calling that interior node z, the formula is equivalent to "$b_0 \rightarrow x = z$" and "$b_1 \rightarrow z = y$". This is known as serial composition. Both parallel and serial composition occur in the following component.

```
com nor(a?,b?,out!):
    a' ∨ b' → out = 1
    a ∨ b → out = 0
moc
```

A diagram of this component is shown in Fig. 2.

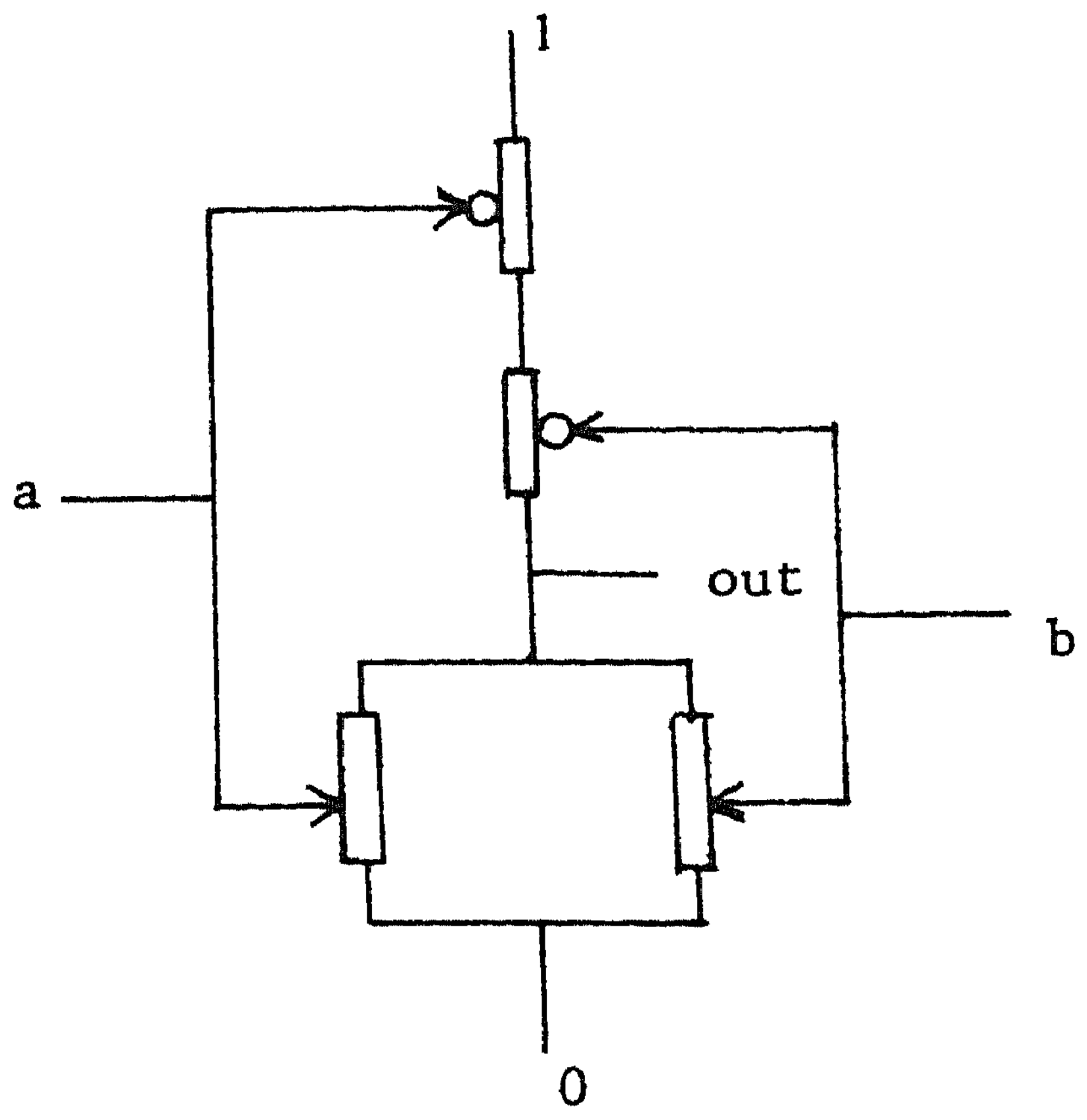


Fig. 2 Diagram of a nor-circuit

Except in very simple cases, components will be composed of subcomponents and connections between them. The latter are (possibly labeled) connections between the ports of the component, {0,1}, interior nodes, and the ports of the subcomponents. These connections thus constitute a logic component, to which we shall refer as the local graph.

A *hierarchically composed component* consists of a logic component, called the *local graph,* and zero or more hierarchically composed subcomponents. X_{loc} and Y_{loc} denote the sets of input and output ports, respectively, of the hierarchically composed component. X_{int}, called the set of *internal input ports* of the local graph, is the union of the sets of output(!) ports of the subcomponents. Y_{int}, the *internal output ports,* is the union of the sets of input(!) ports of the subcomponents. (For a component without subcomponents the local graph is, consequently, the entire component.)

A hierarchically composed component is a component. Its graph is the composition of the local graph and the graphs of the subcomponents, in which the ports of the subcomponents are the internal ports of the local graph. Let X_{tot} be defined as the union of X_{loc} of the local graph and the X_{tot}'s of the subcomponents. A hierarchically composed component has labels i and i' with $i \in X_{tot}$. In hierarchically composed components the labels are thus drawn from a richer set than just X. This requires a slight change in the definitions of separated and driven. In both cases P(X) should be replaced by $P(X_{tot})$.

The following properties show that the restrictions imposed on components can be checked locally, i.e. for each local graph separately. For a more comprehensive treatment of restoring logic the reader is referred to REM 82, which includes the proofs of the properties.

PROPERTY 3.1. A component of which the local graph and all subcomponents are nonfighting is nonfighting.

PROPERTY 3.2. A component of which the local graph and all subcomponents are well-behaved is not necessarily well-behaved.

PROPERTY 3.3. A component is restoring if all its subcomponents are restoring and every vertex $j \in Y$ is driven in the local graph by $X_{int} \cup \{0,1\}$.

The following is an example of a restoring hierarchically composed component. Like in partially ordered computations, s.p signifies port p of subcomponent s.

```
com C(a?,b?,q!,qbar!):
    sub i0,i1: inverter
    a ∧ b → i0.in = 0
    a' ∨ b' → i0.in = i1.out
    a' ∧ b' → i1.in = 0
    a ∨ b → i1.in = i0.out
    q = i0.out
    qbar = i1.out
moc
```

The local graph of component C consists of eight labeled edges and two unlabeled ones. An unlabeled edge between x and y is specified as "x = y".

Consider the four lines representing labeled edges. They may be viewed as equations in 4 unknowns together with conditions (in a and b) under which they are valid. For given a and b there remain two equations. We, furthermore, have that the inputs and outputs of i0 and i1 are each other's inverse. Thus we have, for given a and b, a system of 4 equations in 4 unknowns. Together with the last two lines this gives 6 equations in 6 unknowns. If $a \neq b$ this system has two solutions. If a system has multiple solutions there must have been an earlier moment at which it had one solution and that solution must be one of the multiple solutions. That unique solution is then chosen among the multiple ones. Thus history dependence (or storage or state) may be represented in this framework.

We interpret output q as the value stored. The output qbar is its complement. Component C is a Muller C-element with two inputs (SEITZ 80). The value stored can change only when the input values change. If the inputs are made equal the value is made equal to the inputs. Otherwise the value remains unchanged. Fig. 3 contains a diagram of the component. The C-element plays an important role in delay-insensitive circuits.

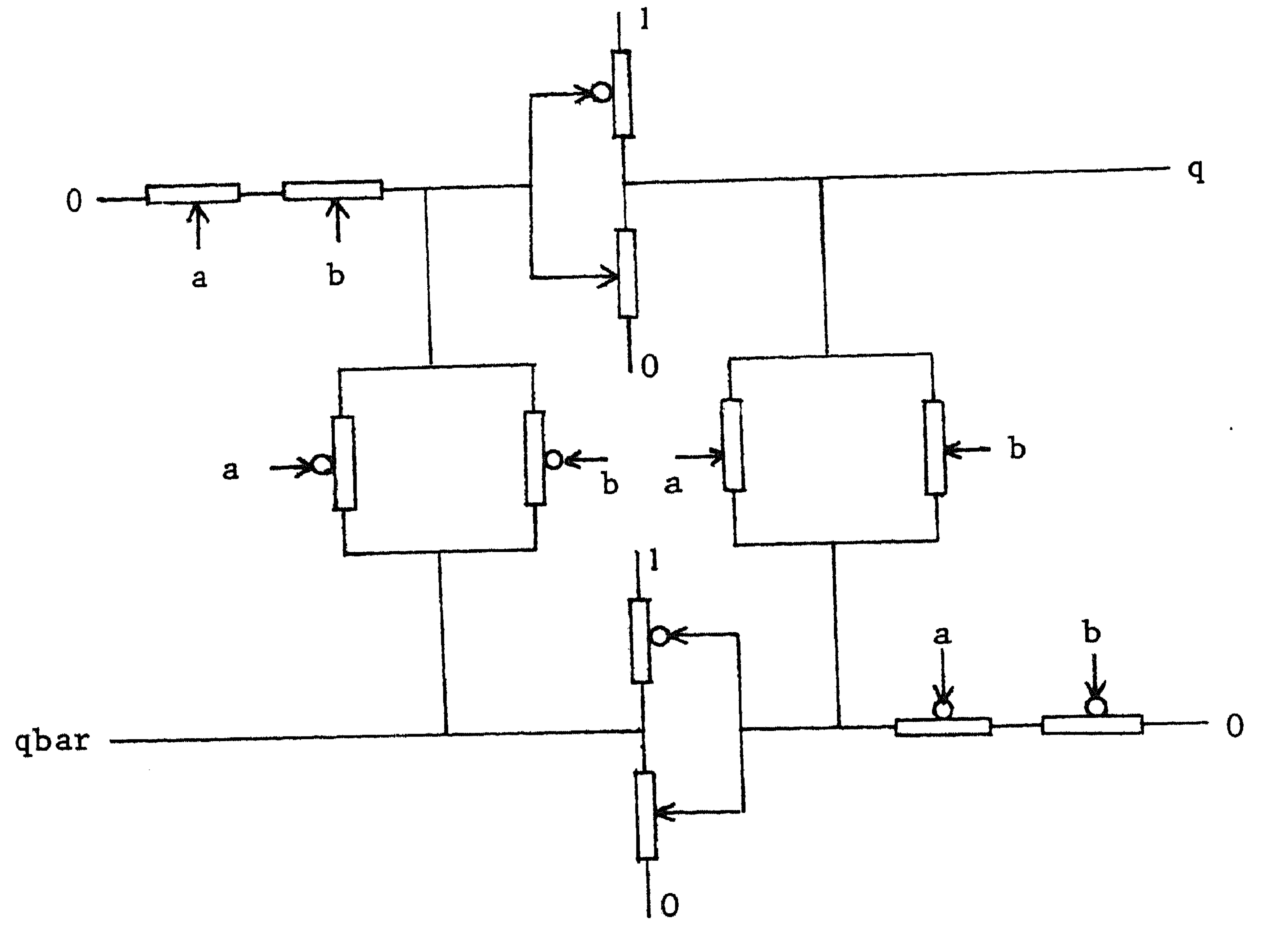


Fig. 3 Diagram of a C-element

As a last example of a restoring component we discuss a tally-circuit,

$$\underline{\text{com}}\ \text{tally}_0(\text{out}_0!);\ \text{out}_0 = 1\ \underline{\text{moc}}$$

for i > 0:

```
com $tally_i(in_0?, \; ... \; ,in_{i-1}?,out_0!, \; ... \; ,out_i!)$.
    sub t: $tally_{i-1}$
    $in_j$ = $t.in_j$ forall j in 0..i-2
    $in_{i-1} \rightarrow out_0$ = 0, $out_j$ = $t.out_{j-1}$ forall j in 1..i
    $in_{i-1}' \rightarrow out_j$ = $t.out_j$ forall j in 0..i-1, $out_i$ = 0
moc
```

Component $tally_i$ has i input ports and i + 1 output ports. The line starting with in_j specifies i − 1 edges, one for every j satisfying $0 \leq j \leq i - 2$. The next line shows that we allow more than one edge to the right of an arrow. It specifies i + 1 connections, each of them labeled in_{i-1}. Component $tally_i$ counts the number of inputs that have value 1. If j $(0 \leq j \leq i)$ inputs are 1 output out_j will be 1 and the other outputs 0. Figure 4 shows a diagram of $tally_3$. This is basically the same circuit as the tally in MEAD & CONWAY 80.

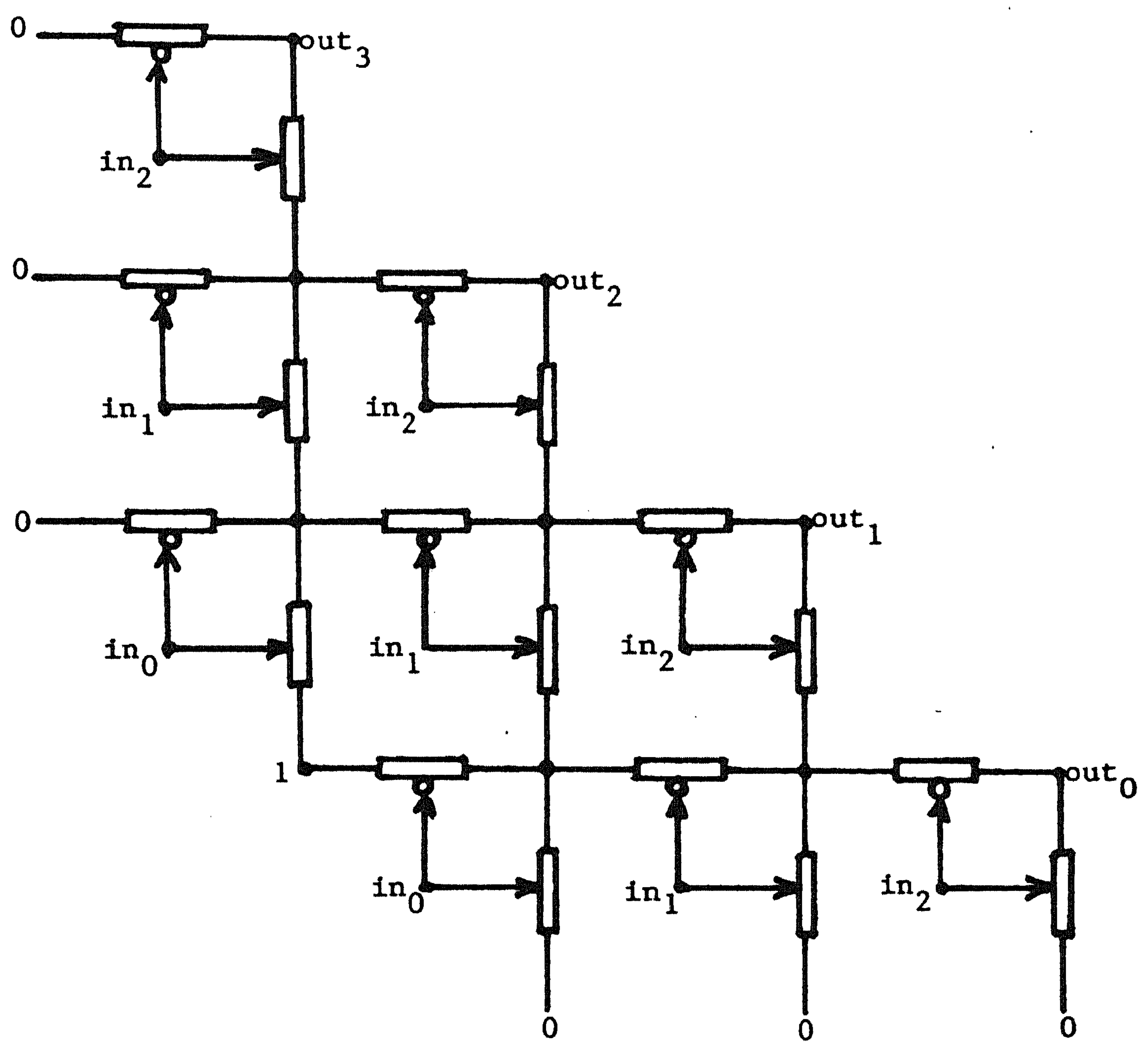


Fig. 4 Diagram of $tally_3$-circuit

4. DELAY-INSENSITIVE SIGNALING

In Section 2 we have become acquainted with the kind of computations we want to consider. Section 3 has given us a clear view of the kind of logic components we can build. But there is still an important gap left between them. This gap has mainly to do with what is customarily called "timing". If a component computes a result, and if that result is again input for another component, how then can we guarantee that the outputs are not used by the second component before they have assumed their values? The traditional method is to provide the components with clock signals that are generated at regular time intervals and that signal the completion of the preceding step. Since this requires estimating the connection delays, this approach is unsuitable for VLSI. In this section we discuss a different technique known as delay-insensitive signaling or self-timed signaling (SEITZ 80).

4.1. A composition operator expressing delay

We introduce a third composition operator: r-composition. It is similar to q-composition, but it expresses unbounded delay. There is a direction in delay: the sending of a signal will always precede its reception. To express this we introduce directed trace structures. (When appropriate, we refer to the trace structures defined in Section 2 as undirected trace structures.)

A directed trace structure is a triple $S = <T,A_0,A_1>$. A_0 and A_1 are disjoint finite subsets of Σ. $T \subset (A_0 \cup A_1)^*$. A_0 is called the *output alphabet* of S, and A_1 its *input alphabet*. The elements of A_0 and A_1 are called its *output symbols* and *input symbols*, respectively. When we say the alphabet of S, we mean $A_0 \cup A_1$.

We compose directed trace structures $S = <T,A_0,A_1>$ and $S' = <U,B_0,B_1>$ only when their alphabets satisfy $A_0 \cap B_0 = \emptyset$ and $A_1 \cap B_1 = \emptyset$. The p- and q-compositions of S and S' are p- and q-compositions of $<T,A_0 \cup A_1>$ and $<U,B_0 \cup B_1>$.

Any symbol a occurring in the intersection of the alphabets of S and S' is an output symbol in one and an input symbol in the other. To distinguish between a as an input symbol and a as an output symbol we introduce postfixed symbols. A symbol a can be postfixed with "!" or "?", yielding a! and a?, respectively. These are two distinct symbols.

Let A_0 and A_1 be disjoint sets of symbols. If B is an alphabet then $B/(A_0,A_1)$ is the alphabet obtained from B by replacing each symbol a in A_0 by the postfixed symbol a! and each symbol a in A_1 by a?. Trace set $T/(A_0,A_1)$ is similarly obtained from trace set T. Let $S = \langle T,A \rangle$ be an undirected trace structure. The undirected trace structure $S/(A_0,A_1)$ is then given by

$$S/(A_0,A_1) = \langle T/(A_0,A_1) \ , \ A/(A_0,A_1) \rangle \ .$$

We introduce a special undirected trace structure DEL. Let a and b be two distinct symbols. DEL(a,b) is the trace structure with $\{a,b\}$ as its alphabet and

$$\{t \in \{a,b\}^* \mid \#_a t' \geq \#_b t' \text{ for every prefix } t' \text{ of } t\}$$

as its trace set.

We now come to the definition of r-composition. Let $S = \langle T,A_0,A_1 \rangle$ and $S' = \langle U,B_0,B_1 \rangle$ be two directed trace structures. Consider the following undirected trace structure.

$$(4.1) \qquad \begin{array}{l} \langle T,A_0 \cup A_1 \rangle/(A_0 \cap B_1,A_1 \cap B_0) \ \underline{q} \\ \langle U,B_0 \cup B_1 \rangle/(B_0 \cap A_1,B_1 \cap A_0) \ \underline{q} \\ \mathrm{DEL}(a_0!,a_0?) \ \underline{q} \ \dots \ \underline{q} \ \mathrm{DEL}(a_{n-1}!,a_{n-1}?) \end{array}$$

in which $\{a_0, \dots ,a_{n-1}\} = (A_0 \cap B_1) \cup (A_1 \cap B_0)$, i.e. the intersection of the alphabets of S and S'.

The *r-composition* of S and S', denoted S $\underline{r}$ s', is trace structure (4.1) with its alphabet partitioned into the output alphabet $(A_0 \backslash B_1) \cup (B_0 \backslash A_1)$ and the input alphabet $(A_1 \backslash B_0) \cup (B_1 \backslash A_0)$.

The trace structures $\mathrm{DEL}(a_i!,a_i?)$ express the delay between the sending of a_i and the reception of a_i.

EXAMPLE 4.1. Consider the program

(4.2) (z!; a; z!; b), (p; z?; q; z?)

(we have postfixed the symbols in the intersection of the alphabets to show their types.) With the comma denoting q-composition (4.2) is equivalent to

(4.3) p; a,q;b

With r-composition (4.2) is equivalent to

(4.4) (a;b), (p;q)

Program (4.3) has two traces and (4.4) six, including the two of (4.2).

There are two problems associated with the delay as expressed by r-composition. The first one is that the trace set of DEL is not regular. DEL is the unbounded SYNC or, more precisely

$$\mathrm{DEL}(a,b) = \bigcup_{k=1}^{\infty} \mathrm{SYNC}_k(a,b)$$

Trace structure DEL(a,b) has as its states $\{[a^i] \mid i \geq 0\}$, which is an infinite set. By introducing unbounded delay we have, consequently, left the realm of the finite-state machines and, hence, that of the regular sets.

The second problem is that one might question the validity of the assumption that a connection has an unbounded buffering capacity. A wire, obviously, does not have this property. On the other hand, wires do exhibit delay. So we cannot simply dispense with r-composition.

Fortunately, matters are not as dim as they may look. We want to restrict our components in such a way that their r-composition equals their q-composition, thus restricting ourselves again to regular sets. This is reflected in the following definition. A composition of a collection of trace structures is called *delay-insensitive* if the collection's q-composition and r-composition yield the same trace sets. For delay-insensitive compositions the trace structures $\mathrm{DEL}(a_i!,a?)$ in (4.1) may, without affecting the composite, be replaced by $\mathrm{SYNC}_1(a_i!,a_i?)$. Hence, the connections need only accomodate buffering capacity 1, which seems a very reasonable assumption for wires.

It is possible to transform trace structures in such a way that their composition is delay-insentive without affecting their composite. We have actually already demonstrated this technique when we constructed component quinsem in Section 2.2. Compare this component with quinsem'. In the latter one the alphabets of the subcomponents are not disjoint: b0.p and b1.v are the same symbol. In quinsem this is implemented by adding the "acknowledge signal" q. The following property shows that this is a general technique.

Let $S = \langle T, A_0, A_1 \rangle$ be a directed trace structure and a a symbol with $a \in A_0 \cup A_1$. Then

$$S[a := a_0 a_1]$$

denotes the trace structure obtained from S by

(i) replacing in every trace of T the symbol a by the sequence $a_0 a_1$;
(ii) replacing a by a_0 in the alphabet that contains a;
(iii) adding a_1 to the alphabet that does not contain a.

(It is assumed that a_0 and a_1 are fresh symbols.)

$$S[\forall a \in C\colon a := a_0 a_1]$$

denotes the trace structure obtained from S by applying the transformations above for all symbols $a \in C$.

<u>PROPERTY 4.1.</u> Let $S = \langle T, A_0, A_1 \rangle$ and $S' = \langle U, B_0, B_1 \rangle$ be two directed trace structures. $A = A_0 \cup A_1$ and $B = B_0 \cup B_1$. Then

$$S \ \underline{q}\ S' =$$
$$S[\forall a \in A \cap B\colon a := a_0 a_1] \ \underline{q}\ S'[\forall a \in A \cap B\colon a := a_0 a_1] =$$
$$A[\forall a \in A \cap B\colon a := a_0 a_1] \ \underline{r}\ S'[\forall a \in A \cap B\colon a := a_0 a_1].$$

Thus we have found a way of making our compositions delay-insensitive. Applying this to Example 4.1 would give rise to the program

$$(z_0!;\ z_1?;\ a;\ z_0!;\ z_1?;\ b),\ (p;\ z_0?;\ z_1!;\ q;\ z_0?;\ z_1!)$$

which, both under q- and under r-composition, is equivalent to (4.3).

<u>4.2. Two delay-insensitive circuits</u>

In Section 3 we discussed logic components composed of switches and connections. The connections carry the values 0 and 1. We looked at how the output values depend on the input values, but we hardly discussed dynamic behaviour, i.e. transitions on the connections. A transition is a change in value. Let a be a point on a connection. A change from 0 to 1 in point a (called a high-going transition) is denoted by $a\uparrow$, and a change from 1 to 0 (a low-going transition) by $a\downarrow$. Consider the program

(4.5) $(v;\ t!)^*,\ (t?;\ w)^*.$

Under q-composition (4.5) yields $v;\ (w,v)^*$, i.e. $\mathrm{SYNC}_2(v,w)$, under r-composition, however, (4.5) yields DEL(v,w). Property 4.1 tells us how to resolve this difference: we change (4.5) into

(4.6) $(v;\ t!;\ t_1?)^*,\ (t_0?;\ t_1!;\ w)^*.$

But let there now be two connections, one for t_0 and one for t_1, and let the symbols t_i stand for transitions on these connections. Assume the connections to be low (carrying the value 0) initially. Since high-going and low-going transitions on the same connection alternate, (4.6) becomes

$$(v;\ t_0\uparrow!;\ t_1\uparrow?;\ v;\ t_0\downarrow;\ t_1\downarrow?)^*,\ (t_0\uparrow?;\ t_1\uparrow!;\ w;\ t_0\downarrow?;\ t_1\downarrow!;\ w)^*$$

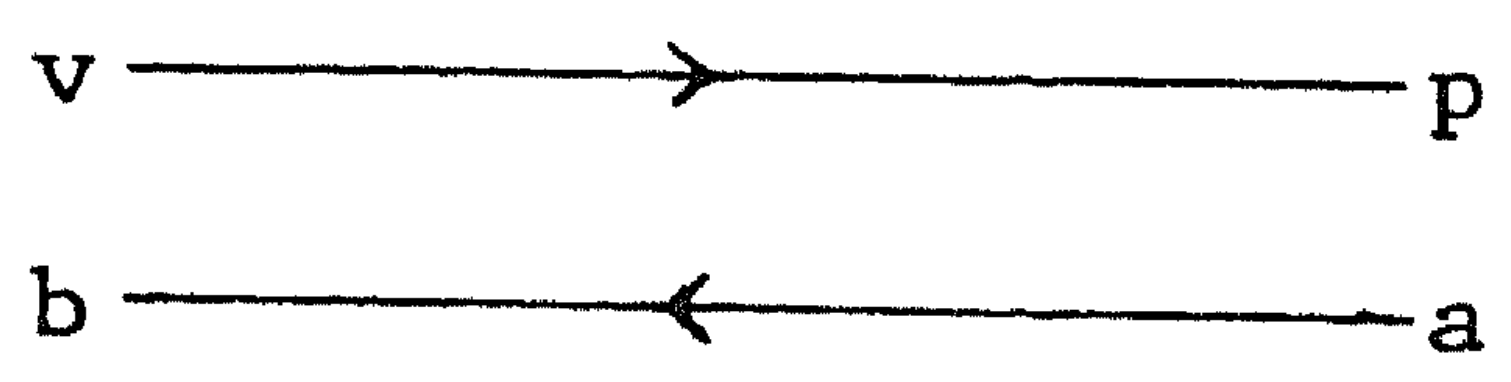
$t_i\uparrow!$ may be read as "drive connection t_i high" and $t_1\uparrow?$ as "observe connection t_i going high". This type of signaling is known as 2-cycle signaling: there are two transitions between successive v's or w's. A more customary way of delay-insensitive signaling is 4-cycle signaling;

$$(v;\ t_0\uparrow!;\ t_1\uparrow?;\ t_0\downarrow!;\ t_1\downarrow?)^*,\ (t_0\uparrow?;\ t_1\uparrow!;\ t_0\downarrow?;\ t_1\downarrow!;\ w)^*.$$

We have now four transitions between successive v's or w's. The 4-cycle scheme is, as we saw, not necessary to achieve delay-insensitivity, but it tends to make the circuits thus communicating simpler than with 2-cycle signaling. The reason for this is that every v (and w) takes place with the pair (t_0,t_1) of connections having the same value (cf. SEITZ 80).

In this section we design two components: a quick return linkage and a binary semaphore. As computations they are not very interesting, but they provide a good insight in the application of trace theory to delay-insensitive signaling. We shall employ 4-cycle signaling. As a consequence, we may use q-composition as our composition operator. We henceforth omit the question marks and exclamation points.

Consider again two connections with 4-cycle signaling. We give names to their endpoints



The 4-cycle signaling is expressed by the order

$$(v\uparrow;\ p\uparrow;\ a\uparrow;\ b\uparrow;\ v\downarrow;\ p\downarrow;\ a\downarrow;\ b\downarrow)^{*}. \tag{4.7}$$

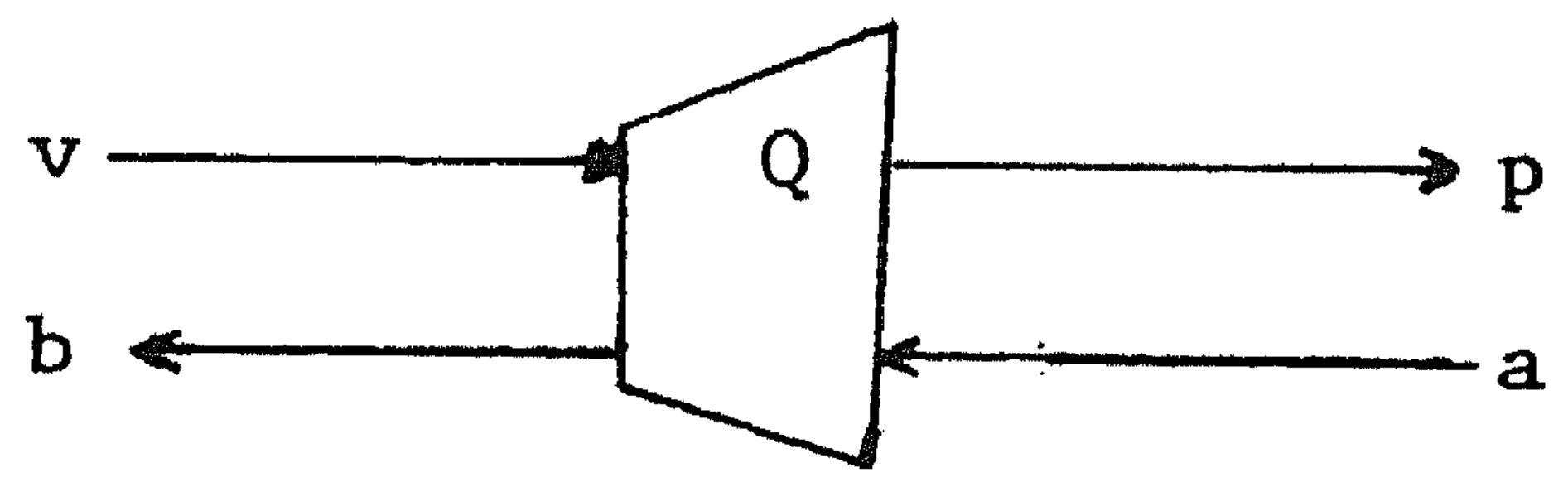
Expression (4.7) may be rewritten as

$$v\uparrow;\ (p\uparrow;\ a\uparrow;\ b\uparrow;\ v\downarrow;\ p\downarrow;\ a\downarrow;\ b\downarrow;\ v\uparrow)^{*}. \tag{4.8}$$

We split in (4.8) the transitions at the left end of the connections from those at the right end, which yields - adding symbols to make it equivalent to (4.8)

$$\begin{aligned} &v\uparrow;\ (t0;\ p\uparrow;\ a\uparrow;\ t1;\ t2;\ p\downarrow;\ a\downarrow;\ t3)^{*},\\ &\quad (t0;\ t1;\ b\uparrow;\ v\downarrow;\ t2;\ t3;\ b\downarrow;\ v\uparrow)^{*}. \end{aligned} \tag{4.9}$$

We can weaken the order expressed in (4.9) by inserting in the connections a so-called quick return linkage



(By drawing arrowheads in the connections we express whether a transition $a\uparrow$ stands for $a\uparrow!$ or for $a\uparrow?$.) The effect of the insertion of the quick return linkage is that $p\downarrow;\ a\downarrow$ does not have to precede $b\downarrow;\ v\uparrow$, i.e. that in expression (4.9) t3 is deleted:

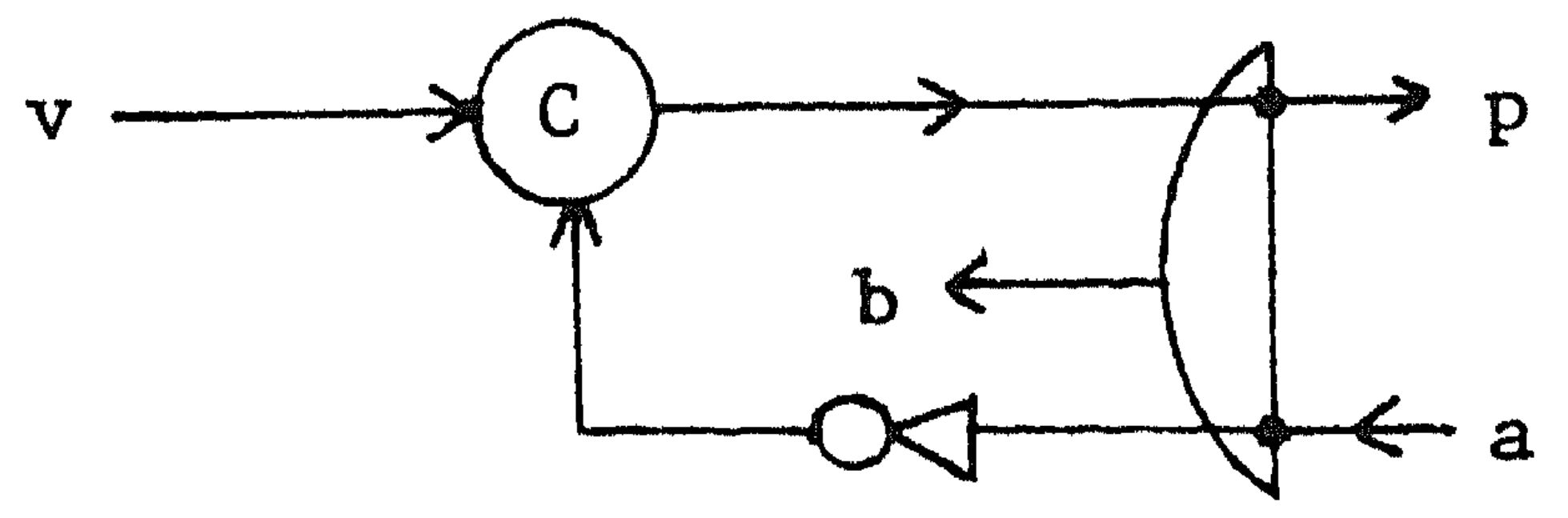
$$\begin{aligned} &v\uparrow;\ (t0;\ p\uparrow;\ a\uparrow;\ t1;\ t2;\ p\downarrow;\ a\downarrow)^{*},\\ &\quad (t0;\ t1;\ b\uparrow;\ v\downarrow;\ t2;\ b\downarrow;\ v\uparrow)^{*}. \end{aligned} \tag{4.10}$$

Expression (4.10) is equivalent to

$$v\uparrow;\ (p\uparrow;\ a\uparrow;\ b\uparrow;\ v\downarrow;\ (p\downarrow;\ a\downarrow),\ (b\downarrow;\ v\uparrow))^{*}. \tag{4.11}$$

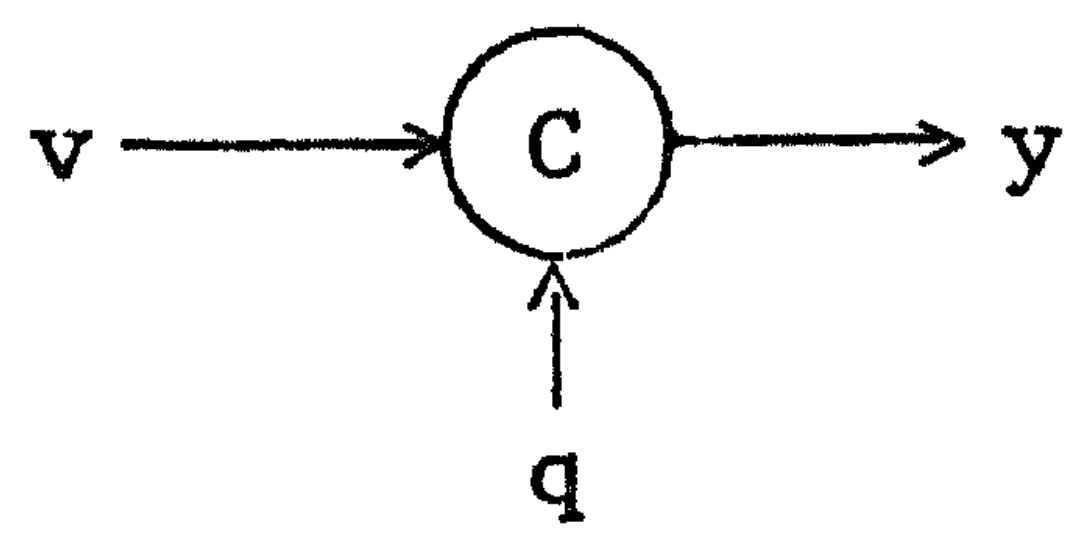
We show that Q can be implemented by

(4.12)



A circuit

(4.13)



denotes a C-element. It implements, as we saw in Section 3.2, the order

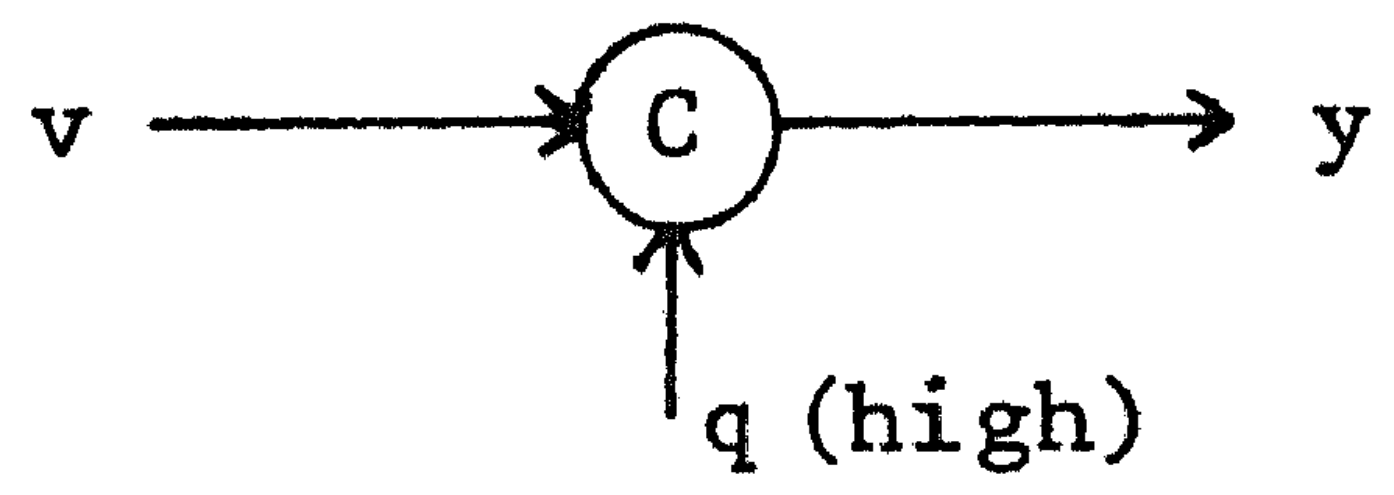
(4.14) $(v\uparrow,\ q\uparrow;\ y\uparrow;\ v\downarrow,q\downarrow;\ y\downarrow)^*.$

Expression (4.14) may be rewritten as

(4.15) $v\uparrow,q\uparrow;\ (y\uparrow;\ v\downarrow,q\downarrow;\ y\downarrow;\ v\uparrow,q\uparrow)^*.$

Consider the case that q is initially high:

(4.16)

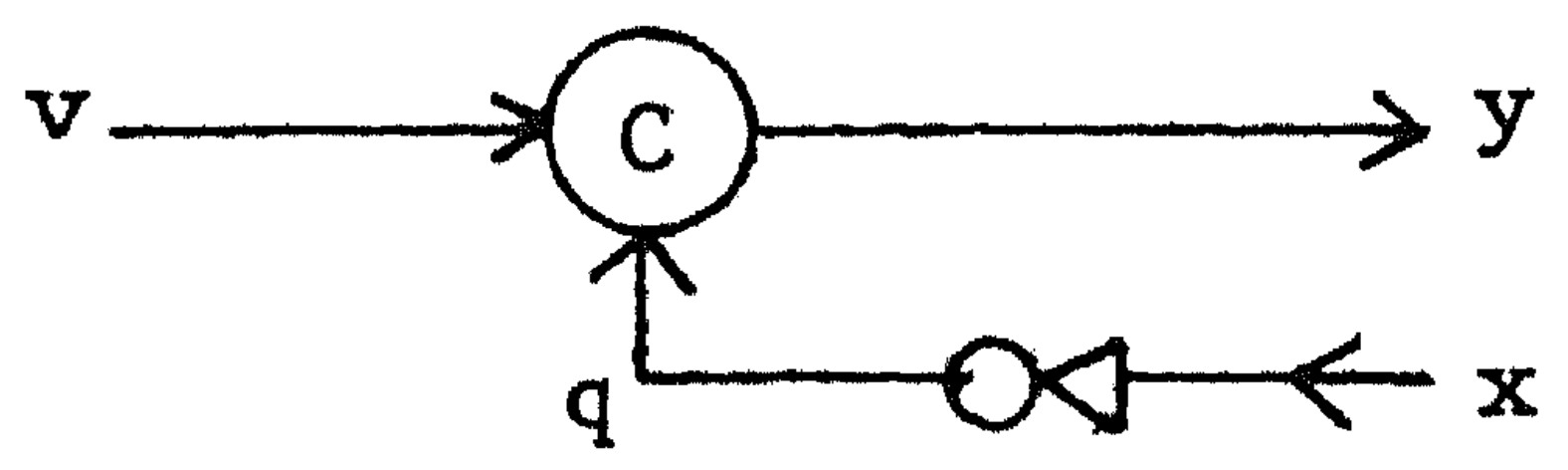


which corresponds to dropping the initial $q\uparrow$:

(4.17) $v\uparrow;\ (y\uparrow;\ v\downarrow,q\downarrow;\ y\downarrow;\ v\uparrow,q\uparrow)^*.$

Next, let circuit (4.16) be extended with an inverter at q:

(4.18)

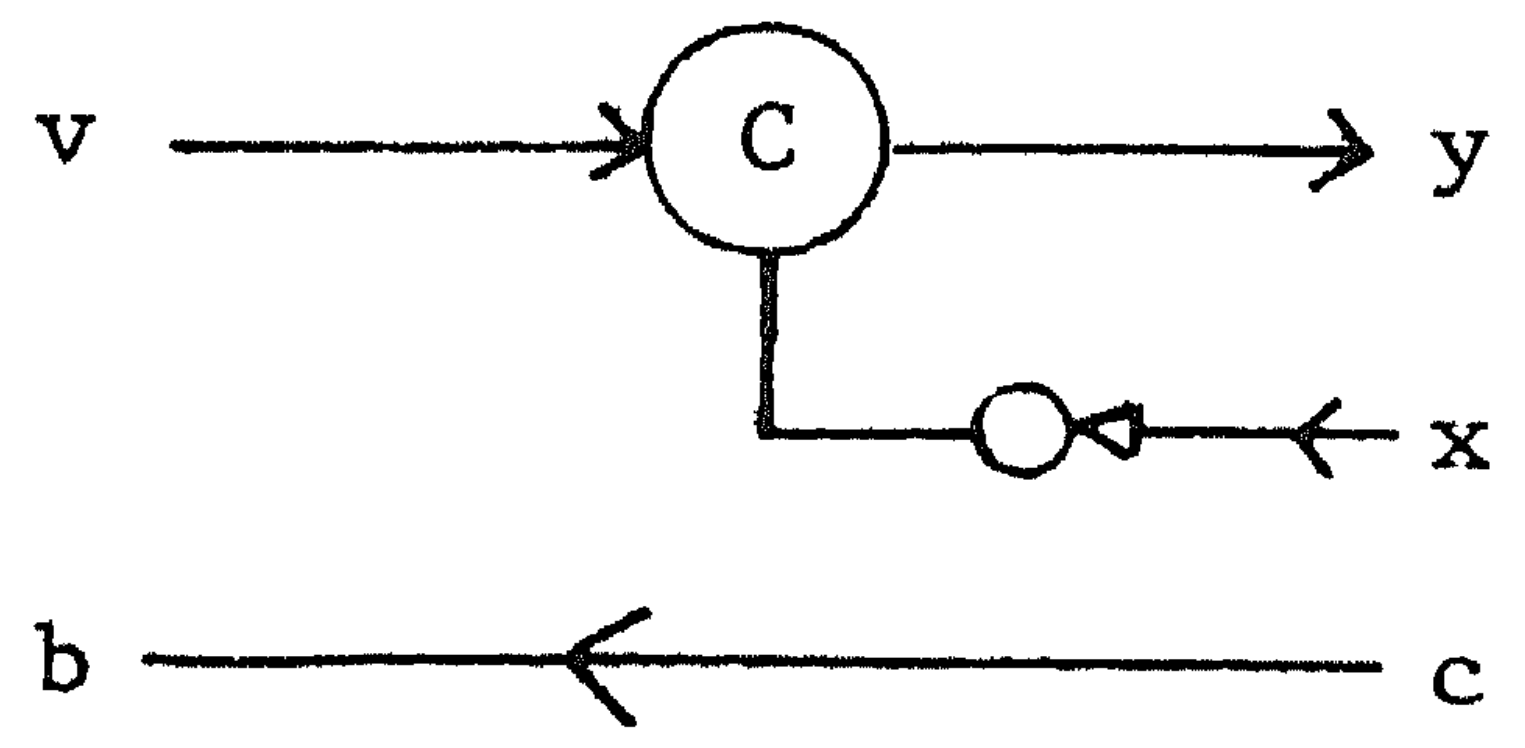


The expression for circuit (4.18) is obtained from (4.17) by replacing $q\downarrow$ and $q\uparrow$ by $x\uparrow$ and $x\downarrow$, respectively:

(4.19) $v\uparrow;\ (y\uparrow;\ v\downarrow, x\uparrow;\ y\downarrow;\ v\uparrow, x\downarrow)^*.$

We extend circuit (4.18) with a wire

(4.20)



The order for the wire is

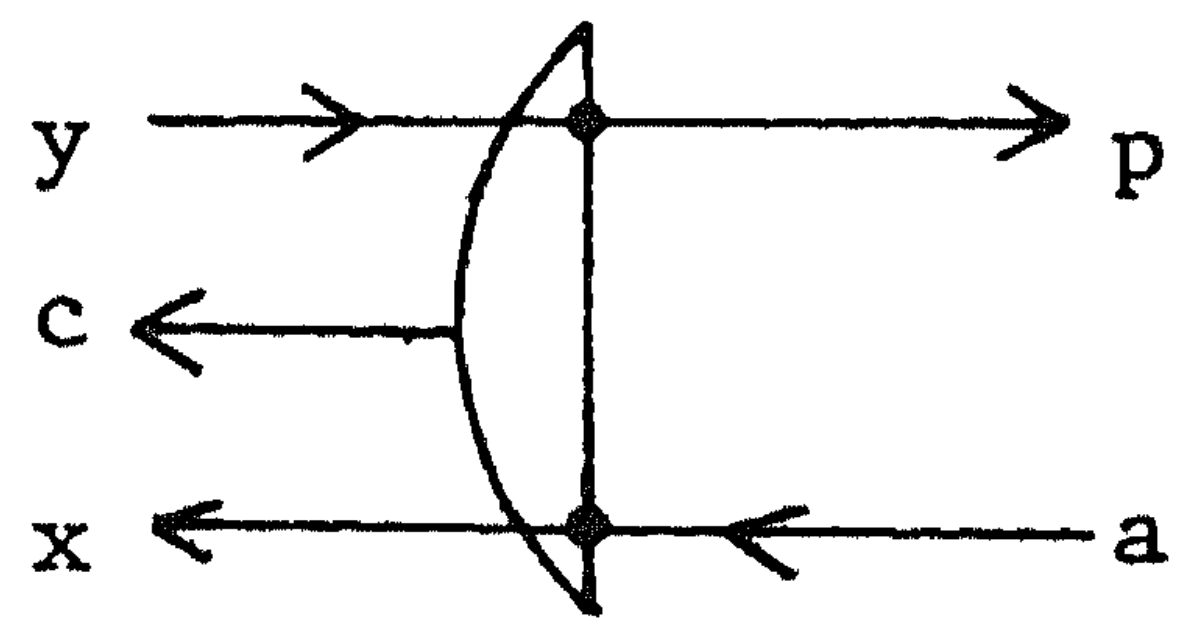
(4.21) $(c\uparrow;\ b\uparrow;\ c\downarrow;\ b\downarrow)^*.$

Circuit (4.20) has a left and a right environment. With both environments 4-cycle signaling is employed. This yields the following combination of (4.19) and (4.21) as the order for circuit (4.20):

(4.22) $v\uparrow;\ (y\uparrow;\ (c\uparrow;\ b\uparrow;\ v\downarrow),\ x\uparrow;\ y\downarrow;\ (c\downarrow;\ b\downarrow;\ v\uparrow), x\uparrow)^*.$

A circuit

(4.23)

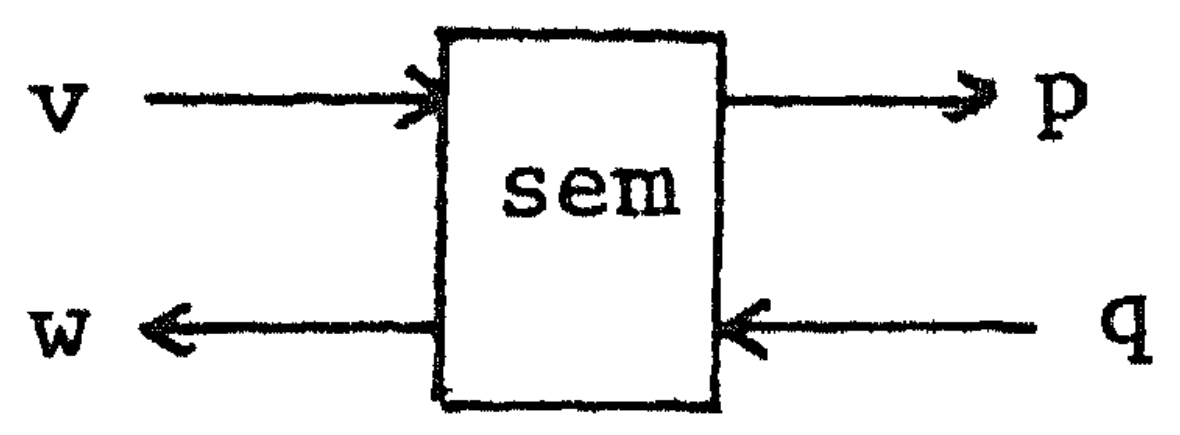


denotes an and-element with input reproduction. It has a left and a right environment, using 4-cycle signaling with both environments: the order for the left environment is $(y\uparrow;\ c\uparrow, x\uparrow;\ y\downarrow;\ c\downarrow, x\downarrow)^*$ and for the right environment $(p\uparrow;\ a\uparrow;\ p\downarrow;\ a\downarrow)^*$. The order for circuit (4.20) is the following combination of these two orders

(4.24) $(y\uparrow;\ p\uparrow;\ a\uparrow;\ c\uparrow, x\uparrow;\ y\downarrow;\ c\downarrow, (p\downarrow;\ a\downarrow;\ x\downarrow))^*.$

Circuit (4.12) is the composition of circuits (4.20) and (4.23). The q-composition of their expressions, (4.22) and (4.24), is (4.11), which shows that circuit (4.12) implements (4.11).

We next discuss the design of a binary semaphore. A binary semaphore is a component with a left and a right environment.



It implements the order expressed by

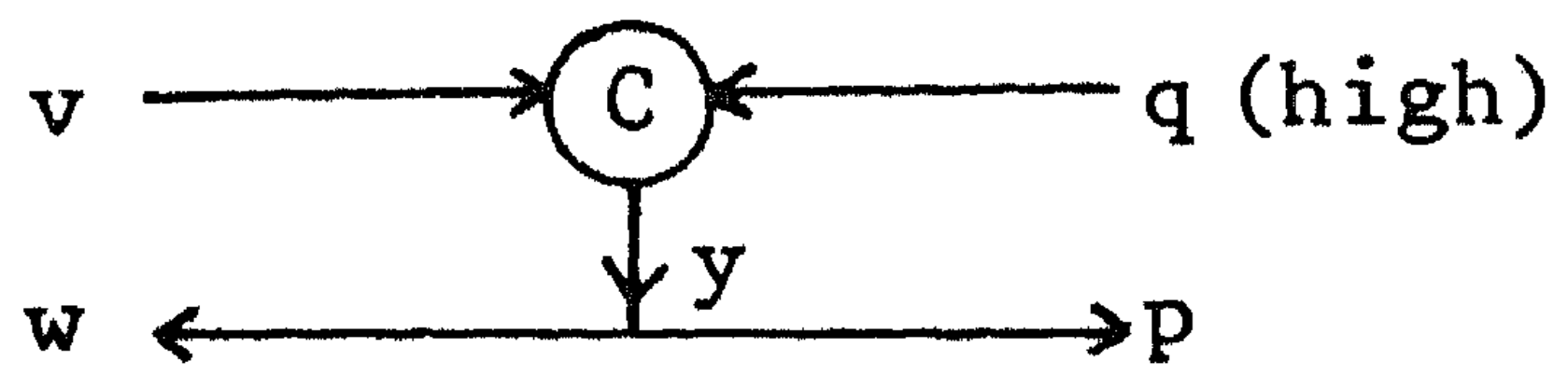
(4.25) $$(v\uparrow;\ t0;\ w\uparrow;\ v\downarrow;\ w\downarrow)^*,\ (t0;\ p\uparrow;\ q\uparrow;\ p\downarrow;\ q\downarrow)^*.$$

Expression (4.25) may be rewritten as

(4.26) $$v\uparrow;\ (t0;\ w\uparrow;\ v\downarrow;\ w\downarrow;\ v\uparrow)^*,\ (t0;\ p\uparrow;\ q\uparrow;\ p\downarrow;\ q\downarrow)^*.$$

Consider again circuit (4.16). We split output y into outputs w and p:

(4.27)



The order for circuit (4.27) is obtained by replacing in (4.17) $y\uparrow$ and $y\downarrow$ by $w\uparrow, p\uparrow$ and $w\downarrow, p\downarrow$, respectively:

(4.28) $$v\uparrow;\ (w\uparrow, p\uparrow;\ v\downarrow, q\downarrow;\ w\downarrow, p\downarrow;\ v\uparrow, q\uparrow)^*.$$

Next we split the environment of circuit (4.27) into a left and a right environment, each of which uses 4-cycle signaling: the order for the left environment is $v\uparrow;\ (w\uparrow;\ v\downarrow;\ w\downarrow;\ v\uparrow)^*$ and for the right environment $(p\uparrow;\ q\downarrow;\ p\downarrow;\ q\uparrow)^*$. The result of this splitting is that the environment as a whole imposes less order. The following orders, that were present in (4.28), have been lost:

$w\uparrow;\ q\downarrow$

$p\uparrow;\ v\downarrow$

$w\downarrow;\ q\uparrow$

$p\downarrow;\ v\uparrow$.

Deleting these orders from (4.28) yields

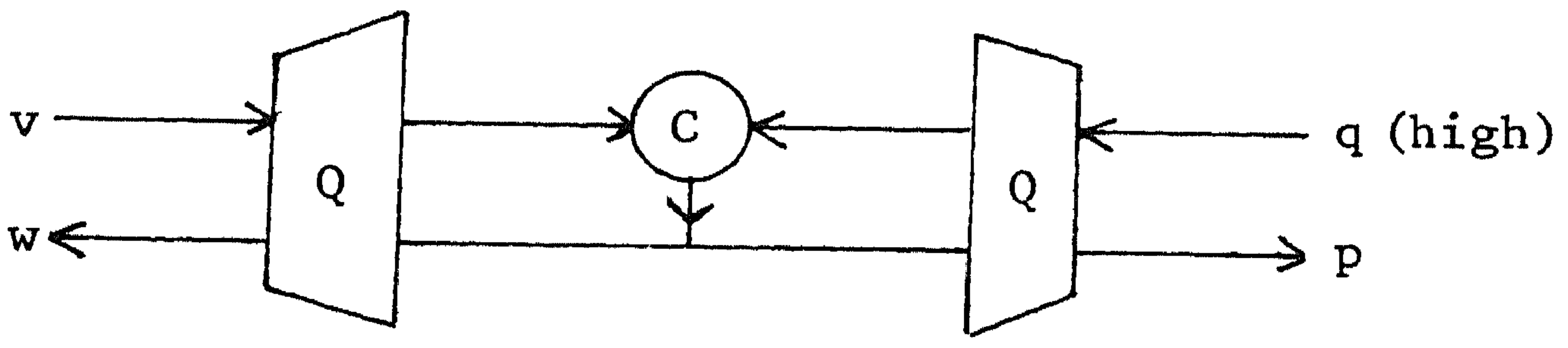
(4.29) $$v\uparrow;\ ((w\uparrow;\ v\downarrow),(p\uparrow;\ q\downarrow);\ (w\downarrow;\ v\uparrow),(p\downarrow;\ q\uparrow))^*.$$

Expression (4.29) is equivalent to

(4.30) $$\begin{gathered} v\uparrow;\ (t0;\ w\uparrow;\ v\downarrow;\ t1;\ w\downarrow;\ v\uparrow)^*. \\ (t0;\ p\uparrow;\ q\downarrow;\ t1;\ p\downarrow;\ q\uparrow)^*. \end{gathered}$$

Expression (4.30) exhibits more order than (4.26). But we know what we can do about that: add a quick return linkage. The effect of adding a quick return linkage on v and w is that $p\uparrow;\ q\downarrow$ does not have to precede $w\downarrow;\ v\uparrow$, i.e. that in expression (4.30) $t1;\ w\downarrow;\ v\uparrow$ is replaced by $t1,(w\downarrow;\ v\uparrow)$. Another quick return linkage on p and q replaces $t1;\ p\downarrow;\ q\uparrow$ by $t1,(p\downarrow;\ q\uparrow)$. Thus the circuit

(4.31)



implements

(4.32) $$\begin{gathered} v\uparrow;\ (t0;\ w\uparrow;\ v\downarrow;\ t1,(w\downarrow;\ v\uparrow))^*, \\ (t0;\ p\uparrow;\ q\downarrow;\ t1,(p\downarrow;\ q\uparrow))^*. \end{gathered}$$

Expression (4.32) is equivalent to (4.26), which shows that circuit (4.31) is a binary semaphore.

Like we showed with component trisem in Section 2.2, we can connect two of these binary semaphores to obtain a ternary semaphore, or k binary semaphores to obtain a (k+1)-ary semaphore.

5. CONCLUSIONS

In the preceding sections we have discussed a number of topics associated with the translation of components denoting partially ordered computations into delay-insensitive circuits. There are also a number of topics we have not touched upon or only touched upon lightly. We outline some of the latter topics in this section.

There are a number of ways in which the translation of components into circuits can be achieved. We discuss two of them.

Some circuits can be translated into regular structures known as programmable logic arrays (PLA's). As an example we consider the full-adder element discussed in Section 2.3 (Example 2.8). We can partition its alphabet into an input and an output alphabet. Such a partition may be given to start with, but there are also efforts being made to derive the partition from the trace set. We give an example of such a derivation that seems applicable in simple cases.

Let $S = \langle T,A \rangle$ be a trace structure. We call two symbols a and b in A related if

$$\{u \in A^* \mid ua \in \mathrm{PREF}(T)\} = \{u \in A^* \mid ub \in \mathrm{PREF}(T)\}.$$

This is an equivalence relation on A. We call each equivalence class containing at least two symbols an input and the symbols in such a class input symbols. The singleton equivalence classes are then the outputs and the symbols in them output symbols.

The definition above yields for the full-adder element three binary inputs: (a0,a1), (b0,b1), and (c0,c1). The other symbols are output symbols. For each output symbol we list the input symbols that precede it, i.e. those that are separated from it by at least one semicolon:

$$d0:\ a0 \wedge b0 \vee a0 \wedge b1 \wedge c0 \vee a1 \wedge b0 \wedge c0$$

$$d1:\ a1 \wedge b1 \vee a0 \wedge b1 \wedge c1 \vee a1 \wedge b0 \wedge c1$$

$$s0:\ a0 \wedge b0 \wedge c0 \vee a1 \wedge b1 \wedge c0 \vee a0 \wedge b1 \wedge c1 \vee a1 \wedge b0 \wedge c1$$

$$s1:\ a0 \wedge b0 \wedge c1 \vee a1 \wedge b1 \wedge c1 \vee a0 \wedge b1 \wedge c0 \vee a1 \wedge b0 \wedge c0$$

Seitz shows an almost delay-insensitive PLA for a full-adder element on p. 251 of SEITZ 80. The 14 terms in the four lines above correspond exactly to the 14 crossings in his PLA at which the horizontal and vertical wires are connected. Seitz's realization may thus be derived directly from our program text.

For more complicated components - components with nested repetition, for example - we can draw upon work done on recognizing regular languages (FLOYD & ULLMAN 80). There is a difference between recognizing a regular language and executing a component: when executing a component, i.e. selecting a trace, the input symbols are the only ones to be recognized. At the appropriate positions in the trace recognized the output symbols in that trace must be generated. These output symbols are then again inputs to and recognized by other components.

An interesting method of recognizing regular languages is described in FOSTER & KUNG 81. The machinery they propose consists of so-called building blocks that communicate in a pipe-lined fashion. Since such a scheme does not require global communication, it seems well-suited for a delay-insensitive implementation. A necessary extension of their machinery in order to use it for the execution of our components is the introduction of a building block that corresponds to the comma.

We have chosen trace theory as the basis of our computations, and we have showed how states (and state transitions) can be derived from trace sets. A more traditional approach is to start with states. Such an approach seems well-suited for clocked systems: the clock separates the computation into a number of steps, each step leading from one state to the next. In delay-insensitive circuits, however, there do not have to be moments at which the circuit is in a well-defined state. We are dealing with events that are only partially ordered. Trace theory seems to be well-suited as a basis for such computations. In BROCK & ACKERMAN 81 it is shown that defining the semantics of (nondeterministic) components as functions from input streams to sets of output streams is not a good approach, since it does not reflect the order between individual items in different streams. They exhibit this shortcoming in a number of approaches described in the literature. Trace theory does not have this defect.

There are at least three extensions to the material presented that need further attention. First of all, we need more theorems on trace structures. These theorems must enhance our capability of designing partially ordered computations and arguing about them. They must be aimed at leaving the realm of the individual traces. A nice example of such a theorem is Property 2.11. It defines the net effect of the composite in terms of the net effects of the parts. Finding such theorems requires good ways of characterizing net effects of computations. The characterization of stacks in terms of well-nested sequences (Example 2.7) was an effort in that direction.

Secondly, we need to study the problem of laying out transistor-connection diagrams (so-called schematics) in the plane. This mapping must be done by a compiler, without interference or consultation of the designer. The fact that the mapping problem is still ill-understood is the main driving force behind the current popularity of graphical "design tools". Such design tools are usually design obstacles in the sense that they prevent the designer from ignoring the physical realization.

Thirdly, we need to extend the notation in which we specify our components. A good extension would be the introduction of values and ports via which these values are communicated. It would make our notation look more like the one discussed in HOARE 78a. Values and ports have types. The declaration of a type defines a method of representing values and ports of that type in terms of symbols. Such a type could, for example, be 16-bit integer. The declaration specifies, among other things, whether the bits of such an integer are communicated serially of in parallel. The components we introduce will thus give us new modes of expression. By a proper choice of components we want to arrive at a mode of expression that one would customarily call a "higher level programming language".

6. ACKNOWLEDGEMENTS

It would have been impossible to write these notes without the cooperation of Jan van de Snepscheut. Many of the concepts introduced are either his or took shape during and as a result of our scientific collaboration.

Carver Mead introduced me to the fascinating world of VLSI. In many long discussions we tried to grasp the essence of computing machinery. His insights have greatly contributed to my understanding of VLSI.

Chuck Seitz taught me the art of dealing with delays. Virtually all I known of delay-insensitive signaling can in some way or another be traced back to him. Charles Molnar and Jo Ebergen are gratefully acknowledged for deepening my insight in delay-insensitive computations.

Arjen Lenstra and Paul Vitányi suggested part of the mathematical formalism used in these notes. Jan Tijmen Udding pointed out a number of shortcommings in Section 4.2. While preparing these notes I discussed their contents with Alain Martin, Randy Bryant, and Young-il Choo. The material has also served as the subject of attention at Edsger W. Dijkstra's Tuesday Afternoon Club. Both occasions resulted in a number of improvements. Acknowledgements are also due to the members of the MC-THE VLSI Working Group, who critically evaluated most of the material presented.

Eindhoven University of Technology and California Institute of Technology are gratefully acknowledged for giving me the opportunity to commute regularly between these two institutes. The combination of their scientific atmospheres proved an excellent breeding-place for the research reported.

7. REFERENCES

BROCK, J.D. & ACKERMAN, W.B. (1981), *Scenarios: a model of non-determinate computation*, Computation Structures Group Memo 206, Massachusetts Institute of Technology, Cambridge, Mass. (Also in the proceedings of the International Colloquium on Formalization of Programming Concepts, Peniscola, Spain. Lecture Notes in Computer Science 107, Springer-Verlag, Berlin)

CAMPBELL, R.H. & HABERMANN, A.N. (1974), *The specification of process synchronization by path expressions*, in "Operating Systems" (Eds. E. Gelenbe & C. Kaiser), pp. 89-102. Lecture Notes in Computer Science 16. Springer-Verlag, Berlin.

CLARK, W.A. (1980), *From electron mobility to logical structure: a view of integrated circuits*, ACM Comp. Surveys 12,3, pp. 325-356.

FLOYD, R.W. & ULLMAN, J.D. (1980), *The compilation of regular expressions into integrated circuits*, in "Proceedings of the 21st Annual Symposium on Foundations of Computer Science", pp. 260-269. IEEE Computer Society. (Also in Journal of the ACM, 29,3, pp. 603-622)

FOSTER, M.J. & KUNG, H.T. (1980), *Recognize regular languages with programmable building-blocks*, in "VLSI 81 - Very Large Scale Integration", pp. 75-84, Academic Press, London.

GINSBURG, S. (1966), *The Mathematical Theory of Context-free languages*. McGraw-Hill, New York, N.Y.

HOARE, C.A.R. (1978), *Towards a theory of communicating sequential processes*, Preliminary Report, Oxford University, Oxford.

HOARE, C.A.R. (1978a), *Communicating sequential processes*, Comm. ACM. 21,8, pp. 666-677.

HOPCROFT, J.E. & ULLMAN, J.D. (1969), *Formal Languages and their Relation* *to Automata*, Addison-Wesley, Reading Mass.

LAUER, P.E. (1981), *Synchronization of concurrent processes without globality assumptions*, ACM Sigplan Notices 16,9, pp. 66-80.

MARTIN, A.J. (1981), *An axiomatic definition of synchronization primitives*, Acta Informatica 16, pp. 219-235.

MEAD, C. & CONWAY, L. (1980), *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass.

MILNER, R. (1980), *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag, Berlin.

REM, M. (1979), *Mathematical aspects of VLSI design*, In "Caltech Conference on VLSI" (Ed. C.L. Seitz), pp. 55-64, California Institute of Technology, Pasadena, Calif.

REM, M. (1982), *On the design of restoring logic circuitry*, Eindhoven University of Technology, Eindhoven, Netherlands. (To appear in the proceedings of the 1982 CREST Summerschool on VLSI, Bristol, U.K.)

SEITZ, C.L. (1979), *Self-timed VLSI systems*, In "Caltech Conference on VLSI" (Ed. C.L. Seitz), pp. 345-355, California Institute of Technology, Pasadena, Calif.

SEITZ, C.L. (1980), *System timing*, In MEAD & CONWAY 80, pp. 218-262.

STUCKI, M.J. & COX, JR. J.R. (1979), *Synchronization strategies*, In "Caltech Conference on VLSI" (Ed. C.L. Seitz), pp. 375-394, California institute of Technology, Pasadena, Calif.

VAN DE SNEPSCHEUT, J.L.A. (1982), *An inventory of trace theory*, Internal report JAN83a, Eindhoven University of Technology, Eindhoven, Netherlands.

PROCESSES AND THE DENOTATIONAL SEMANTICS OF CONCURRENCY

J.W. de Bakker & J.I. Zucker
Mathematisch Centrum, Amsterdam/SUNY at Buffalo, USA

1. INTRODUCTION

The aim of this paper is to present a mathematical study of the semantics of a variety of language concepts in the area of *concurrency*. We shall be concerned with three fundamental notions in this field: *parallel composition, synchronization*, and *communication*, and we shall develop a general framework in which definitions and properties of these notions can be discussed in a systematic way.

The emphasis in the paper is on *definitions* - rather than on pragmatic use - of language concepts. We shall use the methodology of *denotational* semantics. "Denotational" should be contrasted here with "operational": The key idea of the former approach is that expressions in a programming language denote values in mathematical domains equipped with an appropriate structure, whereas in the latter the operations as prescribed by the language constructs are modelled by steps performed by some suitable abstract machine.

In the denotational semantics of sequential programming concepts, a central role is played by the notion of (state-transforming) *function*. Let us use Σ, with elements σ, for the set of *states*. For the present purposes, it suffices to define a state as a mapping from program variables x,y,... to values such as 0,1,... . The denotational meaning of a simple command such as the assignment statement x := x+1 is a function $\phi: \Sigma \to \Sigma$, defined by $\phi(\sigma) = \sigma'$, where $\sigma'(x) = \sigma(x)+1$, and $\sigma'(y) = \sigma(y)$ for all $y \not\equiv x$. Also, the meaning of a composite command, formed by sequential composition ";", such as x := x+1; y := x+y is obtained by forming the function composition $\phi_2 \circ \phi_1$, where ϕ_1 and ϕ_2 are the meanings of the statements x := x+1 and y := x+y, respectively. When we admit nondeterminacy, the situation changes somewhat in that the meaning of a statement is now a function from states to sets of states with a certain structure. Using $\mathcal{P}$ for "power set of", we now use functions $\phi\colon \Sigma \to \mathcal{P}(\Sigma)$. Here as well, composition is easy to define: $\phi_1 \circ \phi_2 = \lambda\sigma.\{\sigma' \mid \sigma' \in \phi_1(\sigma'')$ for some $\sigma'' \in \phi_2(\sigma)\}$, and no *essential* extension of the traditional view of a statement having a state transformation as its meaning is necessary. A fundamental change in this view is needed, however, for the denotational treatment of *parallel* composition. Let $S_1 \parallel S_2$ denote parallel execution of S_1 and S_2: Statements S_1 and S_2 - in the example allowed to share their variables - are executed by arbitrary interleaving of the constituent elementary actions of S_1 and S_2. Consider, for example, a simple program (*): $(A_1;A_2) \parallel (B_1;B_2)$, with A_i, B_i elementary actions (such as x := x+1), and let ϕ_i, ψ_i be the respective meanings of A_i, B_i. Now what happens if we take the ϕ_i, ψ_i simply as functions: $\Sigma \to \Sigma$? We form the compositions $\phi = \phi_2 \circ \phi_1$, $\psi = \psi_2 \circ \psi_1$ and try to define a resulting function *merge* (ϕ,ψ). Here we are stuck, since having formed the compositions ϕ, ψ, we no longer have available their respective operands ϕ_i, ψ_i. (Remember that what we want as resulting function is the union of the (six) possibilities $\phi_2 \circ \phi_1 \circ \psi_2 \circ \psi_1$, $\phi_2 \circ \psi_2 \circ \phi_1 \circ \psi_1, \ldots, \psi_2 \circ \psi_1 \circ \phi_2 \circ \phi_1$.) In an operational approach, the problem does not arise in this form: A *trace* is kept of the computation, e.g. in the form of the (set of the) sequence(s) of elementary actions generated while executing the program, and the meaning of $S_1 \parallel S_2$ is simply the shuffle (in the language theoretic sense) of the traces corresponding to S_1 and S_2. (Other operational approaches are also possible, see e.g. [31,49]. However, they all involve suitably structured sequences of elementary steps.) This preserving of intermediate information in order to be able to describe the final result of interleaving is crucial for a proper treatment of parallellism, and is in fact what we shall do as well in our denotational approach. The basic idea is to extend the notion of function to that of *process*. Here "process" is a generic term, referring to a variety of mathematical objects which have one important property in common, viz. that they are constituted in some way from (possibly infinite) sets of (possibly infinite) sequences. For the example language considered above, the corresponding notion of process is an extension of that of state-transforming function in that it is still a function but now includes the information on how it was built up from the - possibly infinite - sequences of its elementary components. In this introduction we shall not be more precise about the notion of process. What we do underline is that in our theory a process is a semantic rather than a syntactic notion: it is a feature of the mathematical model rather than of the program text

Section 2 of the paper presents the notion of process in some detail. A rigorous treatment of this requires some mathematical machinery involving tools from metric topology. A fundamental role is played by *equations* for *domains* of processes. Such equations are solved essentially by completion techniques - reminiscent of the way Cantor constructed the real numbers from the rationals. Next, the central *operations* upon processes are defined. We consider the convenience in formulating these definitions as an important accomplishment of the theory of processes. Processes are finite or infinite. Defining the operations for the finite cases requires specific attention; the infinite ones are each time obtained in a standard way by continuity arguments. Some of the more tedious mathematical arguments are relegated to the appendices; in section 2 we concentrate on those results which are necessary for an understanding of the central sections of our paper. For the reader who wants to skip *all* mathematical details we provide a brief summary of the relevant results at the end of the section. Sections 3 to 8 constitute the applied part of the paper. In these, it is shown how a rigorous and concise semantics can be designed for certain central notions in concurrency, by an appropriate synthesis of the use of processes with that of more traditional ideas of denotational semantics. Section 3 concentrates on flow of control: It considers a simple language with elementary actions, sequential composition and nondeterministic choice, and iteration or recursion. Adding parallel composition ("$\|$") to this requires for its semantics a rather simple process domain, the so-called *uniform* processes. Iteration and recursion are dealt with in a relatively straightforward way by certain limit constructions. We already mention that an appeal to Banach's fixed point theorem will replace the familiar least fixed point approach of denotational semantics based on complete partially ordered sets. The section also discusses how the *yield* of a uniform process p can be derived from the set of all *paths* in p.

In section 4 we add *synchronization* to the language(s) of section 3. Synchronization restricts the set of all possible interleavings of sequences of elementary actions, and a general mechanism to model this is studied. Section 5 refines the theory by introducing the notion of state - suppressed in sections 3 and 4 - and assignments, and discusses the required extensions to the notions of processes and their yields. Processes are no longer uniform, but depend on the state as an argument, and the previous definitions have to be modified accordingly. As special feature we mention that unbounded nondeterminacy can be dealt with without any additional measures. Section 6 combines the ideas of sections 4 and 5, in that synchronization is now considered for non-uniform processes. Among the topics studied are deadlock, and synchronization through guards in guarded commands. Section 7 extends synchronization to *communication*: At points of synchronization in the parallel execution values are passed from one process to another. A further extension of the notion of process is needed to deal with this. Two major examples of languages with communication are treated: Hoare's Communicating Sequential Processes ([34]), and Milner's Calculus for Communicating Systems ([44]). In section 8 we finally discuss some miscellaneous notions in concurrency, without providing a full treatment as was done in the preceding sections. In the appendices a number of mathematical details omitted in section 2 are filled in.

A few words on the emphasis on *denotational* in the title of our paper are in order. Our arguments for the claim that our approach is denotational are twofold: (i) the systematic use of mathematical models which are used as range for the valuation mappings assigning meaning to the various programming constructs, (ii) the systematic way of adhering to the compositionality principle, allowing homomorphic valuations. However, we are aware of the fact that we have to pay a price for this. The mathematical model contains various notions which, though denotational in style, are operational in spirit. These include the "history" feature of the notion of process itself, and the use of socalled silent moves in dealing with synchronization and recursion.

There is a vast amount of literature on concurrency, and a good part of these papers involve some discussion of the operational semantics of the notion(s) in concurrency. Our understanding of concurrency has been profoundly influenced by the work of R. Milner, starting with [42], continued in papers such as [30,40,43], and culminating in [44]. Though the latter work is primarily operational in spirit, there is still a lot in it which recalls its author's denotational period. Also, for an intuitive understanding of the central notions in concurrency it is an invaluable source. The various notions of process to be studied below will be introduced as solutions of domain equations. The introduction of equations of this type is due to D.Scott - dating back to perhaps the most famous equation for *reflexive* domains: $D = D \to D$ - and has been treated extensively in, e.g., [54] or, more recently, in [55]. A very nice textbook on denotational semantics in general and domain equations in particular is Stoy [57]. (A more introductory text on denotational semantics is Gordon [28]; many advanced topics are treated in Milne & Strachey [41].) Scott's theory did not include nondeterminacy or concurrency, and an extension of his theory dealing with these concepts was proposed by Plotkin ([48]), later simplified somewhat by Smyth ([56]; c.f. also [39]). The first time we saw a domain equation intended to be used for modelling concurrency was in Bekic [12]. In the work of Plotkin and Smyth, domain equations are solved by category-theoretic methods which may be somewhat demanding for the uninitiated reader. We prefer to use other tools, viz. those of metric topology. The use of these has been advocated in recent years by M. Nivat and his colleagues, and applied succesfully in a variety of applications having to do with infinite words or infinite trees modelling infinite computations and the semantics of recursive program schemes with nondeterminacy [5,6,45,46]. The mathematical foundations of our work - as described in section 2 - owes a considerable debt to the work of Nivat's school - though the specific way we use topological completion techniques to solve equations seems to be new.

Our own first venture into the realm of (infinite) processes was De Bakker [9]. Lacking in that paper was a sound mathematical basis for the notion of process. The present topological treatment was first described in De Bakker & Zucker [11], reporting on research which was started during a most enjoyable stay of the first author at Bar-Ilan University and the Weizmann Institute during the summer of 1981.

Further references to the literature - in particular concerned with the various concepts in concurrency we shall encounter in these notes - will be given as we go along.

A preliminary version of this paper was used as lecture notes for the Fourth Advanced Course on Foundations of Computer Science, Amsterdam, June 1982. We are indebted to the students of this course for various questions and comments. We also acknowledge the suggestions of the referee, and discussions with J.A. Bergstra, J.W. Klop, R. Kuiper, L. Lamport, J.J.Ch.Meyer, and G. Plotkin.

2. PROCESSES

In this section we show how processes p can be introduced as elements of domains P which are obtained as solutions of domain equations of the form (*): $P = \mathcal{T}(P)$. The techniques used to solve (*) are taken from metric topology. A variety of equations (*) is considered, determining a variety of process domains of increasing complexity. Furthermore, a number of operations upon processes are defined, viz. composition $(p_1 \circ p_2)$, union $(p_1 \cup p_2)$, and merge $(p_1 \| p_2)$, and various properties of these operations are presented. A few of the proofs of the supporting mathematical facts are not contained in this section but can be found in the appendix. A brief summary of the relevant results is given at the end of the section.

We begin by recalling a few basic facts from metric topology. We assume known the notions of metric space, Cauchy sequence (CS) in a metric space, isometry (distance-preserving bijection), limits and closed sets, completeness of a metric space, and the theorem stating that each metric space (M,d) can be completed to (i.e. isometrically embedded in) a complete metric space. Throughout our paper, we shall only consider spaces (M,d) such that the metric d has values in the interval $[0,1]$.

These notions are sufficient to solve the first domain equation for processes. This equation is very simple, and introduced only for the sake of illustrating the method used in solving such equations. Let A be any set. We consider the equation

$$P = \{p_0\} \cup (A \times P) \tag{2.1}$$

where p_0 is the *nil* process, and "×" is the usual cartesian product. Intuitively, it is not difficult to see that the (greatest) solution set P should consist of p_0, all finite sequences of the form $\langle a_1, \langle a_2, \ldots, \langle a_n, p_0\rangle \ldots\rangle\rangle$, for $n \geq 1$, together with all infinite sequences $\langle a_1, \langle a_2, \ldots\rangle\rangle$. The role of the nil process p_0 may be somewhat unusual in this equation, in that it replaces the more familiar empty sequence. However, it will remain with us all through the paper, and we ask the reader to exercise some patience in trying to appreciate its use.

We now obtain the solution of (2.1) in a more rigorous manner:

<u>DEFINITION 2.1</u>. Let (P_n, d_n), $n = 0,1,\ldots$, be a collection of metric spaces defined inductively by: $P_0 = \{p_0\}$, $d_0(p',p'') = 0$ (since $p',p'' \in P_0 \Longleftrightarrow p' = p'' = p_0$), $P_{n+1} = \{p_0\} \cup (A \times P_n)$, d_{n+1} is given by: $d_{n+1}(p',p'') = 0$ if $p' = p'' = p_0$, $d_{n+1}(p',p'') = 1$ if $p' = p_0$, $p'' \neq p_0$ or $p' \neq p_0$, $p'' = p_0$. Otherwise, $p' = \langle a_1, p_1\rangle$, $p'' = \langle a_2, p_2\rangle$ for some $a_1, a_2 \in A$, $p_1, p_2 \in P_n$, and we put

$$d_{n+1}(p',p'') = d_{n+1}(\langle a_1,p_1\rangle,\langle a_2,p_2\rangle) = \begin{cases} 1, & \text{if } a_1 \neq a_2 \\ \frac{1}{2}\, d_n(p_1,p_2), & \text{if } a_1 = a_2 \end{cases}$$

It is not difficult to verify that d_n is indeed a metric on P_n. As next step, we define $P_\omega \overset{\text{df.}}{=} \bigcup_n P_n$ and $d \overset{\text{df.}}{=} \bigcup_n d_n$. E.g., take $p' = \langle a_1,\langle a_2,\langle a_3,p_0\rangle\rangle\rangle$, $p'' = \langle a_1,\langle a_2,\langle a_3,\langle a_4,p_0\rangle\rangle\rangle\rangle$. Then $d(p',p'') = d_m(p',p'')$ (any $m \geq 4$) $= \frac{1}{2}\, d_{m-1}(\langle a_2,\langle a_3,p_0\rangle\rangle,\langle a_2,\langle a_3,\langle a_4,p_0\rangle\rangle\rangle) = \ldots = \frac{1}{8}\, d_{m-3}(p_0,\langle a_4,p_0\rangle) = \frac{1}{8} * 1 = \frac{1}{8}$.

DEFINITION 2.2.

a. $P_\omega = \bigcup_n P_n$, $d = \bigcup_n d_n$

b. (P,d) is the *completion* of (P_ω,d).

Standard properties of the completion technique yield that we may take P as consisting of P_ω together with all limit points $p = \lim_n p_n$, with $\langle p_n\rangle_n$ a Cauchy sequence such that $p_n \in P_n$. It is now straightforward to show that

LEMMA 2.3. P *satisfies* (2.1).

Proof. Let $P' \overset{\text{df.}}{=} \{p_0\} \cup (A \times P)$. We define isometries $\phi\colon P \to P'$, $\psi\colon P' \to P$ in the following manner. First we consider ϕ. If $p = p_0$, we take $\phi(p) = p_0$; clearly, $\phi(p) \in P'$ in that case. Otherwise, $p = \lim_n p_n$ with $\langle p_n\rangle_n$ a CS (if $p \in P_n$, for some $n \geq 1$, p is identified with a CS which is eventually constant), and we may assume without lack of generality that $p_n = \langle a,q_n\rangle$, for some a and all n, such that $\langle q_n\rangle_n$ is also a CS. Now let $q = \lim_n q_n$. We take $\phi(p) = \langle a,q\rangle$. We leave the definition of ψ, and verification that ϕ,ψ are indeed isometries to the reader. □

The trouble taken to solve (2.1) may seen somewhat inordinate. It was done this way to familiarize the reader with this style of argument - which will pay off later - rather than for the solution of this problem in its own right.

Processes p which are elements of sets P as defined (e.g.) by equation (2.1) have a *degree*, written as *deg*(p), and defined in

DEFINITION 2.4. $deg(p_0) = 0$, $deg(p) = n$ if $p \in P_n \backslash P_{n-1}$, for some $n \geq 1$, and $deg(p) = \infty$, otherwise.

For processes p,q in P as defined in (2.1) we now give the definition of their *composition* $p \circ q$:

DEFINITION 2.5. $p \circ q$ is defined (by induction on $deg(q)$)

a. $p \circ p_0 = p$, $p \circ \langle a,q' \rangle = \langle a, p \circ q' \rangle$ if $deg(\langle a,q' \rangle) < \infty$

b. $p \circ \lim_i q_i = \lim_i (p \circ q_i)$, for q_i finite

Example: $\langle a_1, \langle a_2, p_0 \rangle\rangle \circ \langle a_3, p_0 \rangle = \langle a_3, \langle a_1, \langle a_2, p_0 \rangle\rangle\rangle$. We see that composition is (almost) concatenation in reverse order.

LEMMA 2.6.

a. *If* $\langle q_i \rangle_i$ *is a* CS *then so are* $\langle p \circ q_i \rangle_i$ *(this justifies definition* 2.5b) *and* $\langle q_i \circ p \rangle_i$.

b. "$\circ$" *is continuous in both arguments, i.e.,* $(\lim_i p_i) \circ q = \lim_i (p_i \circ q)$, *and* $p \circ \lim_i q_i = \lim_i (p \circ q_i)$, *for all* p_i, q_i *such that* $\langle p_i \rangle_i, \langle q_i \rangle_i$ *are* CS.

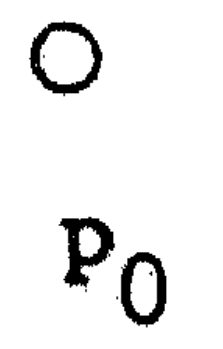
c. "$\circ$" *is associative*

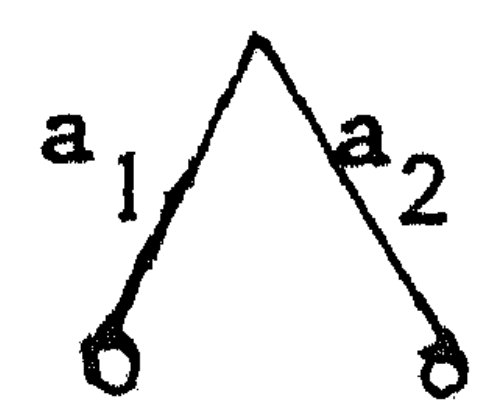
Proof. This lemma being a special case of later results, we omit its proof. □

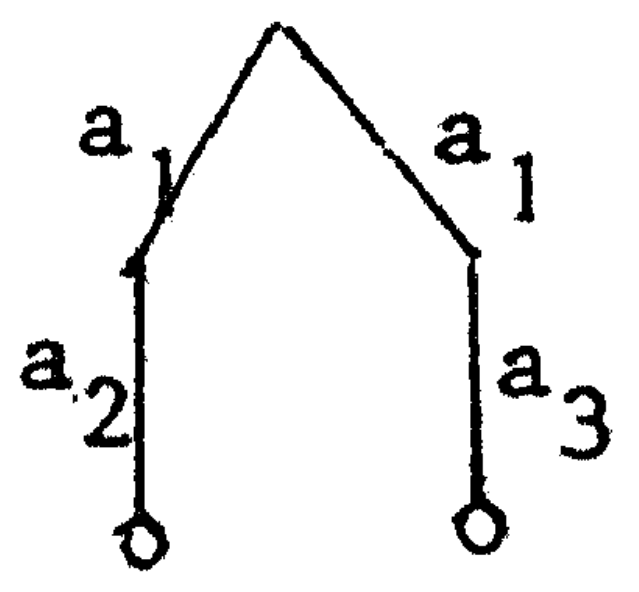
We now turn to the solution of a more interesting equation. The resulting processes are not simply (finite or infinite) sequences, but - roughly, a precise statement follows - *sets* of such sequences. We want to solve

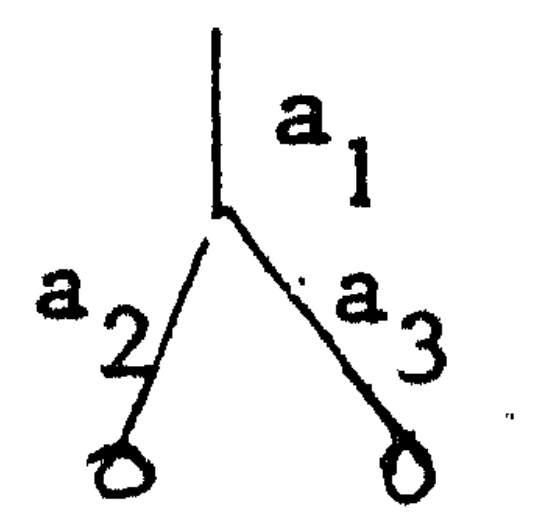
$$(2.2) \qquad P = \{p_0\} \cup \mathcal{P}_c(A \times P)$$

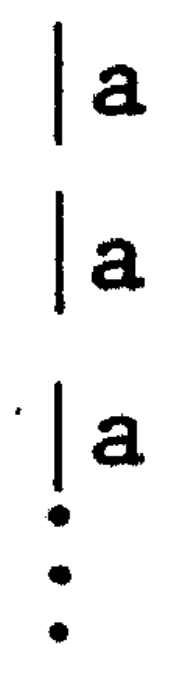
where $\mathcal{P}(\cdot)$ denotes all subsets of $(\cdot)$, and $\mathcal{P}_c(\cdot)$ all *closed* subsets of $(\cdot)$ (closed with respect to the metric to be introduced in a moment). Before going into the mathematical details, we consider a few simple examples. Possible elements of P are p_0, $\{\langle a_1,p_0 \rangle, \langle a_2,p_0 \rangle\}$, $\{\langle a_1, \{\langle a_2,p_0 \rangle\}\rangle, \langle a_1, \{\langle a_3,p_0 \rangle\}\rangle\}$, $\{\langle a_1, \{\langle a_2,p_0 \rangle \ \langle a_3,p_0 \rangle\}\rangle\}$, or $\{\langle a, \{\langle a, \{\langle a, \ldots \rangle\}\rangle\}\rangle\}$. In pictures, these processes may be represented by

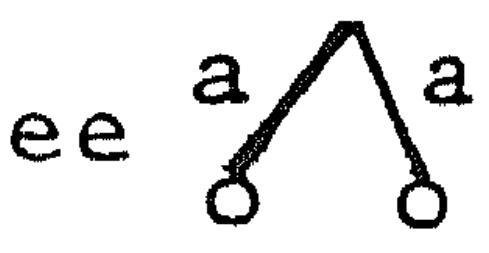










We see that these processes closely resemble (unordered) trees. However, as essential difference we have that "nodes" in a process have a set - rather than a multiset - of successors: A tree  has no corresponding process.

The topological treatment of the solution of (2.2) requires some preparations. Firstly, we extend distances d as follows:

DEFINITION 2.7. Let (M,d) be a metric space and let X,Y be subsets of M. We define

a. $d(x,Y) = \inf_{y \in Y} d(x,y)$

b. $d(X,Y) = \max(\sup_{x \in X} d(x,Y), \sup_{y \in Y} d(y,X))$

(By convention, $\inf \emptyset = 1$, $\sup \emptyset = 0$.)

Remark. The distance d(X,Y) is the Hausdorff distance between sets. It should be distinguished from $d'(X,Y) = \inf_{x \in X, y \in Y} d(x,y)$, which does not determine a metric.

For the Hausdorff distance we have

LEMMA 2.8. *Let* (M,d) *be a metric space, and let* $P_c(M)$ *be the collection of all closed subsets of* M. *Then* $(P_c(M),d)$ *is a metric space.*

Proof. See [19] or [22]. □

Remark. Given a metric space (M,d), d is said to be an *ultrametric* on M if it satisfies the "strong triangle inequality" $\forall x,y,z \in M\, [d(x,z) \le \max(d(x,y),d(y,z))]$. It is easy to see that if d is an ultrametric on M, then so is the induced Hausdorff metric on $P_c(M)$. It will follow (as can easily be shown) that every process domain P considered in this article will have an ultrametric with, moreover, $\max \{d(p,q) \mid p,q \in P\} = 1$.

An important technical result which plays a central role in the theory developed below is the following theorem of Hahn [29](cf.also [22]):

THEOREM 2.9. *If* (M,d) *is complete then so is* $(P_c(M),d)$. *Also, for* $\langle X_n \rangle_n$ *a* CS *in* $P_c(M)$, *we have that*

$$\lim_n X_n = \{x \mid x = \lim_n x_n,\ x_n \in X_n,\ \langle x_n \rangle_n \text{ a CS in } M\}.$$

Proof. See Appendix A.

We now proceed with the construction solving (2.2). We introduce metric spaces (P_n,d_n), extending the techniques as applied before with sets and their (Hausdorff) distances:

DEFINITION 2.10. The collection of metric spaces (P_n,d_n), $n = 0,1,\ldots$, is defined by $P_0 = \{p_0\}$, $d_0(p',p'') = 0$, $P_{n+1} = \{p_0\} \cup \mathcal{P}(A \times P_n)$, $d_{n+1}(p',p'')$ is as before for $p' = p_0$ or $p'' = p_0$. Otherwise, $p' = X \subseteq A \times P_n$, $p'' = Y \subseteq A \times P_n$, and we take $d_{n+1}(X,Y)$ as the Hausdorff distance induced by the distance between points $d_{n+1}(x,y)$, where (as before), for $x = \langle a_1,p_1\rangle$, $y = \langle a_2,p_2\rangle$

$$d_{n+1}(x,y) = \begin{cases} 1, \text{ if } a_1 \neq a_2 \\ \frac{1}{2}\, d_n(p_1,p_2), \text{ if } a_1 = a_2 \end{cases} .$$

Example. Take $a_2 \neq a_3$. Then $d_2(\{\langle a_1,\{\langle a_2,p_0\rangle,\langle a_3,p_0\rangle\}\rangle\},\{\langle a_1,\{\langle a_2,p_0\rangle\}\rangle, \langle a_1,\{\langle a_3,p_0\rangle\}\rangle\}) = \frac{1}{2}$.

As before, we take $P_\omega = \bigcup_n P_n$, $d = \bigcup_n d_n$, and (P,d) is defined as the completion of (P_ω,d). We have

THEOREM 2.11. $P = \{p_0\} \cup \mathcal{P}_c(A \times P)$, *where* $\mathcal{P}_c(\cdot)$ *stands for all subsets of* $(\cdot)$ *which are closed with respect to the metric* d.

The proof needs a definition and a lemma.

DEFINITION 2.12.

a. Let $p \in P_\omega$. We define $p^{(n)}$, $n = 0,1,\ldots$, by: If $p = p_0$ then $p^{(n)} = p_0$, $n = 0,1,\ldots$. Otherwise, $p^{(0)} = p_0$, $p^{(n+1)} = \{\langle a,q^{(n)}\rangle \mid \langle a,q\rangle \in p\}$.

b. Let $p \in P \backslash P_\omega$. Then $p = \lim_i p_i$, $p_i \in P_i$, $\langle p_i\rangle_i$ a CS. We then put $p^{(n)} = \lim_i p_i^{(n)}$.

c. For $X \subseteq A \times P$ we put $X^{(n+1)} = \{\langle a,p^{(n)}\rangle \mid \langle a,p\rangle \in X\}$, $n = 0,1,\ldots$.

LEMMA 2.13.

a. *For each* p, $p = \lim_n p^{(n)}$

b. *For* $X \subseteq A \times P$, $\langle X^{(n)}\rangle_n$ *is a* CS *and* $\lim_n X^{(n)} = \bar{X}$, *where* $\bar{X}$ *is the closure of* X. *Hence, for* X *closed,* $X = \lim_n X^{(n)}$

Proof. We only prove part b. Clearly, for $m < n$, $d(X^{(n)},X^{(m)}) \leq 1/2^m$, and we see that $\langle X^{(n)}\rangle$ is a CS. We now show that $X \subseteq \lim_n X^{(n)}$. Let $\langle a,p\rangle \in X$. Then $\langle a,p\rangle = \langle a,\lim_n p^{(n)}\rangle = \lim_n \langle a,p^{(n)}\rangle \in \lim_n X^{(n)}$. Each $X^{(n)}$ is closed in P_{n+1} (*all* subsets of each P_n are closed, since distances between points are at least $1/2^n$ and so there are no non-trivial CS in P_n); hence, $\lim_n X^{(n)}$ exists and is closed. From this and $X \subseteq \lim_n X^{(n)}$ it follows that $\bar{X} \subseteq \lim_n X^{(n)}$. Conversely, let $p \in \lim_n X^{(n)}$. By theorem 2.9, $p = \lim_n p_n$, where $p_n \in X^{(n)}$, $\langle p_n \rangle_n$ a CS. Hence, $p_n = q_n^{(n)}$ for some $q_n \in X$. Then $p = \lim_n q_n$, i.e., p belongs to the closure $\bar{X}$ of X.

We now prove theorem 2.11. Similarly to what we did in the proof of lemma 2.3, we show that P satisfies (2.2) by establishing an isometry between the spaces P and $P' \overset{\text{df.}}{=} \{p_0\} \cup \mathcal{P}_c(A \times P)$. We define two bijections $\phi: P \to P'$, $\psi: P' \to P$, as follows:

(i) If $p = p_0$, then $\phi(p) = p_0$. Otherwise, $p = \lim_n p_n$, $p_n \in P_n$, $\langle p_n \rangle_n$ a CS, $p_n \neq p_0$ for n sufficiently large. For these n, by the definition of P_n we have that p_n is a subset of $A \times P_{n-1}$, hence closed in $A \times P$; thus, $\langle p_n \rangle_n$ is a CS of closed sets in $A \times P$. We now take for $\phi(p)$ the closed subset of $A \times P$ which equals $\lim_n p_n$.

(ii) If $p' = p_0$ then $\psi(p') = p_0$. Otherwise, take $p' = X \in \mathcal{P}_c(A \times P)$. By Lemma 2.13b, $X = \lim_n X^{(n)}$. For each $n > 0$, put $p_n = X^{(n)} \in P_n$. Since $\langle X^{(n)} \rangle_n$ is a CS in P', $\langle p_n \rangle_n$ is a CS in P. So we define $\psi(p') = \lim_n p_n$.

We leave it to the reader to verify that ϕ, ψ are the required isometric mappings. This concludes the proof of theorem 2.11. □

We proceed with the introduction of the operations "$\circ$", "$\cup$", "$\|$" for processes p in P solving (2.2). By the preceding theory we know that for each process p, either p is p_0, or p is finite and $p = X \in \mathcal{P}_c(A \times P)$, or p is infinite and $p = \lim_i p^{(i)}$, $\langle p^{(i)} \rangle_i$ a CS, with $p^{(i)} \in P_i$, $i = 0, 1, \ldots$.

DEFINITION 2.14. Let $X, Y \in \mathcal{P}_c(A \times P)$ with $\deg(X), \deg(Y) < \infty$.

a. (composition) $p \circ p_0 = p$, $p \circ X = \{p \circ x \mid x \in X\}$, $p \circ \langle a, q \rangle = \langle a, p \circ q \rangle$, and $p \circ \lim_i q^{(i)} = \lim_i (p \circ q^{(i)})$.

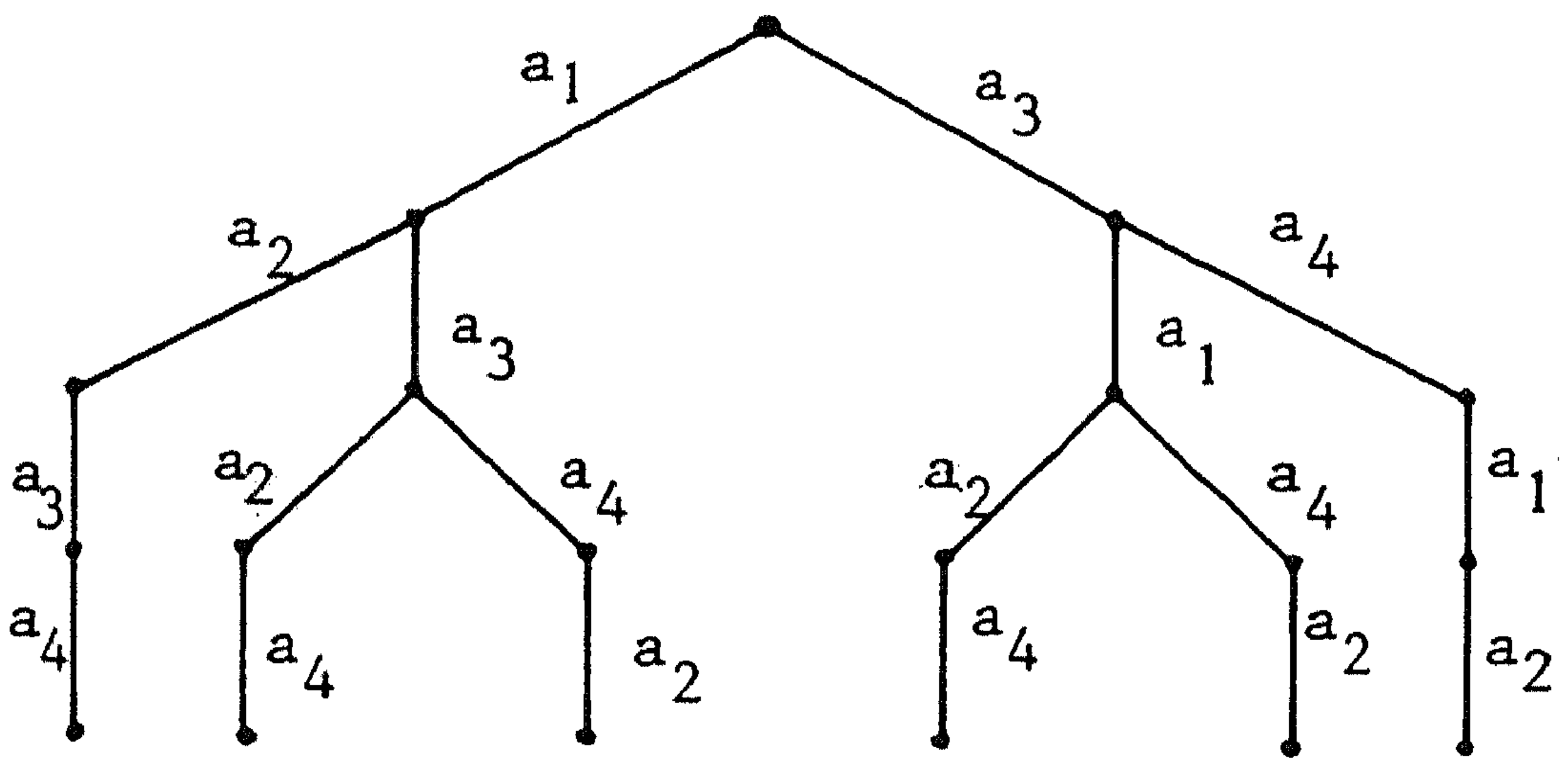
b. (union) $p_0 \cup p = p \cup p_0 = p$, $X \cup Y$ is the set-theoretic union of the two sets X,Y. Also, $(\lim_i p^{(i)}) \cup (\lim_j q^{(j)}) = \lim_k (p^{(k)} \cup q^{(k)})$.

c. (merge) $p \| p_0 = p_0 \| p = p$, $X \| Y = \{X \| y \mid y \in Y\} \cup \{x \| Y \mid x \in X\}$, $X \| \langle a, p \rangle = \langle a, X \| p \rangle$, $\langle a, p \rangle \| X = \langle a, p \| X \rangle$, and $(\lim_i p^{(i)}) \| (\lim_j q^{(j)}) = \lim_k (p^{(k)} \| q^{(k)})$.

Example. $p_1 \| p_2 \overset{\text{df.}}{=} \{<a_1,\{<a_2,p_0>\}>\} \| \{<a_3,\{<a_4,p_0>\}>\} =$
$\{<a_1,\{<a_2,p_0>\} \| p_2>\} \cup \{<a_3,p_1 \| \{<a_4,p_0>\}>\} =$
$\{<a_1,\{<a_2,p_2>\} \cup \{<a_3,\{<a_2,p_0>\} \| \{<a_4,p_0>\}>\}>,$
$<a_3,\{<a_4,p_1>\} \cup \{<a_1,\{<a_2,p_0>\} \| \{<a_4,p_0>\}>\}>\} = \ldots =$
$\{<a_1,\{<a_2,\{<a_3,\{<a_4,p_0>\}>\}>,$
$<a_3,\{<a_2,\{<a_4,p_0>\}>,<a_4,\{<a_2,p_0>\}>\}>\}>,$
$<a_3,\ldots>\}.$

(The reader should compare this with the (language-theoretic) *shuffle* of two words a_1a_2 and a_3a_4, yielding a set of six words $\{a_1a_2a_3a_4, a_1a_3a_2a_4, \ldots, a_3a_4a_1a_2\}$.)

The following picture describes the result:



Definition 2.14 is justified in

LEMMA 2.15.

a. *For finite* q,q', $d(p \circ q, p \circ q') \leq d(q,q')$

b. *For finite* q_n, *if* $<q_n>_n$ *is a* CS *then so is* $<p \circ q_n>_n$
(*Hence, the definition* $p \circ q = \lim_n (p \circ q^{(n)})$ *is well-formed*)

c. *Part* a *holds for all* q,q'

d. *If* $q_n \to q$ *then* $p \circ q_n \to p \circ q$ ("$\circ$" *is continuous in its second argument*)

e. *For finite* p,q,p',q', $d(p \cup p', q \cup q') \leq \max(d(p,q),d(p',q'))$

f. *For finite* p_n,q_n, *if* $<p_n>_n$, $<q_n>_n$ *are* CS, *then so is* $<p_n \cup q_n>_n$
(*Hence, the definition* $p \cup q = \lim_n (p^{(n)} \cup q^{(n)})$ *is well-formed*)

g. *Part* f *holds for all* p,p',q,q'

h. *If* $p_n \to p$, $q_n \to q$ *then* $p_n \cup q_n \to p \cup q$ ("$\cup$" *is continuous in both arguments*)

i. *For finite* p,q,q',p', $d(p \| q, p' \| q') \leq \max(d(p,p'),d(q,q'))$

j-ℓ. *Similarly to* f-h *for* $\|$

m. "$\circ$" *is continuous in its first argument*

n. "$\circ$","$\cup$","$\|$" *are associative,* "$\cup$" *and* "$\|$" *are commutative.*

Proof. See Appendix B. $\square$

We continue with the consideration of domain equations which determine more complex processes. Calling processes in (2.2) *uniform*, we consider the non-uniform processes defined in

$$(2.3) \qquad P = \{p_0\} \cup (A \to \mathcal{P}_c(B\times P))$$

Processes p are now (either p_0 or) *functions*, such that for each a, p(a) is a closed set $\{\ldots,\langle b_i,p_i\rangle,\ldots\}_{i\in I}$, where the index set I depends on a: I = I(a). The solution of (2.3) is very similar to the ones given above. A new element is the distance between functions. We give

DEFINITION 2.16. The collection of spaces (P_n,d_n), n = 0,1,..., is defined as follows: P_0 and d_0 are as before. $P_{n+1} = \{p_0\} \cup (A \to \mathcal{P}(B\times P_n))$, $d_{n+1}(p',p'')$ is as before for $p' = p_0$ or $p'' = p_0$. Otherwise, $d_{n+1}(p',p'') = \sup_{a\in A} d_{n+1}(p'(a),p''(a))$, where the distance between the sets p'(a),p"(a) is the usual Hausdorff distance induced by the distance between points $d_{n+1}(\langle b_1,p_1\rangle,\langle b_2,p_2\rangle)$ given by

$$d_{n+1}(\langle b_1,p_1\rangle,\langle b_2,p_2\rangle) = \begin{cases} 1, \text{ if } b_1 \neq b_2 \\ \tfrac{1}{2}\, d_n(p_1,p_2), \text{ if } b_1 = b_2. \end{cases}$$

As before, d_n determines a metric on P_n, P_ω is defined as $\cup_n P_n$, $d = \cup_n d_n$, and (P,d) is the completion of (P_ω,d). We have

THEOREM 2.17. $P = \{p_0\} \cup (A \to \mathcal{P}_c(B\times P))$.

Proof. By appropriately adapting the proof of theorem 2.11. For example, we treat the isometry $\phi: P \to P'$, where $P' \overset{\text{df.}}{=} \{p_0\} \cup (A \to \mathcal{P}_c(B\times P))$. Let $p = \lim_n p_n$, $\langle p_n\rangle_n$ a CS in P. We indicate how to obtain $\phi(p)$ as a function in $(A \to \mathcal{P}_c(B\times P))$. Take any $a \in A$. Since $\langle p_n\rangle_n$ is a CS, so is $\langle p_n(a)\rangle_n$. As CS of closed sets, $\langle p_n(a)\rangle_n$ has as limit a closed set, say X_a, where $X_a \subseteq B \times P$. Now put $\phi(p) = \lambda a.X_a$. We have to check (i)ϕ is well defined, i.e., if (p=) $\lim_n p_n = \lim_n q_n$, then $\lim_n p_n(a) = \lim_n q_n(a)$, (ii) ϕ is 1-1, i.e., $\phi(p) = \phi(q) \Rightarrow p = q$, (iii) ϕ is onto, and (iv) ϕ preserves distances. We treat only (ii). Assume that, for all a, $\lim_n p_n(a) = \lim_n q_n(a)$. To show $p = q$, i.e., $\lim_n p_n = \lim_n q_n$. Since $\langle p_n \rangle_n$, $\langle q_n \rangle_n$ are CS, we have $\forall\varepsilon\exists N\forall m,n \geq N\ [d(p_m,p_n) < \varepsilon/2,\ d(q_m,q_n) < \varepsilon/2]$. Thus, $(*)$ $\forall m,n \geq N$ $\forall a[d(p_m(a),p_n(a)) < \varepsilon/2]$, $(**)$ $\forall m,n \geq N\ \forall a[d(q_m(a),q_n(a)) < \varepsilon/2]$. Letting $m \to \infty$ in $(*)$, $(**)$ we have $p_m(a) \to p(a)$, $q_m(a) \to q(a)$. Thus $\forall n \geq N\ \forall a[d(p_n(a),p(a)) \leq \varepsilon/2,\ d(q_n(a), q(a)) \leq \varepsilon/2]$. From this, since $p(a) = q(a)$, we obtain $\forall n \geq N\ [d(p_n(a),q_n(a)) \leq \varepsilon]$. Taking sup over all a we get $\forall n \geq N\ [d(p_n,q_n) \leq \varepsilon]$. By a standard argument then $d(p,q) \leq \varepsilon$. Since this holds for any ε we conclude that $p = q$. □

The operations "$\circ$","$\cup$","$\|$" can be extended to non-uniform processes.

DEFINITION 2.18. We only consider processes of finite nonzero degree, the treatment of the remaining cases being the usual one.

a. (composition) $p \circ \lambda a.X = \lambda a.(p \circ X)$, where $p \circ X = \{p \circ x \mid x \in X\}$, and $p \circ \langle b,q\rangle = \langle b,p \circ q\rangle$

b. (union) $(\lambda a.X) \cup (\lambda a.Y) = \lambda a.(X \cup Y)$

c. (merge) $(\lambda a.X) \| (\lambda a.Y) = \lambda a.(\{x \| (\lambda a.Y) \mid x \in X\} \cup \{(\lambda a.X) \| y \mid y \in Y\})$ where $\langle b,p\rangle \| (\lambda a.Y) = \langle b,p \| \lambda a.Y\rangle$, and $(\lambda a.X) \| \langle b,q\rangle = \langle b,(\lambda a.X) \| q\rangle$

Remark. Observe the difference between clauses b and c, in that we do *not* put $(\lambda a.X) \| (\lambda a.Y) = \lambda a.(X \| Y)$ (with $X \| Y$ defined appropriately). In other words, though we have, for $p,q \neq p_0$, that $p \cup q = \lambda a.(p(a) \cup q(a))$, for $p \| q$ we do not have $p \| q = \lambda a.(p(a) \| q(a))$ but, instead, $p \| q = \lambda a.((p(a) \| q) \cup (p \| q(a)))$.

Operations "$\circ$","$\cup$" and "$\|$" for non-uniform processes satisfy the natural extension of Lemma 2.15:

LEMMA 2.19. *As* Lemma 2.15, *but now for the operations as given in* definition 2.18.

Proof. Left to the reader.

The last equation in the list of domain equations is

$$(2.4) \qquad P = \{p_0\} \cup (A \to P_c((B \times P) \cup (C \to P))).$$

We only give the definition of the metric spaces (P_n,d_n), leaving elaboration of the details concerning the isometries necessary to establish (2.4) to the reader. We have

DEFINITION 2.20. The metric spaces (P_n, d_n), $n = 0,1,\ldots$, are defined by: P_0, d_0 are as before, $P_{n+1} = \{p_0\} \cup (A \to \mathcal{P}((B \times P_n) \cup (C \to P_n)))$, $d_{n+1}(p', p'')$ is as before for $p' = p_0$ or $p'' = p_0$. Otherwise, $d_{n+1}(p', p'') = \sup_{a \in A} d_{n+1}(p'(a), p''(a))$, where $d_{n+1}(X,Y)$ is the Hausdorff distance between sets induced by the distance between points $d_{n+1}(x,y)$, where $d_{n+1}(\langle b,p\rangle, \lambda c.p') = 1 = d_{n+1}(\lambda c.p', \langle b,p\rangle)$, $d_{n+1}(\langle b_1,p_1\rangle, \langle b_2,p_2\rangle)$ is as usual, and $d_{n+1}(\lambda c.p_1, \lambda c.p_2) = \sup_{c \in C} d_n(p_1, p_2)$.

The operations for $p \in P$, with P solving (2.4) are given in

DEFINITION 2.21. We only consider processes of finite nonzero degree.

a. $p \circ \lambda a.X = \lambda a.(p \circ X)$, $p \circ X = \{p \circ x \mid x \in X\}$, $p \circ \langle b,q\rangle = \langle b, p \circ q\rangle$, $p \circ \lambda c.p' = \lambda c.(p \circ p')$

b. $\cup$: Omitted.

c. $(\lambda a.X) \| (\lambda a.Y) = \lambda a.(\{x \| (\lambda a.Y) \mid x \in X\} \cup \{(\lambda a.X) \| y \mid y \in Y\})$, where $\langle b,p\rangle \| \lambda a.Y = \langle b, p \| \lambda a.Y\rangle$ and similarly for $(\lambda a.X) \| \langle b,p\rangle$, $(\lambda c.p') \| (\lambda a.Y) = \lambda c.(p' \| \lambda a.Y)$, and similarly for $(\lambda a.X) \| (\lambda c.p')$.

As the last lemma of this section we claim

LEMMA 2.22. *The operations* "$\circ$","$\cup$","$\|$" *have the usual properties.*

Proof. Omitted. $\square$

Having arrived at the end of this section, we summarize the main results:

1. Process domains P are obtained as solutions of equations of the form

   a. $P = \{p_0\} \cup (A \times P)$
   b. $P = \{p_0\} \cup \mathcal{P}_c(A \times P)$, where $\mathcal{P}_c(\cdot)$ stands for all closed subsets of $(\cdot)$
   c. $P = \{p_0\} \cup (A \to \mathcal{P}_c(B \times P))$ (idem)
   d. $P = \{p_0\} \cup (A \to \mathcal{P}_c((B \times P) \cup (C \to P)))$ (idem)

2. Processes p are either nil (p_0), or finite and of finite degree *deg*(p), or infinite and (topological) limit of a sequence $\langle p^{(i)} \rangle_i$ with $p^{(i)}$ finite. (For the definitions of the $p^{(i)}$ see point 5 below.)

3. Operations upon processes are composition ("$\circ$"), union ("$\cup$") and merge ("$\|$"). They are defined as follows ($\cup$, $\|$ only for process domains solving b,c,d above; X,Y are always finite elements of $\mathcal{P}_c(\cdot)$):

3.1. $p \circ q$ is defined by induction on $deg(q)$:

$p \circ p_0 = p$, $p \circ X = \{p \circ x \mid x \in X\}$, $p \circ \langle a, q\rangle = \langle a, p \circ q\rangle$, $p \circ \lambda a.X = \lambda a.(p \circ X)$,
$p \circ \langle b, q\rangle = \langle b, p \circ q\rangle$, $p \circ \lambda c.q = \lambda c.p \circ q$, $p \circ \lim_i q^{(i)} = \lim_i (p \circ q^{(i)})$

3.2. $p \cup q$ is defined by

$p \cup p_0 = p_0 \cup p = p$, $X \cup Y$ is the set-theoretic union of X and Y,
$(\lambda a.X) \cup (\lambda a.X) = \lambda a.(X \cup Y)$, $(\lim_i p^{(i)}) \cup (\lim_j q^{(j)}) = \lim_k (p^{(k)} \cup q^{(k)})$

3.3. $p \| q$ is defined by induction on $deg(p) + deg(q)$:

$p \| p_0 = p_0 \| p = p$, $X \| Y = \{x \| Y \mid x \in X\} \cup \{X \| y \mid y \in Y\}$,
$(\lambda a.X) \| (\lambda a.Y) = \lambda a.(\{x \| \lambda a.Y \mid x \in X\} \cup \{\lambda a.X \| y \mid y \in Y\})$,
$\langle a, p\rangle \| Y = \langle a, p \| Y\rangle$, $Y \| \langle a, p\rangle = \langle a, Y \| p\rangle$,
$\langle b, p\rangle \| (\lambda a.Y) = \langle b, p \| \lambda a.Y\rangle$, and similarly for $(\lambda a.Y) \| \langle b, p\rangle$
$(\lambda c.q) \| (\lambda a.Y) = \lambda c.(q \| (\lambda a.Y))$, and similarly for $(\lambda a.Y) \| (\lambda c.q)$,
$(\lim_i p^{(i)}) \| (\lim_j q^{(j)}) = \lim_k (p^{(k)} \| q^{(k)})$.

4. The above operations are continuous and satisfy the usual properties such as commutativity ($\cup, \|$), associativity ($\circ, \cup, \|$), etc.

5. With respect to each of the equations a to d, $p_0^{(n)} = p_0$, $n = 0, 1, \ldots$, and, for $p \neq p_0$, $p^{(0)} = p_0$.
Moreover, for $n = 0, 1, \ldots$,
(For a) $p^{(n+1)} = \langle a, q^{(n)}\rangle$, where $p = \langle a, q\rangle$

(For b) $p^{(n+1)} = \{\langle a, q^{(n)}\rangle \mid \langle a, q\rangle \in p\}$
(For c) $p^{(n+1)} = \lambda a.\{\langle b, q^{(n)}\rangle \mid \langle b, q\rangle \in p(a)\}$
(For d) $p^{(n+1)} = \lambda a.(\{\langle b, q^{(n)}\rangle \mid \langle b, q\rangle \in p(a)\} \cup \{\lambda c.q^{(n)} \mid \lambda c.q \in p(a)\})$.

3. FLOW OF CONTROL: MERGE WITH ITERATION OR RECURSION

In this section we introduce the first two of the series of languages studied in sections 3-8. Both languages have elementary actions, sequential composition, nondeterministic choice and (arbitrary, i.e. not synchronized) merge. Language L_0 has moreover iteration (*), and language L_1 has recursion. We shall use $\underline{A}$, with typical elements $\underline{a}$, for the class of elementary (atomic) actions. In later refinements of the theory, actions $\underline{a}$ will be replaced by assignment statements. Throughout the paper, we use a self-explanatory variant of BNF for syntactic definitions.

DEFINITION 3.1. The language L_0 (regular flow of control + merge) with elements S, is defined by

$$S ::= \underline{a} \mid \underline{skip} \mid S_1;S_2 \mid S_1 \cup S_2 \mid S_1 \| S_2 \mid S^*.$$

For the definition of the semantics of L_0 we use a domain of uniform processes P_0. We assume that its constituent set A is a (possibly infinite) alphabet such that for each elementary action $\underline{a} \in \underline{A}$ there is a corresponding $a \in A$. Let, moreover, ε be the empty word (with respect to the alphabet A). We give

DEFINITION 3.2. The domain P_0 is given as solution of

$$P_0 = \{p_0\} \cup P_c((A \cup \{\varepsilon\}) \times P_0).$$

Remark. Properly speaking, this requires adaptation of the definitions of section 2 for uniform processes with the convention that $a \in A \cup \{\varepsilon\}$, together with natural definitions such as: $a_1 = a_2$ if a_1 and a_2 are both ε, or denote the same element of A.

We now define the semantics of L_0 by providing a mapping $M: L_0 \to P_0$. Thus, M determines for each language element S a corresponding process p. (Mappings such as M are often called *valuations* in denotational semantics. They serve to associate *meaning* - mathematical objects - to the syntactic constructs in a certain class (here L_0), and in this way embody the heart of a denotational semantics definition.)

DEFINITION 3.3. The valuation $M: L_0 \to P_0$ is defined by

a. $M(\underline{a}) = \{\langle a,p_0\rangle\}$, where a corresponds to $\underline{a}$, $M(\underline{skip}) = \{\langle\varepsilon,p_0\rangle\}$

b. $M(S_1;S_2) = M(S_2) \circ M(S_1)$, $M(S_1 \cup S_2) = M(S_1) \cup M(S_2)$, $M(S_1 \| S_2) = M(S_1) \| M(S_2)$

c. $M(S^*) = \lim_i p_i$, where ($p_0 = p_0$ and)

$$p_{i+1} = (p_i \circ M(S)) \cup \{\langle\varepsilon,p_0\rangle\}.$$

Remarks.

1. Since the elementary actions are left unspecified, there is not much we can do with them in the semantic definition. Therefore, we simply map them onto some corresponding elementary process.
2. The simplicity of clause b is a reward of our preparatory work in section 2. Operations upon (uniform) processes "$\circ$","$\cup$","$\|$" have become available,

and they can be used directly to model the corresponding syntactic composition rules.

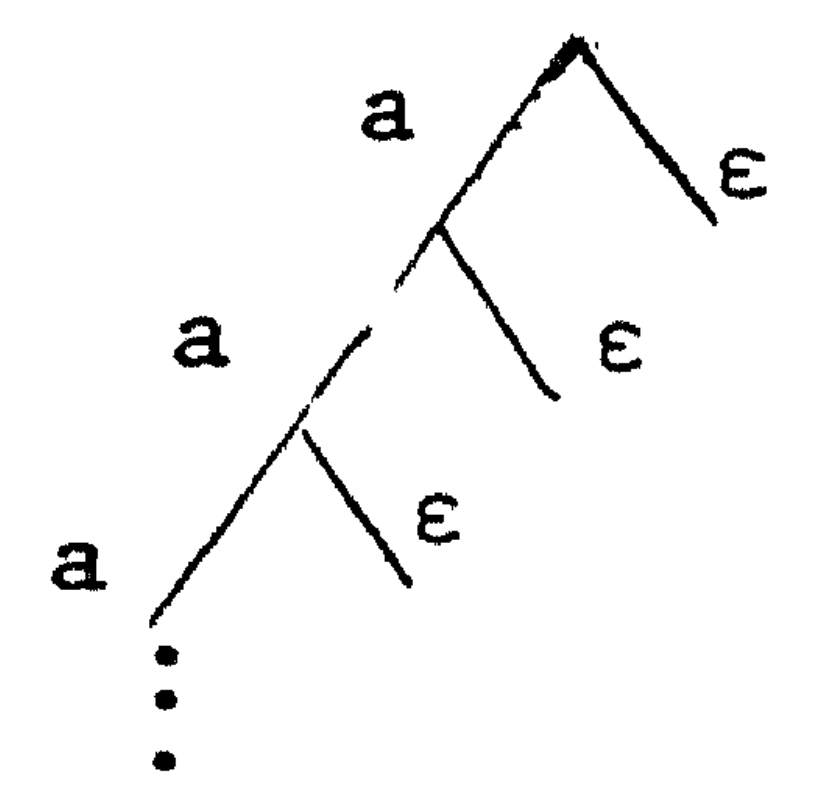
3. In order to understand the definition of S^*, recall the equivalence $S^* = S;S^* \cup \underline{\text{skip}}$. Now define a mapping $T\colon P_0 \to P_0$ by putting $T = \lambda p.((p \circ M(S)) \cup \{\langle\varepsilon,p_0\rangle\})$. Here $\{\langle\varepsilon,p_0\rangle\}$ is the dummy process, i.e., the semantic equivalent of the syntactic $\underline{\text{skip}}$ action. It follows from general properties of the operations "$\circ$","$\cup$" (see Appendix B) that the mapping T is *contracting*, viz. that, for all p',p", $d(T(p'),T(p'')) \le \frac{1}{2}d(p',p'')$ (this uses that $M(S) \neq p_0$ for all S). By a classical result in metric topology (the Banach fixed point theorem) we may then conclude that the sequence $p_0, T(p_0), T^2(p_0), \ldots$ is a Cauchy sequence which converges to a limit p satisfying $p = T(p)$. (In fact, this limit is independent of the starting process p_0, and yields the *unique* fixed point of T.)

Examples

1. $M(\underline{a}_1;\underline{a}_2) = M(\underline{a}_2) \circ M(\underline{a}_1) = \{\langle a_2,p_0\rangle\} \circ \{\langle a_1,p_0\rangle\} = \{\langle a_1,\{\langle a_2,p_0\rangle\}\rangle\}$.
2. $M((\underline{a}_1;\underline{a}_2) \parallel (\underline{a}_3;\underline{a}_4)) = \{\langle a_1,\{\langle a_2,p_0\rangle\}\rangle\} \parallel \{\langle a_3,\{\langle a_4,p_0\rangle\}\rangle\} = \ldots =$
   $\{\langle a_1,\{\langle a_2,\{\langle a_3,\{\langle a_4,p_0\rangle\}\rangle\}\rangle,$
   $\langle a_3,\{\langle a_2,\{\langle a_4,p_0\rangle\}\rangle,\langle a_4,\{\langle a_2,p_0\rangle\}\rangle\}\rangle\}\rangle,$
   $\langle a_3,\ldots\rangle\}$
   (Cf. the example after definition 2.14).
3. $M(\underline{a}^*) = p = \lim_i p_i$, where $p_{i+1} = (p_i \circ \{\langle a,p_0\rangle\}) \cup \{\langle\varepsilon,p_0\rangle\}$.

   Hence, $p = \{\langle\varepsilon,p_0\rangle,$
   $\langle a,\{\langle\varepsilon,p_0\rangle,$
   $\langle a,\{\langle\varepsilon,p_0\rangle,$
   $\langle a,\ldots$

   In a picture, $M(\underline{a}^*)$ is described by



We observe that $\underline{a}^*$ means executing $\underline{a}$ zero or more times, including infinite repetition of $\underline{a}$.

We next turn to the recursive case. We shall employ the notation of the μ-calculus for recursion (see, e.g. [10,32]). For the reader who has not seen this before, the following explanation may help: Think of a parameterless recursive procedure Q in some Algol-like language. Q has a declaration of the form, say, Q ⇐ ...Q...Q..., where ...Q...Q... is the procedure body with two recursive calls of Q. We note that the procedure variable Q is *bound* in this declaration (systematically renaming it would make no difference). A call of Q in the main program corresponds in the notation of the μ-calculus to the statement $\mu\xi[\ldots\xi\ldots\xi\ldots]$, where the bound variable ξ is from some alphabet of procedure variables X. In this way, procedure declarations disappear, and inner calls are taken care of by the bound variable mechanism.

DEFINITION 3.4. Let X, with elements ξ, be the set of procedure variables. The language L_1 (general recursion with merge) is defined by: Let $S \in L_1$. Then

$$S ::= \underline{a} \mid \underline{skip} \mid S_1 ; S_2 \mid S_1 \cup S_2 \mid S_1 \| S_2 \mid \xi \mid \mu\xi[S].$$

For the semantics of L_1 we take process domain P_1 equal to P_0. In order to handle the variables ξ, we introduce an environment E, with elements η, defined by $E = X \to P_1$, and we define the meaning of a statement $S \in L_1$ with respect to E. In other words, we take $M: L_1 \to (E \to P_1)$; its definition is given in

DEFINITION 3.5.

a. $M(\underline{a})(\eta) = \{\langle a, p_0\rangle\}$, $M(\underline{skip})(\eta) = \{\langle \varepsilon, p_0\rangle\}$

b. $M(S_1;S_2)(\eta) = M(S_2)(\eta) \circ M(S_1)(\eta)$

$M(S_1 \cup S_2)(\eta) = M(S_1)(\eta) \cup M(S_2)(\eta)$

$M(S_1 \| S_2)(\eta) = M(S_1)(\eta) \| M(S_2)(\eta)$

c. $M(\xi)(\eta) = \eta(\xi)$

$M(\mu\xi[S])(\eta) = \lim_i p_i$, where ($p_0 = p_0$ and)

$p_{i+1} = \{\langle \varepsilon, M(S)(\eta\{p_i/\xi\})\rangle\}$.

Remarks.

1. Clauses a and b are exactly as in definition 3.3, apart from the extra argument η which is just carried along.
2. In the definition of the meaning of the μ-construct we observe a complication. The reader who is familiar with the treatment of (sequential) recursive procedures in denotational semantics would probably have expected the definition $p_{i+1} = M(S)(\eta\{p_i/\xi\})$. (Note that this specializes

to the previous treatment of iteration by taking $S^* \equiv \mu\xi[S;\xi \cup \underline{skip}]$.) This may work as well, but we have not been able to prove that, defining the mapping $T' = \lambda p.\mathcal{M}(S)(\eta\{p/\xi\})$, the sequence $\langle T'^i(p_0)\rangle_i$ is a CS for arbitrary $S \in \mathcal{L}_1$. (Bergstra & Klop [13] prove that $\langle T'^i(q)\rangle_i$ is a CS for each q. However, the resulting limit depends, in general, on q, and the problem remains which q to choose.) Therefore, we have introduced an extra step in defining $T = \lambda p.\{\langle\varepsilon,\mathcal{M}(S)(\eta\{p/\xi\})\rangle\}$. This indeed ensures that T is contracting and, as before, $\lim_i T^i(p_0)$ exists and equals the unique fixed point of T. Operationally, the ε-step may be seen as reflecting the action of procedure entry. By way of example we obtain that $\mathcal{M}(\mu\xi[\xi])(\eta) = \{\langle\varepsilon,\{\langle\varepsilon,\{\langle\varepsilon,\ldots\rangle\}\rangle\}\rangle\}$ (an infinite sequence of empty steps) C.f. also the discussion in [17].

In definitions 3.3 and 3.5 we have shown how to associate a process p with statements $S \in \mathcal{L}_0$ or $S \in \mathcal{L}_1$. In case one is interested only in the set of all possible sequences of elementary actions determined by executing S - rather than in its meaning $p = \mathcal{M}(S)$ as a whole; note that a process contains more information than the set of its constituent paths - we apply a new (unary) operation upon process p, determining its *yield* p^+. For this, we need the auxiliary definition of *path* of a process:

DEFINITION 3.6. Let $p \in P_0$, and let $a, a_i \in A \cup \{\varepsilon\}$. A *path* for p is a (finite or infinite) sequence (*): $\langle a_1,p_1\rangle,\langle a_2,p_2\rangle,\ldots,\langle a_i,p_i\rangle,\ldots$ such that

(i) $\langle a_1,p_1\rangle \in p$ and $\langle a_{i+1},p_{i+1}\rangle \in p_i$, $i = 1,2,\ldots$,

(ii) sequence (*) is either infinite or, when finite, terminates with $\langle a_n,p_n\rangle$ $(n\geq 1)$, with $p_n = p_0$.

Remark. Note that, by this definition, p_0 has no paths. Moreover, note that we do not allow a finite path terminating in $\langle a_n,p_n\rangle$ with $p_n = \emptyset$ (the empty set is also a process!)

Now let $A^\infty \overset{df.}{=} A^* \cup A^\omega$, i.e., A^∞ is the set of all finite (possibly empty) and infinite sequences of elements in A. Also, let "$\cdot$" denote concatenation of words over A. We put

DEFINITION 3.7. $p^+ \subseteq A^\infty$ is defined to consist of all words $w \in A^\omega$ such that either $w = a_1 \cdot a_2 \cdot \ldots \cdot a_n$, where $\langle a_1,p_1\rangle,\langle a_2,p_2\rangle,\ldots,\langle a_n,p_n\rangle$ is a finite path for p, or $w = a_1 \cdot a_2 \cdot \ldots \cdot a_i \cdot \ldots$, where $\langle a_1,p_1\rangle,\langle a_2,p_2\rangle,\ldots,\langle a_i,p_i\rangle,\ldots$ is an infinite path for p.

Remark. Remember that $a_i \in A \cup \{\varepsilon\}$. Thus, the a_i occurring in the above equations for w may disappear in the resulting concatenation in case $a_i = \varepsilon$.

Examples. $p_0^+ = \emptyset$, $\{<\varepsilon,p_0>\}^+ = \{\varepsilon\}$, $\{<a_1,\{<\varepsilon,\{<a_2,p_0>\}>\}>\}^+ = \{a_1 \cdot \varepsilon \cdot a_2\} = \{a_1a_2\}$, $\{<a_1,\{<a_2,p_0>\}>,<a_1,\{<a_3,p_0>\}>\}^+ = \{<a_1,\{<a_2,p_0>,<a_3,p_0>\}>\}^+ = \{a_1a_2,a_1a_3\}$.

From the last example we conclude that, for $p_1 \neq p_2$, we may have that $p_1^+ = p_2^+$. We may define $p_1 \sim p_2 \overset{\text{df.}}{\Longleftrightarrow} p_1^+ = p_2^+$, and study properties of this equivalence relation. (A more refined equivalence relation is Milner's observation equivalence, cf.[44].)

Finally, one may use the yield operation in the semantics of languages such as L_0 or L_1, by investigating the mapping M^+ defined by $M^+(S) = M(S)^+$. This mapping obtains the sequences of elementary actions prescribed by the execution of S . For example, $M^+((S_1;S_2) \cup (S_1;S_3)) = M^+(S_1;(S_2 \cup S_3))$, whereas M differs on these two arguments. For languages such as L_0,L_1, consideration of the yield $M(S)^+$ is probably not very fruitful. Later (section 5) we shall encounter languages where the role of the yield operation is more important.

4. SYNCHRONIZATION

We add a synchronization construct to the language L_0 - leaving to the reader a similar extension of L_1. This section owes much to the pioneering studies of Milner on the nature of synchronization [40,42,43,44].

We introduce the language L_2 as an extension of L_0 by adding a class of synchronization commands $C,\bar{C}$. Synchronization commands always appear in pairs such that $\bar{C}$ corresponds to C (and C to $\bar{C}$). Before trying to explain their meaning, we first give the syntax for L_2.

DEFINITION 4.1. The language L_2, with elements S, is defined by

$$S ::= \underline{a} \mid \underline{\text{skip}} \mid C \mid \bar{C} \mid S_1;S_2 \mid S_1 \cup S_2 \mid S_1 \| S_2 \mid S^* \mid S \backslash C.$$

In order to obtain some understanding for the meaning of these synchronization commands, let us take $S' \equiv S_1;C;S_2$, $S'' \equiv S_3;\bar{C};S_4$, and let us consider $(S' \| S'')\backslash C$. Its intended meaning is that the merge of S' and S" is synchronized by the pair $C,\bar{C}$ such that, instead of the full merge of

$S_1;S_2$ with $S_3;S_4$, we only retain $(S_1 \| S_3);\ (S_2 \| S_4)$. The role of the *restriction* operation $S\backslash C$ may be phrased roughly as deleting from the execution of S all execution sequences which contain C and $\bar{C}$ in a way where synchronization failed. In an example such as $S' \equiv \underline{a}_1;C;\underline{a}_2$, $S'' \equiv \underline{a}_3;\bar{C};\underline{a}_4$, one such failing sequence is, e.g., $\underline{a}_1;C;\underline{a}_2;\underline{a}_3;\bar{C};\underline{a}_4$.

For the definition of the semantics of L_2 we introduce the domain P_2 as given by the equation

$$(4.1) \qquad P_2 = \{p_0\} \cup P_c((A \cup \Gamma \cup \{\varepsilon\}) \times P_2)$$

Here, as before, for each $\underline{a} \in L_2$ there is a corresponding $a \in A$. Moreover, for $C,\bar{C} \in L_2$ there are corresponding elements $\gamma,\bar{\gamma}$ in Γ. An arbitrary element of the set $A \cup \Gamma \cup \{\varepsilon\}$ will in the sequel be denoted by β.

Remark. Processes in P_2 are close to Milner's synchronization trees. An important difference, however, is our use of sets rather than multisets, for the collection of "successors" of the "nodes" in a process.

We now give

DEFINITION 4.2. The valuation $M\colon L_2 \to P_2$ is given by

a. $M(\underline{a}) = \{<a,p_0>\}$, where a corresponds to $\underline{a}$
   $M(C) = \{<\gamma,p_0>\}$, where γ corresponds to C
   $M(\bar{C}) = \{<\bar{\gamma},p_0>\}$, where $\bar{\gamma}$ corresponds to $\bar{C}$
b. $M(S_1;S_2) = M(S_2) \circ M(S_1)$
   $M(S_1 \cup S_2) = M(S_1) \cup M(S_2)$
   $M(S_1 \| S_2) = M(S_1) \| M(S_2)$; for "$\|$" see def. 4.3
   $M(S\backslash C) = M(S)\backslash\gamma$; for "\" see def. 4.3
c. $M(S^*) = \lim_i p_i$, where ($p_0 = p_0$ and)
   $p_{i+1} = (p_i \circ M(S)) \cup \{<\varepsilon,p_0>\}$.

This definition assumes a refined definition of the merge operation "$\|$" between processes, and a (new) definition of $p\backslash\gamma$. These are provided in

DEFINITION 4.3.

a. Let $p_1 \|_u p_2$ be a notation for the merge of two uniform processes - over the set $A \cup \Gamma \cup \{\varepsilon\}$ - as defined in section 2. We define $p_1 \| p_2$ - for p_1,p_2 of finite nonzero degree - by: $X \| Y = (X \|_u Y) \cup \{<\varepsilon, p' \| p''> \mid <\gamma,p'> \in X,\ <\bar{\gamma},p''> \in Y$, for corresponding γ,γ and arbitrary $p',p''\}$.

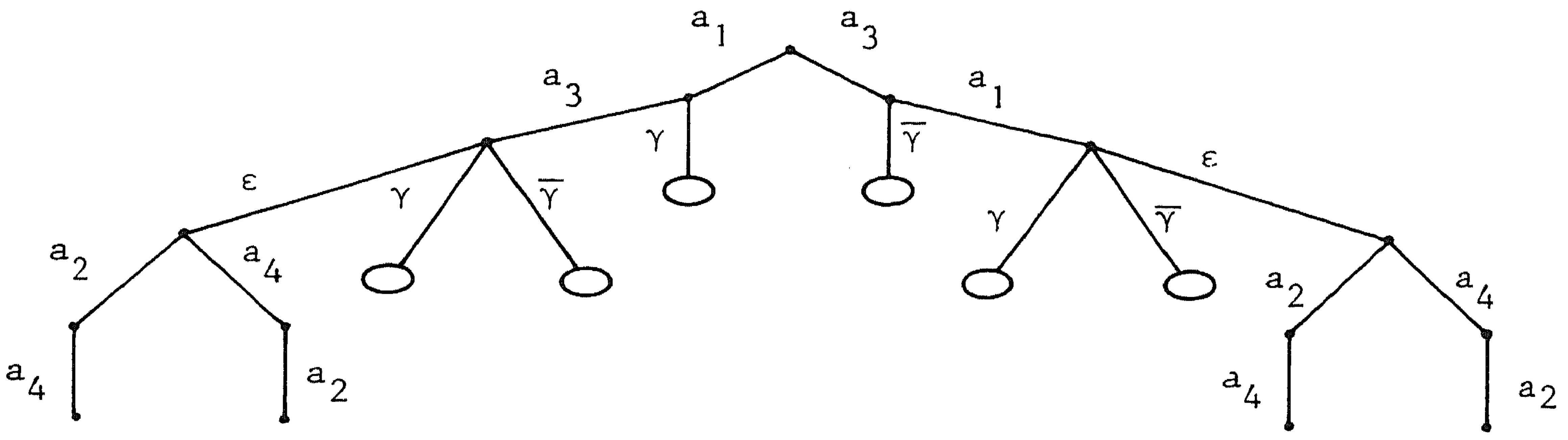
b. $p_0\backslash\gamma = p_0$

$X\backslash\gamma = \{<\beta,p'\backslash\gamma> \mid <\beta,p'> \in X,\ \beta \neq \gamma,\bar{\gamma}\}, \quad \deg(X) < \infty$

$(\lim_i\, p^{(i)})\backslash\gamma = \lim_i\, (p^{(i)}\backslash\gamma)$.

Note how the restriction operation $p\backslash\gamma$ deletes from process p all pairs $<\gamma,p'>$ or $<\bar{\gamma},p'>$ which are element of p or one of its subprocesses.

As an example of definition 4.2, 4.3 we consider the programs $S' \| S''$ and $(S' \| S'')\backslash C$, where $S' \equiv \underline{a}_1;C;\underline{a}_2$ and $S'' \equiv \underline{a}_3;\bar{C};\underline{a}_4$. We obtain for $M(S_1 \| S_2)$ the process depicted in



Here the leaves marked by ◯ contain trees which disappear as the result of the \C operation. Thus, all failing attempts at synchronization are deleted, and the result only contains a_i -steps with two ε-steps interspersed.

We conclude with a few words on the yield operation in this case. For $p \in P_2$, p^+ determines a set of (finite or infinite) paths over the alphabet $A \cup \Gamma$. In case $p^+ \subseteq A^\infty$, one might call p *proper*. E.g., for p such that $p = M(S\backslash C)$, where synchronization in S only uses $C,\bar{C}$, we expect that $p^+ \subseteq A^\infty$. This expresses that unsuccesful attempts at synchronization do not contribute to p^+, since there is no contribution to p^+ from paths in p terminating in the empty process (cf. the remark following definition 3.6).

5. STATES AND ASSIGNMENT

Until now, our languages contained only elementary actions the meaning of which was left unspecified. We next introduce the notion of state, extend the syntax of our languages with assignment and tests, and discuss the corresponding extension for the processes used in their semantics. First we present some preliminary definitions, introducing simple expressions, tests, and their meanings with respect to some state.

DEFINITION 5.1.

a. Let *Var*, with elements $\underline{x},\underline{y},\ldots$ be the class of simple variables. Let V be some domain of values ($\mathbb{Z}$ might be an example) and let $\Sigma \overset{\text{df.}}{=} \mathit{Var} \to V$. Let $W = \{tt, ff\}$ be the set of truth-values.

b. Let $v \in V$, $\sigma \in \Sigma$, $\underline{x} \in \mathit{Var}$. We define the *variant* notation, turning state σ into a state $\sigma\{v/\underline{x}\}$, by putting

$$\sigma\{v/\underline{x}\}(\underline{y}) = \begin{cases} v, \text{ if } \underline{x} \equiv \underline{y} \\ \sigma(\underline{y}) \text{ if } \underline{x} \not\equiv \underline{y}. \end{cases}$$

c. We introduce the classes *Exp*, with elements s,t, of *expressions* and *Test*, with elements b, of *logical expressions*. We assume given valuations $\mathcal{V}\colon \mathit{Exp} \to (\Sigma \to V)$ and $\mathcal{W}\colon \mathit{Test} \to (\Sigma \to W)$.
(The precise nature of *Exp* and *Test* does not concern us here; all we require is that their evaluation always terminates. In a specific instance, taking, e.g., $\mathbb{Z}$ for V, one might think of expressions such as $x+(y*z)$, and tests such as $x > y+z$.)

We continue with the definition of the syntax of language L_3. It extends L_0 with assignment and tests. Synchronization will reappear in section 6 (this postponement is only for reasons of presentation).

DEFINITION 5.2. The language L_3, with elements S, is defined by

$$S ::= \underline{x} := s \mid \underline{\text{skip}} \mid b \mid S_1;S_2 \mid S_1 \cup S_2 \mid S_1 \| S_2 \mid S^* \mid \underline{x} := ?$$

Remarks.

1. The intuitive meaning of $\underline{x} := s$, $\underline{\text{skip}}$, $S_1;S_2$, $S_1 \cup S_2$, $S_1 \| S_2$, S^* should be clear.
2. A test statement b may succeed or fail, depending on whether the test b evaluates to tt or ff in the current state. More familiar constructions such as $\underline{\text{if}}$ b $\underline{\text{then}}$ S_1 $\underline{\text{else}}$ S_2 $\underline{\text{fi}}$ or $\underline{\text{while}}$ b $\underline{\text{do}}$ S $\underline{\text{od}}$ are expressed in L_3 by $(b;S_1)\cup(\neg b;S_2)$ or $(b;S)^*;\neg b$, respectively.
3. $\underline{x} := ?$ is the *random assignment*, introduced not so much because it is our favorite language concept, but rather to illustrate that semantics using processes can deal with it without any technical problems (contrary to the situation in traditional denotational semantics, cf. [4,7,10,20]).

For the semantics of L_3 we introduce the class of processes P_3. This involves an essential extension of the processes as considered upto now, in that a process $p(\neq p_0)$ is now a function depending on Σ.

<u>DEFINITION 5.3</u>. The class of process P_3 is defined as solution of the domain equation

$$P_3 \equiv \{p_0\} \cup (\Sigma \to P_c(\Sigma \times P_3)). \tag{5.1}$$

We observe that (5.1) is an equation for a domain of non-uniform processes of the type considered in section 2, equation (2.3). By the general theory as developed there, operations $p_1 \circ p_2$, $p_1 \cup p_2$, $p_1 \| p_2$ are again meaningful (the latter, for the moment, without the synchronization refinement).

We define the valuation $M : L_3 \to P_3$ in

<u>DEFINITION 5.4</u>. The semantics of L_3 is given by

a. $M(\underline{x}{:=}s) = \lambda\sigma.\{<\sigma\{V(s)(\sigma)/\underline{x}\ ,p_0>\}$, $M(\underline{\text{skip}}) = \lambda\sigma.\{<\sigma,p_0>\}$
b. $M(b) = \lambda\sigma.$ $\underline{\text{if}}$ $W(b)(\sigma)$ $\underline{\text{then}}$ $\{<\sigma,p_0>\}$ $\underline{\text{else}}$ $\emptyset$ $\underline{\text{fi}}$
c. $M(S_1;S_2) = M(S_2) \circ M(S_1)$, $M(S_1 \cup S_2) = M(S_1) \cup M(S_2)$, $M(S_1 \| S_2) = M(S_1) \| M(S_2)$.
d. $M(S^*) = \lim_i p_i$, where ($p_0 = p_0$ and)
$p_{i+1} = (p_i \circ M(S)) \cup \lambda\sigma.\{<\sigma,p_0>\}$
e. $M(\underline{x}{:=}?) = \lambda\sigma.\{<\sigma\{v/\underline{x}\},p_0> \mid v \in V\}$.

Remarks.

1. Note how the dummy process, previously represented by $\{<\varepsilon,p_0>\}$, is now replaced by $\lambda\sigma.\{<\sigma,p_0>\}$.
2. Note that, in clause e, the set $X = \{<\sigma\{v/\underline{x}\},p_0> \mid v \in V\}$ is a subset of $P_{3,1} \overset{\text{df.}}{=} \{p_0\} \cup (\Sigma \to P(\Sigma \times \{p_0\}))$; that X is closed requires no more argument than the observation that *all* subsets of each of $P_{3,k}$ (where $P_{3,0} = \{p_0\}$, $P_{3,k+1} = \{p_0\} \cup (\Sigma \to P_c(\Sigma \times P_{3,k}))$, for $k \geq 0$) are (trivially) closed: distances between points are at least $1/2^{k-1}$ (for $k \geq 1$), and no nontrivial CS exists in $P_{3,k}$. Thus, we see how unbounded nondeterminacy fits smoothly into our theory. It should be remarked, however, that the continuity problems caused by unbounded nondeterminacy in classical denotational semantics are now transferred to the same problem for the yield function (to be defined in definition 5.6).

Examples.

1. $M(\underline{x}:=0;\ \underline{y}:=\underline{x}+1) = \lambda\sigma.\{\langle\sigma\{0/\underline{x}\},\lambda\bar{\sigma}.\{\langle\bar{\sigma}\{V(\underline{x}+1)(\bar{\sigma})/\underline{y}\},p_0\rangle\}\rangle\}$
2. $M((\underline{x}:=0;\underline{y}:=\underline{x}+1) \parallel \underline{x}:=1) =$

$$\lambda\sigma.\{\langle\sigma\{0/\underline{x}\},\lambda\bar{\sigma}.\{\langle\bar{\sigma}\{V(\underline{x}+1)(\bar{\sigma})/\underline{y}\},p_0\rangle\}\rangle\} \parallel \lambda\bar{\bar{\sigma}}.\{\langle\bar{\bar{\sigma}}\{1/\underline{x}\},p_0\rangle\} = \ldots =$$
$$\begin{aligned}\lambda\sigma.\{&\langle\sigma\{1/\underline{x}\},\lambda\bar{\sigma}.\{\langle\bar{\sigma}\{0/\underline{x}\},\lambda\bar{\bar{\sigma}}.\{\langle\bar{\bar{\sigma}}\{V(\underline{x}+1)(\bar{\bar{\sigma}})/\underline{y}\},p_0\rangle\}\rangle\}\rangle,\\ &\langle\sigma\{0/\underline{x}\},\lambda\bar{\sigma}.\{\langle\bar{\sigma}\{1/\underline{x}\},\lambda\bar{\bar{\sigma}}.\{\langle\bar{\bar{\sigma}}\{V(\underline{x}+1)(\bar{\bar{\sigma}})/\underline{y}\},p_0\rangle\}\rangle\}\rangle,\\ &\langle\sigma\{0/\underline{x}\},\lambda\bar{\sigma}.\{\langle\bar{\sigma}\{V(\underline{x}+1)(\bar{\sigma})/\underline{y}\},\lambda\bar{\bar{\sigma}}.\{\langle\bar{\bar{\sigma}}\{1/\underline{x}\},p_0\rangle\}\rangle\}\rangle\}.\end{aligned}$$

3. $M((\underline{x}=\underline{y});\underline{z}:=1 \cup (\underline{x}\neq\underline{y});\underline{z}:=2) =$

$$\begin{aligned}\lambda\sigma.(&\underline{\mathrm{if}}\ \sigma(\underline{x})=\sigma(\underline{y})\ \underline{\mathrm{then}}\ \{\langle\sigma,\lambda\bar{\sigma}.\{\langle\bar{\sigma}\{1/\underline{z}\},p_0\rangle\}\rangle\}\ \underline{\mathrm{else}}\ \emptyset\ \underline{\mathrm{fi}} \cup\\ &\underline{\mathrm{if}}\ \sigma(\underline{x})\neq\sigma(\underline{y})\ \underline{\mathrm{then}}\ \{\langle\sigma,\lambda\bar{\sigma}.\{\langle\bar{\sigma}\{2/\underline{z}\},p_0\rangle\}\rangle\}\ \underline{\mathrm{else}}\ \emptyset\ \underline{\mathrm{fi}})\end{aligned}$$

Contrary to the situation in the previous sections, it is now of some importance to study the notion of yield for $p \in P_3$. We need the following definitions:

<u>DEFINITION 5.5</u> (*paths* for $\langle\sigma,p\rangle$).
A (finite or infinite) sequence $\langle\sigma_1,p_1\rangle,\langle\sigma_2,p_2\rangle,\ldots,$ is a path for $\langle\sigma,p\rangle$ whenever

(i) $\langle\sigma_1,p_1\rangle = \langle\sigma,p\rangle$, and $\langle\sigma_{i+1},p_{i+1}\rangle \in p_i(\sigma_i)$, $i = 1,2,\ldots$

(ii) the sequence is either infinite or, when finite, terminates with $\langle\sigma_n,p_n\rangle$, $n \geq 1$, such that $p_n = p_0$.

The yield of a non-uniform process p may intuitively be understood as follows: Supply p with an argument σ. The pair $\langle\sigma,p\rangle$ determines the set of all paths for $\langle\sigma,p\rangle$. Terminating paths have leaves $\bar{\sigma}$ which are included in the output set, nonterminating paths are reflected by the appearance of $\perp$ in the output set. Here $\perp$ is the undefined state corresponding to nonterminating computations. Its role is fundamental in traditional denotational semantics, but rather less so in our theory.

<u>DEFINITION 5.6</u>. For $p \in P_3$ we define $p^+\colon \Sigma \to P(\Sigma\cup\{\perp\})$ by putting $p_0^+ = \lambda\sigma.\emptyset$, and, for $p \neq p_0$, $p^+ = \lambda\sigma.(\langle\sigma,p\rangle)^+$, where $\langle\sigma,p\rangle^+$ is given by

$$\begin{aligned}\langle\sigma,p\rangle^+ = \{\bar{\sigma} \mid &\text{ there exists a path for } \langle\sigma,p\rangle \text{ terminating with } \langle\bar{\sigma},p_0\rangle\}\\ &\cup (\underline{\mathrm{if}}\ \langle\sigma,p\rangle \text{ has infinite paths } \underline{\mathrm{then}}\ \{\perp\}\ \underline{\mathrm{else}}\ \emptyset\ \underline{\mathrm{fi}}).\end{aligned}$$

Example. Consider the processes $p_1 = M(\underline{x}:=0;\underline{y}:=\underline{x}+1)$ and $p_2 = M((\underline{x}:=0;\underline{y}:=\underline{x}+1) \parallel \underline{x}:=1)$ discussed in the example following definition 5.4. First consider p_1. The pair $\langle\sigma,p_1\rangle$ has as (only) path the sequence $\langle\sigma,p_1\rangle$, $\langle\sigma\{0/\underline{x}\},\lambda\bar{\sigma}.\{\bar{\sigma}\{V(\underline{x}+1)(\bar{\sigma})/\underline{y}\},p_0\rangle\}\rangle$, $\langle\sigma\{0/\underline{x}\}\{V(\underline{x}+1)(\sigma\{0/\underline{x}\})/\underline{y}\},p_0\rangle$, and we see that $p_1^+ = \lambda\sigma{:}\{\sigma\{0/\underline{x}\}\{1/\underline{y}\}\}$.

For p_2^+ we obtain in a similar fashion $p_2^+ = \lambda\sigma.\{\sigma\{0/\underline{x}\}\{1/\underline{y}\},\sigma\{1/\underline{x}\}\{2/\underline{y}\}, \sigma\{1/\underline{y}\}\{1/\underline{x}\}\}$.

We now consider what happens when we extend $\mathcal{L}_3$ with recursion. We only supply the pertinent definitions which should be sufficient for the reader who has understood $\mathcal{L}_1$:

DEFINITION 5.7 (recursion).

a. (syntax) Let $S \in \mathcal{L}_4$. We define (omitting $\underline{x} := ?$ for simplicity):

$$S ::= \underline{x} := s \,|\, b \,|\, S_1;S_2 \,|\, S_1 \cup S_2 \,|\, S_1 \| S_2 \,|\, \xi \,|\, \mu\xi[S].$$

b. Let $P_4 \overset{\text{df.}}{=} P_3$, and let $E = \mathcal{X} \to P_4$, with $\eta \in E$. We define

$\mathcal{M}: \mathcal{L}_4 \to (E \to P_4)$ by

$\mathcal{M}(\underline{x}:=s)(\eta) = \lambda\sigma.\{<\sigma\{\mathcal{V}(s)(\sigma)/\underline{x}\},p_0>\}$,

$\mathcal{M}(b)(\eta) = \lambda\sigma.$ $\underline{\text{if}}$ $\mathcal{W}(b)(\sigma)$ $\underline{\text{then}}$ $\{<\sigma,p_0>\}$ $\underline{\text{else}}$ $\emptyset$ $\underline{\text{fi}}$

$\mathcal{M}(S_1;S_2)(\eta) = \mathcal{M}(S_2)(\eta)\circ\mathcal{M}(S_1)(\eta),\ldots,$

$\mathcal{M}(\xi)(\eta) = \eta(\xi)$,

$\mathcal{M}(\mu\xi[S])(\eta) = \lim_i p_i$, where ($p_0 = p_0$ and)

$\qquad p_{i+1} = \lambda\sigma.\{<\sigma,\mathcal{M}(S)(\eta\{p_i/\xi\})>\}$.

Thus, apart from the use of $\lambda\sigma.\{<\sigma,\ldots>\}$ instead of $\{<\varepsilon,\ldots>\}$ - as we saw in definition 3.5 - the definitions are a straightforward continuation of the preceding theory.

6. STATES, ASSIGNMENT AND SYNCHRONIZATION

We now extend the language $\mathcal{L}_3$ introduced in the previous section with synchronization commands. We proceed in two stages: Firstly, we add to $\mathcal{L}_3$ commands $C,\bar{C}$ as considered previously in section 4. Secondly, we further extend $\mathcal{L}_3$ with *guarded commands* and, in particular, with guards establishing synchronization. (For simplicity, we return to $\mathcal{L}_3$ rather than extending $\mathcal{L}_4$.)

DEFINITION 6.1 ($\mathcal{L}_3$ with synchronization). The language $\mathcal{L}_5$ with elements S, is defined by

$$S ::= \underline{x} := s \,|\, \underline{\text{skip}} \,|\, b \,|\, S_1;S_2 \,|\, S_1 \cup S_2 \,|\, S_1 \| S_2 \,|\, S^* \,|\, C \,|\, \bar{C} \,|\, S\backslash_1 C \,|\, S\backslash_2 C \,|\, \Delta.$$

We observe two restriction operations $\backslash_1$ and $\backslash_2$. The former is the direct counterpart of the $\backslash$ - operation in the uniform case (section 4); the latter is aimed at modelling *deadlock*. In our interpretation, this occurs in a situation where a failing attempt at synchronization has no alternative. This phenomenon is then signalled by the appearance of the *dead* process in the result. The statement Δ is the *abort* statement. We assume from now on that Σ contains the special dead state δ.

Next, we introduce the process domain P_5:

DEFINITION 6.2. Process domain P_5 satisfies the equation

$$P_5 = \{p_0\} \cup (\Sigma \to \mathcal{P}_c((\Sigma \cup \Gamma) \times P_5))$$

We define the semantics of L_5 in

DEFINITION 6.3. The valuation M: $L_5 \to P_5$ is given by

$M(\underline{x}{:=}s) = \lambda\sigma.\ \underline{\text{if}}\ \sigma = \delta\ \underline{\text{then}}\ \{<\delta,p_0>\}\ \underline{\text{else}}\ \{<\sigma\{V(s)(\sigma)/\underline{x}\}>,\ p_0>\}\ \underline{\text{fi}}$

$M(\underline{\text{skip}}) = \lambda\sigma.\ \{<\sigma,p_0>\}$

$M(b) = \lambda\sigma.\ \underline{\text{if}}\ \sigma = \delta\ \underline{\text{then}}\ \{<\delta,p_0>\}\ \underline{\text{else}}\ \underline{\text{if}}\ W(b)(\sigma)\ \underline{\text{then}}\ \{<\sigma,p_0>\}\ \underline{\text{else}}\ \emptyset\ \underline{\text{fi}}\ \underline{\text{fi}}$

$M(S_1;S_2) = M(S_2) \circ M(S_1), M(S_1 \cup S_2) = M(S_1) \cup M(S_2)$,

$M(S_1 \| S_2) = M(S_1) \| M(S_2)$, with "$\|$" defined below

$M(S^*) = \lim_i p_i$, with ($p_0 = p_0$ and)

$\qquad p_{i+1} = (p_i \circ M(S)) \cup \lambda\sigma.\{<\sigma,p_0>\}$

$M(C) = \lambda\sigma.\underline{\text{if}}\ \sigma = \delta\ \underline{\text{then}}\ \{<\delta,p_0>\}\ \underline{\text{else}}\ \{<\gamma,p_0>\}\ \underline{\text{fi}}$, and similarly for $M(\bar{C})$

$M(S\backslash_i C) = M(S)\backslash_i\ \gamma$, with $\backslash_i$ to be defined below, $i = 1,2$

$M(\Delta) = \lambda\sigma.\{<\delta,p_0>\}$

The definitions of "$\|$", "$\backslash_i$" are given in

DEFINITION 6.4. Let β range over $\Sigma \cup \Gamma$. We only give the definitions for p,q of finite nonzero degree:

a. $(\lambda\sigma.X) \| (\lambda\sigma.Y) =$

$\lambda\sigma.(\{x \| \lambda\sigma.Y \mid x \in X\} \cup \{\lambda\sigma.Y \| y \mid y \in Y\} \cup \{<\sigma,p' \| q'> \mid <\gamma,p'> \in X, <\bar{\gamma},q'> \in Y\})$;

here $<\beta,p'> \| \lambda\sigma.Y = <\beta,p' \| \lambda\sigma.Y>$, $\lambda\sigma.X \| <\beta,q'> = <\beta,\lambda\sigma.X \| q'>$

b. $p\backslash_1\gamma = \lambda\sigma.\{<\beta,p'\backslash_1\gamma> \mid <\beta,p'> \in p(\sigma), \beta \neq \gamma,\bar{\gamma}\}$

$p\backslash_2\gamma = \lambda\sigma.\{<\beta,p'\backslash_2\gamma> \mid <\beta,p'> \in p(\sigma), \beta \neq \gamma,\bar{\gamma}\}(\overset{\text{df.}}{=} X)$

$\qquad \cup(\underline{\text{if}}\ (p(\sigma) \neq \emptyset) \wedge (X{=}\emptyset)\ \underline{\text{then}}\ \{<\delta,p_0>\}\ \underline{\text{else}}\ \emptyset\ \underline{\text{fi}})$

We see that in $S\backslash_1 C$, failed attempts at synchronization through $C,\bar{C}$ are not signalled (pairs $<\gamma,p'>$, $<\bar{\gamma},p''>$ are simply deleted), whereas in $S\backslash_2 C$ the failed attempts at synchronization are signalled when they are without alternatives (i.e. in case the set X, obtained from $p(\sigma)$ by deleting pairs $<\gamma,p'>$, $<\gamma,p''>$, equals $\emptyset$).

Examples

1. We determine $M(S)$, where $S \equiv (\underline{x}:=0;C;\underline{x}:=1) \parallel (\underline{y}:=2;\bar{C};\underline{y}:=3)\backslash_2 C$.
Let $M(\underline{x}:=1 \parallel \underline{y}:=3) \overset{\text{df.}}{=} p$. Then $M(S)$ = (omitting dead states for simplicity)
$\lambda\sigma.\{<\sigma\{0/\underline{x}\},\lambda\bar{\sigma}.\{<\bar{\sigma}\{2/\underline{y}\}, \lambda\bar{\bar{\sigma}}.\{<\bar{\bar{\sigma}},p>\}>\}>,$
$\quad <\sigma\{2/\underline{y}\},\lambda\bar{\sigma}.\{<\bar{\sigma}\{0/\underline{x}\},\lambda\bar{\bar{\sigma}}.\{<\bar{\bar{\sigma}},p>\}>\}>\}$
Here the $\lambda\bar{\bar{\sigma}}.\{<\bar{\bar{\sigma}},\ldots>\}$ terms result from the synchronization of the $<\gamma,p'>,<\bar{\gamma},p''>$ terms; also, all pairs $<\gamma,\ldots>,<\bar{\gamma},\ldots>$ are deleted by the restriction operation (no dead states are introduced ; $\backslash_1$ and $\backslash_2$ are indistinguishable in this example). Cf. also the example after definition 4.3.

2. Let $p_1 \overset{\text{df.}}{=} \lambda\sigma.\{<\sigma_1,\lambda\bar{\sigma}.\{<\gamma,p_2'>,<\sigma_3,p_0>\}>\}$,
$\quad p_2 \overset{\text{df.}}{=} \lambda\sigma.\{<\sigma_1,\lambda\bar{\sigma}.\{<\gamma,p_2'>\}>,<\sigma_3,p_0>\}$
Then $\quad p_1\backslash_1\gamma = p_1\backslash_2\gamma = \lambda\sigma.\{<\sigma_1,\lambda\bar{\sigma}.\{<\sigma_3,p_0>\}>\}$
$\quad p_2\backslash_1\gamma = \lambda\sigma.\{<\sigma_2,\lambda\bar{\sigma}.\emptyset>,<\sigma_3,p_0>\}$,
$\quad p_2\backslash_2\gamma = \lambda\sigma.\{<\sigma_1,\lambda\bar{\sigma}.\{<\delta,p_0>\}>,<\sigma_3,p_0>\}$
We see that in process p_2 its subprocess $\lambda\bar{\sigma}.\{<\gamma,p_2'>\}$ has no alternatives for synchronization through γ; hence, deadlock is signalled as the result of restriction.

3. Consider the program $S \equiv ((C \cup (\underline{x}:=1)) \parallel \bar{C})\backslash_2 C$
Let $\sigma_1 = \sigma\{1/\underline{x}\}$. We obtain for $M(S)$ - again ignoring dead states:
$(\lambda\sigma.\{<\gamma,p_0>,<\sigma_1,p_0>\} \parallel \lambda\bar{\sigma}.\{<\bar{\gamma},p_0>\})\backslash_2\gamma =$
$\lambda\sigma.\{<\sigma,p_0>,<\gamma,\lambda\bar{\sigma}.\{<\bar{\gamma},p_0>\}>,<\sigma_1,\lambda\bar{\sigma}.\{<\bar{\gamma},p_0>\}>,$
$\quad <\bar{\gamma},\lambda\bar{\bar{\sigma}}.\{<\gamma,p_0>,<\bar{\bar{\sigma}}_1,p_0>\}>\}\backslash_2\gamma =$
$\lambda\sigma.\{<\sigma,p_0>,<\sigma_1,\lambda\bar{\sigma}.\{<\delta,p_0>\}>\}$
We see that S amounts to either the skip statement, or setting $\underline{x}$ to 1 after which deadlock occurs.

We conclude this part with a few words on p^+ for $p \in P_5$. Let, as usual, β range over $\Sigma \cup \Gamma$. We say that a (finite or infinite) sequence
(*): $<\beta_1,p_1>,<\beta_2,p_2>,\ldots$ is a *path* for $<\sigma,p>$ whenever
(i) $<\beta_1,p_1> = <\sigma,p>, \beta_i \in \Sigma$ and $<\beta_{i+1},p_{i+1}> \in p_i(\beta_i)$, $i = 1,2,\ldots$, and
(ii) the sequence (*) is either infinite or, when finite, terminates in $<\beta_n,p_n>$ with $p_n = p_0$ or $\beta_n \in \Gamma$.

We now define $p^+ : \Sigma \to \mathcal{P}(\Sigma\cup\Gamma\cup\{\bot\})$ by putting $p_0^+ = \lambda\sigma.\emptyset$ and, for $p \neq p_0$, $p^+ = \lambda\sigma.(<\sigma,p>^+)$, where $<\sigma,p>^+$ is given by

$\{\beta_n \mid$ there exists some terminating path for $\langle\sigma,p\rangle$ with $\langle\beta_n,p_n\rangle$
$= \langle\beta_n,p_0\rangle$ or $\langle\beta_n,p_n\rangle = \langle\gamma,p_n\rangle\}$
$\cup$ ($\underline{\text{if}}$ $\langle\sigma,p\rangle$ has an infinite path $\underline{\text{then}}$ $\{\bot\}$ $\underline{\text{else}}$ $\emptyset$ $\underline{\text{fi}}$)
Note that the definition of p^+ assumes the possibility of γ-leaves in the process tree. Normally, this will only occur as the result of some error, since suitable use of the $\backslash_i$ operations will have deleted all ocurrences of any γ from processes p obtained as meaning $M(S)$ of some $S \in L_5$.

We now turn to the consideration of guarded commands and, in particular, of synchronization through guards. We introduce L_6 in

DEFINITION 6.5 (L_5 with guarded commands). The language L_6 is defined by

$$\begin{aligned} S ::= {} & \underline{x} := s \mid \ldots \mid \Delta \mid \qquad (\ldots \text{ as in definition 6.1})\\ & \underline{\text{if}}\ b_1 \to S_1 \ \square \ldots \square\ b_n \to S_n\ \underline{\text{fi}} \mid \underline{\text{do}}\ b_1 \to S_1 \ \square \ldots \square\ b_n \to S_n\ \underline{\text{od}} \mid \\ & \underline{\text{if}}\ b_1;C_1 \to S_1 \ \square \ldots \square\ b_n;C_n \to S_n\ \underline{\text{fi}} \mid \\ & \underline{\text{do}}\ b_1;C_1 \to S_1 \ \square \ldots \square\ b_n;C_n \to S_n\ \underline{\text{od}} \end{aligned}$$

The constructs $\underline{\text{if}} \ldots \underline{\text{fi}}$ and $\underline{\text{do}} \ldots \underline{\text{od}}$ with simple tests as guards are as in Dijkstra [20]; the constructs $b_i\,;C_i \to S_i$ (synchronization through guards) are a simple case of Hoare's CSP (see next section). The meaning of the first two constructs is easy to define: We take $P_6 = P_5$, and define M: $L_6 \to P_6$ by (omitting the clauses which are as in definition 6.3):

DEFINITION 6.6.

a. $M(\underline{\text{if}}\ b_1 \to S_1 \ \square \ldots \square\ b_n \to S_n\ \underline{\text{fi}}) =$
$M(b_1;S_1 \cup \ldots \cup b_n;S_n \cup \neg b_1 \wedge \ldots \wedge \neg b_n;\ \Delta)$

b. $M(\underline{\text{do}}\ b_1 \to S_1 \ \square \ldots \square\ b_n \to S_n\ \underline{\text{od}}) =$
$M((b_1;S_1 \cup \ldots \cup b_n;S_n)^*;(\neg b_1 \wedge \ldots \wedge \neg b_n))$

Remarks

1. Note how, for the $\underline{\text{if}} \ldots \underline{\text{fi}}$ command, if all guards fail Δ is executed; abortion is thus modelled - just as deadlock - by delivering the dead state
2. Definition 6.6b expresses that $\underline{\text{do}} \ldots \underline{\text{od}}$ is equivalent to $\underline{\text{while}}\ b_1 \vee \ldots \vee b_n\ \underline{\text{do}}\ b_1;S_1 \cup \ldots \cup b_n;S_n\ \underline{\text{od}}$
3. For a remark on a possible different interpretation of $b_i \to S_i$ in guarded commands see remark 8.2.

The definition of the other two cases is more involved:

DEFINITION 6.7.

$$\begin{aligned}
&\mathcal{M}(\ \underline{\mathrm{if}}\ b_1;C_1 \to S_1\ \square\ldots\square\ b_n;C_n \to S_n\ \underline{\mathrm{fi}}) = \\
&\lambda\sigma.\ \underline{\mathrm{if}}\ \sigma = \delta\ \underline{\mathrm{then}}\ \{<\delta,p_0>\}\ \underline{\mathrm{else}} \\
&\quad (\underline{\mathrm{if}}\ \mathcal{W}(b_1)(\sigma)\ \underline{\mathrm{then}}\ \{<\gamma_1, \mathcal{M}(S_1)>\}\ \underline{\mathrm{else}}\ \emptyset\ \underline{\mathrm{fi}} \cup \ldots \cup \\
&\quad\ \underline{\mathrm{if}}\ \mathcal{W}(b_n)(\sigma)\ \underline{\mathrm{then}}\ \{<\gamma_n, \mathcal{M}(S_n)>\}\ \underline{\mathrm{else}}\ \emptyset\ \underline{\mathrm{fi}} \cup \\
&\quad\ \underline{\mathrm{if}}\ \mathcal{W}(\neg b_1 \wedge\ldots\wedge\neg b_n)(\sigma)\ \underline{\mathrm{then}}\ \{<\delta,p_0>\}\ \underline{\mathrm{else}}\ \emptyset\ \underline{\mathrm{fi}})\ \underline{\mathrm{fi}}
\end{aligned}$$

Definition 6.7 is perhaps best understood by discussing an example. We use a slight variation on the official syntax, by allowing an $\underline{\mathrm{if}} \ldots \underline{\mathrm{fi}}$ construct with both b;C and b-type of guards. Also, the guard $\underline{\mathrm{true}}$; C is abbreviated to C. Let

$$\begin{aligned}
S_1 &\equiv (\underline{\mathrm{if}}\ C \to \underline{\mathrm{skip}}\ \ \square\ \underline{\mathrm{true}} \to \underline{x}:=1\ \underline{\mathrm{fi}} \parallel \underline{x}:=2)\backslash_2 C \\
S_2 &\equiv (\underline{\mathrm{if}}\ \underline{\mathrm{true}} \to C\ \ \square\ \underline{\mathrm{true}} \to \underline{x}:=1\ \underline{\mathrm{fi}} \parallel \underline{x}:=2)\backslash_2 C
\end{aligned}$$

We show that the deadlock behaviour of these two cases differs. In fact, putting $\sigma_1 = \sigma\{1/\underline{x}\}$, $\sigma_2 = \sigma\{2/\underline{x}\}$, $p_\varepsilon = \lambda\sigma.\{<\sigma,p_0>\}$, $p_1 = \lambda\sigma.\ \{<\sigma_1,p_0>\}$, $p_2 = \lambda\sigma.\{<\sigma_2,p_0>\}$ (and ignoring the case $\sigma = \delta$ for simplicity), we obtain

$$\begin{aligned}
\mathcal{M}(S_1) &= (\lambda\sigma.\{<\gamma,p_\varepsilon>,<\sigma,p_1>\} \parallel p_2)\backslash_2\ \gamma \\
\mathcal{M}(S_2) &= (\lambda\sigma.\{<\sigma,\lambda\bar{\sigma}.\{<\gamma,p_0>\}>,<\sigma,p_1>\} \parallel p_2)\backslash_2\ \gamma
\end{aligned}$$

Hence,

$$\begin{aligned}
\mathcal{M}(S_1) &= \lambda\sigma.\{<\gamma,p_\varepsilon \parallel p_2>\ ,<\sigma,\ p_1 \parallel p_2>, \\
&\qquad <\sigma_2,\ \lambda\bar{\sigma}.\{<\gamma,p_\varepsilon>,<\bar{\sigma},p_1>\}>\}\backslash_2\ \gamma \\
\mathcal{M}(S_2) &= \lambda\sigma.\{<\sigma,\lambda\bar{\sigma}.\{<\gamma,p_0>\} \parallel p_2>,<\sigma,p_1 \parallel p_2>, \\
&\qquad <\sigma_2,\lambda\bar{\bar{\sigma}}.\{<\bar{\bar{\sigma}},\lambda\bar{\sigma}.\{<\gamma,p_0>\}>,<\bar{\bar{\sigma}},p_1>\}>\}\backslash_2\ \gamma \\
&= \lambda\sigma.\{<\sigma,\lambda\bar{\sigma}.\{<\gamma,p_2>,<\sigma_2,\lambda\bar{\bar{\sigma}}.\{<\gamma,p_0>\}>\}>,<\sigma,p_1 \parallel p_2>, \\
&\qquad <\sigma_2,\lambda\bar{\bar{\sigma}}.\{<\bar{\bar{\sigma}},\lambda\bar{\sigma}.\{<\gamma,p_0>\}>,<\bar{\bar{\sigma}},p_1>\}>\}\backslash_2\ \gamma
\end{aligned}$$

Thus,

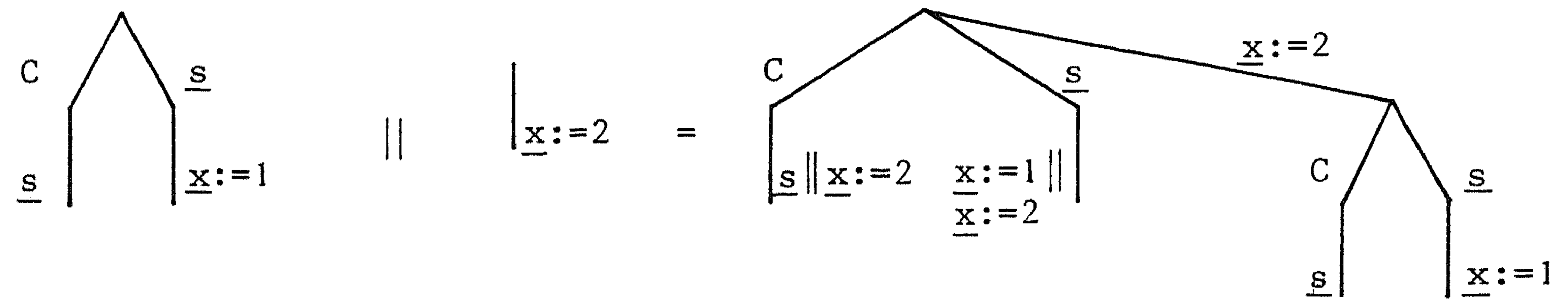
$$\mathcal{M}(S_1) = \lambda\sigma.\{<\sigma,p_1 \parallel p_2>\ ,<\sigma_2,\ \lambda\bar{\sigma}.\{<\bar{\sigma},p_1>\}>\}$$

($\mathcal{M}(S_1)$ shows no deadlock)

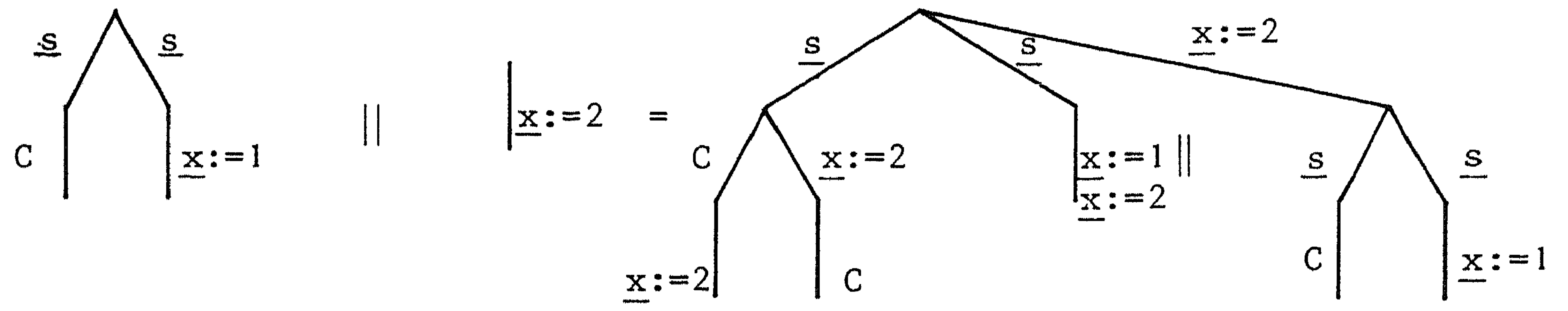
$$\begin{aligned}
\mathcal{M}(S_2) &= \lambda\sigma.\{<\sigma,\ \lambda\bar{\sigma}.\ \{<\sigma_2,\ \lambda\bar{\bar{\sigma}}.\ \{<\delta,p_0>\}>\}>,\ <\sigma,\ p_1 \parallel p_2>, \\
&\qquad <\sigma_2,\lambda\bar{\bar{\sigma}}.\{<\bar{\bar{\sigma}},\ \lambda\bar{\sigma}.\ \{<\delta,p_0>\}>,\ <\bar{\bar{\sigma}},p_1>\}>\}
\end{aligned}$$

($\mathcal{M}(S_2)$ has two possibilities of deadlock)

Some pictures may clarify the situation. Let s be short for skip (or,equivalently, for true). S_1 may be pictured as



and S_2 as



In the first resulting picture, the two branches labelled by C both have an alternative. In the second resulting picture, there are two C - branches without alternative which are turned into dead branches by $\backslash_2$ C.

We conclude this section with the definiton of the semantics of the construct (*): do $b_1 ; C_1 \to S_1 \,\Box\ldots\Box\, b_n ; C_n \to S_n$ od. Defining the meaning of (*) turns out to be fairly involved - at least, we have not been able to come up with a simpler treatment. The problem we have is best explained by comparing statements do $b \to S$ od and do $C \to S$ od. For the former we have the equivalent construct $(b;S)^*;\neg b$ - iterate b;S as long as b is true - and for the latter we would like to be able to write ,by analogy, something like $(C;S)^*; \neg C$. This is not well-defined in L_6. However, it suggests the following approach for dealing with (*): Introduce, besides synchronization elements $\gamma,\bar{\gamma} \in \Gamma$ also elements $\neg\gamma,\neg\bar{\gamma}$ in a set $\neg\Gamma$. The function of $\neg\gamma$ or $\neg\bar{\gamma}$ is, roughly, to express commitment *not* to use the possibility of a $\gamma,\bar{\gamma}$ synchronization - and, instead, deliver the equivalent of a *skip* -statement. We (once more) redefine "$\|$ ". The essence of the new definition consists in (i) $\langle\neg\gamma,\ldots\rangle$ encountering some $\langle\bar{\gamma},\ldots\rangle$ gives no contribution to the result, and (ii) remaining occurrences of $\neg\gamma$ in the result are turned into skip steps by the restriction operation $\backslash_2$. By way of example we consider the merge of the following two sets (returning for a moment to the uniform case for easier notation):

$$(6.1)\qquad \{\langle\gamma,p_1\rangle,\langle\neg\gamma,p_0\rangle\} \parallel \{\langle\bar{\gamma},p_2\rangle,\langle\beta,p_0\rangle\}\backslash_2\,\gamma$$

We want the outcome of (6.1) to consist of the following parts:

(i) $\langle\gamma,\ldots\rangle$ and $\langle\bar{\gamma},\ldots\rangle$,to be deleted by $\backslash_2\,\gamma$

(ii) $\langle\varepsilon,(p_1\parallel p_2)\backslash_2\,\gamma\rangle$, achieved as a result of succesful synchronization between $\langle\gamma,p_1)$ and $\langle\bar{\gamma},p_2\rangle$

(iii) $\langle\neg\gamma,\{\langle\beta,p_0\rangle\}\rangle,\langle\beta,\{\langle\gamma,p_1\rangle,\langle\neg\gamma,p_0\rangle\}\rangle$ as intermediate result, turned by the redefined $\backslash_2$ into
$\langle\varepsilon,\{\langle\beta,p_0\rangle\}\rangle,\ \langle\beta,\{\langle\varepsilon,p_0\rangle\}\rangle$

(iv) no pairs as result of the merge of $\langle\neg\gamma,p_0\rangle$ with $\langle\bar{\gamma},p_2\rangle$

Formally , the various parts of the definition are collected in

DEFINITION 6.8.

a. $P_6' = \{p_0\} \cup (\Sigma \to P_c((\Sigma \cup \Gamma \cup \neg\Gamma) \times P_6'))$

b. For $p,q \in P_6'$ we defined $p\parallel q$ (for $p = \lambda\sigma.X, q = \lambda\sigma.Y$ of finite non-zero degree) as follows: Let β range over $\Sigma \cup \Gamma \cup \neg\Gamma$.

$$(\lambda\sigma.X)\parallel(\lambda\sigma.Y) =$$
$$\lambda\sigma.(\{(x\parallel\lambda\sigma.Y)\mid x \in X\} \cup \{(\lambda\sigma.X\parallel y)\mid y \in Y\} \cup \{\langle\sigma,p'\parallel p''\rangle\mid \langle\gamma,p'\rangle \in X, \langle\bar{\gamma},p''\rangle \in Y\})$$

where $\langle\beta,p\rangle\parallel\lambda\sigma.Y = \langle\beta,p\parallel\lambda\sigma.Y'\rangle$

$$\text{where } Y' = \begin{cases} Y, \text{if } \beta \notin \neg\Gamma \\ Y\backslash\{\langle\beta',p'\rangle\mid \beta = \neg\gamma \text{ and } \beta' = \bar{\gamma} \text{ for some } \gamma \in \Gamma\} \text{ if } \beta \in \neg\Gamma \end{cases}$$

and similarly for $(\lambda\sigma.X)\parallel\langle\beta,p\rangle$

c. $$p\backslash_2\,\gamma = \lambda\sigma.\{\langle\beta',p'\backslash_2\gamma\rangle\mid\langle\beta',p'\rangle \in p(\sigma), \beta \neq \gamma,\bar{\gamma}\}\ (\overset{\text{df.}}{=}X_1) \cup$$
$$\{\langle\sigma,p''\backslash_2\,\gamma\rangle\mid\langle\neg\gamma,p''\rangle \in p(\sigma) \text{ or } \langle\neg\bar{\gamma},p''\rangle \in p(\sigma)\}\ (\overset{\text{df.}}{=}X_2) \cup$$
$$\underline{\text{if}}\ (X_1=\emptyset) \wedge (X_2=\emptyset) \wedge p(\sigma) \neq \emptyset\ \underline{\text{then}}\ \{\langle\delta,p_0\rangle\}\ \underline{\text{else}}\ \emptyset\ \underline{\text{fi}}$$

d. $M(\underline{\text{do}}\ b_1;C_1 \to S_1 \square\ldots\square\ b_n;C_n \to S_n\ \underline{\text{od}}) = \lim_i p_i$, where $(p_0 = p_0$ and$)$

$$p_{i+1} = \lambda\sigma.\ \underline{\text{if}}\ \sigma = \delta\ \underline{\text{then}}\ \{\langle\delta,p_0\rangle\}\ \underline{\text{else}}$$
$$\{\langle\gamma_k, p_i \circ M(S_k)\rangle,\langle\neg\gamma_k,q_{K_\sigma\backslash\{k\}}\rangle\mid k \in K_\sigma\} \cup$$
$$\underline{\text{if}}\ K_\sigma = \emptyset\ \underline{\text{then}}\ \{\langle\sigma,p_0\rangle\}\ \underline{\text{else}}\ \emptyset\ \underline{\text{fi}}$$

where
$$K_\sigma = \{k\mid 1 \leq k \leq n, W(b_k)(\sigma) = tt\}$$
$$q_\emptyset = p_0$$
$$q_{K'} = \lambda\sigma.\{\langle\neg\gamma_{k'},q_{K'\backslash\{k'\}}\rangle\mid k' \in K'\},\quad K' \subseteq \{1,\ldots,n\},\quad K' \neq \emptyset.$$

Clause d of the definition combines a number of ideas. Firstly, the iteration aspect is best understood by comparing it with a similarly structured

definition of the simple $\underline{\mathrm{do}}$ ... $\underline{\mathrm{od}}$ construct $S' \equiv \underline{\mathrm{do}}\ b_1 \rightarrow S_1 \,\square \ldots \square\, b_n \rightarrow S_n$ $\underline{\mathrm{od}}$. For this we can take $\mathcal{M}(S') = \lim_i p_i$, where ($p_0 = p_0$ and) $p_{i+1} = \lambda\sigma.\ \underline{\mathrm{if}}\ \sigma = \delta\ \underline{\mathrm{then}}\ \{\langle\delta,p_0\rangle\}\ \underline{\mathrm{else}}\ (\{p_i \circ \mathcal{M}(S_k) \mid k \in K_\sigma\} \cup (\underline{\mathrm{if}}\ K_\sigma = \emptyset\ \underline{\mathrm{then}}$ $\{\langle\sigma,p_0\rangle\}\ \underline{\mathrm{else}}\ \emptyset\ \underline{\mathrm{fi}})\underline{\mathrm{fi}}$, where $K_\sigma = \{k \mid 1 \le k \le n,\ \mathcal{W}(b_k)(\sigma) = tt\}$. Secondly, it contains synchronization elements γ_k prefixed to $p_i \circ \mathcal{M}(S_k)$ similarly to the use of γ_k prefixing $\mathcal{M}(S_k)$ in definition 6.7. Thirdly, the $\langle \neg\gamma_k, q_{K_\sigma \backslash \{k\}}\rangle$ parts are based on the ideas on the use of $\neg\gamma$'s discussed above. For $K_\sigma = \{1,2,3\}$ we obtain for $\langle \neg\gamma_1, q_{K_\sigma \backslash \{1\}}\rangle$ the following pair: $\langle \neg\gamma_1, \lambda\sigma.\{\langle \neg\gamma_2, \lambda\bar{\sigma}.\{\langle \neg\gamma_3,p_0\rangle\}\rangle, \langle \neg\gamma_3, \lambda\bar{\sigma}.\{\langle\neg\gamma_2, p_0\rangle\}\rangle\}\rangle$.
The reason for the accumulation of the $\neg\gamma_{k'}$ is that only if *all* synchronization through $\gamma_{k'}$ – for which the corresponding $b_{k'}$ is true – fails should skip be the outcome of $\mathcal{M}(S)$. The last part of clause d ensures that if all b_k are false, $\mathcal{M}(S)$ equals skip.

Example. We determine $\mathcal{M}(S)$, for

$$S \equiv \{\ \underline{\mathrm{do}}\ \bar{C} \rightarrow \underline{a}_1\ \underline{\mathrm{od}} \parallel (\bar{C};\ \underline{a}_2 \cup \underline{a}_3)\} \backslash_2 C,$$

where we have returned to the uniform case for simplicity. We obtain, successively, for $\mathcal{M}(S)$:

$$\begin{array}{l}
((\lim_i p_{i+1}) \parallel \{\langle\bar{\gamma}, \{\langle a_2,p_0\rangle\}\rangle, \langle a_3,p_0\rangle\}) \\
\quad = \\
\quad \lim_i\ ((p_{i+1} \parallel \{\langle\bar{\gamma},\{\langle a_2,p_0\rangle\}\rangle, \langle a_3,p_0\rangle\}) \backslash_2 \gamma) \\
\quad = \\
\quad \lim_i\ ((\langle\{\gamma, p_i \circ \{\langle a_1,p_0\rangle\}\rangle, \langle\neg\gamma,p_0\rangle\} \parallel \{\langle\bar{\gamma},\{\langle a_2,p_0\rangle\}\rangle,\langle a_3,p_0\rangle\}) \backslash_2 \gamma) \\
\quad = \\
\quad \lim_i\ \{\langle\varepsilon,\ \{\langle a_1,p_i\rangle\} \parallel \{\langle a_2,p_0\rangle\} \backslash_2 \gamma\rangle, \langle\varepsilon, \{\langle a_3,p_0\rangle\}\rangle, \langle a_3,\{\langle\varepsilon,p_0\rangle\}\rangle\} \\
\quad = \\
\quad \lim_{i\ge 1} \{\langle\varepsilon,\ \{\langle a_1,\{\langle\gamma,\ldots\rangle,\langle\neg\gamma,p_0\rangle\} \parallel \{\langle a_2,p_0\rangle\} \backslash_2 \gamma\rangle, \\
\qquad\qquad\qquad \langle a_2,\{\langle a_1,p_i \backslash_2 \gamma\rangle\}\rangle\}\rangle, \\
\qquad\qquad \langle\varepsilon,\ \{\langle a_3,p_0\rangle\}\rangle, \langle a_3,\{\langle\varepsilon,p_0\rangle\}\rangle\} \\
\quad = \\
\qquad\quad \{\langle\varepsilon,\ \{\langle a_1,\ \{\langle\varepsilon,\{\langle a_2,p_0\rangle\}\rangle,\langle a_2,\{\langle\varepsilon,p_0\rangle\}\rangle\}\rangle, \\
\qquad\qquad\qquad \langle a_2,\ \{\langle a_1,\{\langle\varepsilon,p_0\rangle\}\rangle\}\rangle, \\
\qquad\qquad \langle\varepsilon,\ \{\langle a_3,p_0\rangle\}\rangle, \langle a_3,\{\langle\varepsilon,p_0\rangle\}\rangle\}
\end{array}$$

(where in the final process we have dropped the $\lim_{i\ge 1}$-prefix, since it is independent of i).

7. COMMUNICATION: CSP AND CCS

In this section we define the semantics of two languages where communication is a central concept, viz. Hoare's Communicating Sequential Processes (CSP) [33], and Milner's Calculus for Communicating Systems (CCS) [44].

We begin with CSP, and use the following syntax for a somewhat abstracted version of it:

DEFINITION 7.1 (a version of CSP). The language L_7, with elements S, is defined by

$$S ::= \underline{x} := s \mid \underline{\mathrm{skip}} \mid b \mid S_1;S_2 \mid S_1 \cup S_2 \mid S_1 \| S_2 \mid S^* \mid C?\underline{x} \mid C!s \mid S\backslash C \mid \Delta.$$

To clarify the correspondence between L_7 and CSP, we consider a number of constructs in the syntax of CSP proper:

1. $\{P_1::\ldots P_2?\underline{x}\ldots \| P_2::\ldots P_1!s\ldots]$. This corresponds in L_7 to $\{(\ldots C?\underline{x}\ldots) \| (\ldots C!s\ldots)\}\backslash C$. We see firstly that "$\|$" in L_7 and CSP correspond. Furthermore, communication over the "channel" $P_1 \leftrightarrow P_2$ (using the matching pair $P_2?\underline{x}$ occurring in P_1 and $P_1!s$ occurring in P_2) is mirrored by the pair of communication commands $C?\underline{x}$, $C!s$. (In general, there will be one pair $C?\ldots, C!\ldots$ for each channel $P_i \leftrightarrow P_j$; at the $\ldots$, varying arguments may appear.) Moreover, a restriction construct $S\backslash C$ – with the same meaning as the $S\backslash_2 C$ construct of section 6 – is used. In general, there will be as many restrictions $((S\backslash C)\backslash C')\ldots$ as there are channels $C, C', \ldots$ in the program.
2. Full CSP has constructs of the form $b;C?\underline{x}$ or $b;C!S$ appearing as guards in $\underline{\mathrm{if}}\ldots\underline{\mathrm{fi}}$ or $\underline{\mathrm{do}}\ldots\underline{\mathrm{od}}$ commands. The treatment of these requires a combination of the techniques described in the previous section with those for communication described below. We leave it to the reader to work out the details of this.

We have made no attempt at modelling the distributed termination convention of CSP.

For the definition of the semantics of L_7 we need a new class of processes. The set V is used – as before – for the set of values to be assigned to the variables $\underline{x}$, $\underline{y}$ of the program, as well as for the values communicated over the channels C.

DEFINITION 7.2. The domain P_7 is defined by the equation

$$P_7 = \{p_0\} \cup (\Sigma \to \mathcal{P}_c((\Sigma\cup\Gamma)\times(P_7\cup(V\times P_7)\cup(V\to P_7))))$$

We observe in $\mathcal{P}_c(\cdot)$ an extension of the definition as used for P_5: $\mathcal{P}_c((\Sigma\cup\Gamma)\times P_5)$ is replaced by $\mathcal{P}_c((\Sigma\cup\Gamma)\times(P_7\cup(V\times P_7)\cup(V\to P_7)))$. The domain we now consider is a variation on the process domain of the general format as discussed in section 2, equation (2.4). We leave the details of the necessary modifications of the underlying mathematics to the reader. We shall use π for a typical element of the set $V\to P_7$. As before, we assume that Σ contains a dead state δ, and that for each pair C?, C! in the language there is a corresponding pair $\gamma,\bar{\gamma}$ in Γ.

The semantics of $\mathcal{L}_7$ is described in

DEFINITION 7.3. The valuation $\mathcal{M}$: $\mathcal{L}_7 \to P_7$ is given by

a. $\mathcal{M}(\underline{x}:=s),\ldots,\mathcal{M}(S^*)$ are as in definition 5.4. In particular, $\mathcal{M}(S_1 \| S_2) = \mathcal{M}(S_1) \| \mathcal{M}(S_2)$. For processes p_1,p_2 in P_7, $p_1 \| p_2$ will be redefined below.

b. $\mathcal{M}(C?x) = \lambda\sigma.\{<\gamma,\lambda v.\lambda\bar{\sigma}.\{<\bar{\sigma}\{v/\underline{x}\},p_0>\}>\}$

$\mathcal{M}(C!s) = \lambda\sigma.\{<\bar{\gamma},<\mathcal{V}(s)(\sigma),p_0>>\}$

c. $\mathcal{M}(S\backslash C) = \mathcal{M}(S)\backslash\gamma$

d. $\mathcal{M}(\Delta) = \lambda\sigma.\{<\delta,p_0>\}$.

Clause b is the crucial one; it should be understood with respect to the new definition of "$\|$" contained in

DEFINITION 7.4.

a. $p\|q$ is defined as usual for p or q equal p_0 or of infinite degree. Otherwise, $p = \lambda\sigma.X, q = \lambda\sigma.Y$, and we put

$$(\lambda\sigma.X)\|(\lambda\sigma.Y) = \lambda\sigma.(\{(x\|(\lambda\sigma.Y))\,|\,x\in X\}\cup\{((\lambda\sigma.X)\|y)\,|\,y\in Y\}\cup$$
$$\{<\sigma,p'\|p''>\,|\,<\gamma,\pi>\in X,<\bar{\gamma},<v,p''>>\in Y,\pi(v) = p'\}).$$

Let β be a typical element of $\Sigma\cup\Gamma$, and π of $V\to P_7$. We put $<\beta,p>\|\lambda\sigma.Y = <\beta,p\|\lambda\sigma.Y>, <\beta,<v,p>>\|\lambda\sigma.Y = <\beta,<v,p\|\lambda\sigma.Y>>, <\beta,\pi>\|\lambda\sigma.Y = {} = <\beta.\lambda w.\pi(w)>\|\lambda\sigma.Y = <\beta,\lambda w.(\pi(w)\|\lambda\sigma.Y)>$, and similarly for $\lambda\sigma.X\|<\beta,p>$, etc.

b. $p\backslash\gamma = p\backslash_2\gamma$, with $\backslash_2$ as in definition 6.4.

The heart of the definition is the third term on the right-hand side of the formula for $(\lambda\sigma.X)\|(\lambda\sigma.Y)$. Here the value v is transmitted between

$p = \lambda\sigma.X = \lambda\sigma.\{\ldots,<\gamma,\pi>,\ldots\}$ and $q = \lambda\sigma.Y = \lambda\sigma.\{\ldots,<\bar{\gamma},<v,p''>>,\ldots\}$, determining as possible candidate for continuation in the process $(p\,||\,q)$ when applied to σ, the process $p'\,||\,p''$, with $p' = \pi(v)$: At the synchronization point corresponding to the pair $<\gamma,\bar{\gamma}>$, the value v is supplied to the function π determining process $p' = \pi(v)$ as part of the continuation $p'\,||\,p''$. Let us apply definitions 7.3, 7.4 to the simple example $S \equiv (C?\underline{x}\,||\,C!1)\backslash C$. We obtain $M(S) = M((C?\underline{x}\,||\,C!1)\backslash C) = p\backslash\gamma$, where $p = M((C?\underline{x}\,||\,C!1)$. By definition 7.3, we obtain for p: $p = \lambda\sigma.\{<\gamma,\lambda v.\lambda\bar{\sigma}.\{<\bar{\sigma}\{v/\underline{x}\},p_0>\}>\}\,||\,\lambda\bar{\bar{\sigma}}.\{<\bar{\gamma},$ $<1,p_0>>\} =$ (by def. 7.4)
$\lambda\sigma.(<\gamma,\ldots>,<\bar{\gamma},\ldots>,<\sigma,[\lambda v.\lambda\bar{\sigma}.\{<\bar{\sigma}\{<v/\underline{x}\},p_0>\}](1)\,||\,p_0>\}) =$
$\lambda\sigma.(\ldots,<\sigma,\lambda\bar{\sigma}.\{<\bar{\sigma}\{1/\underline{x}\},p_0>\}>)$. Applying the definition of $\backslash\gamma$ to this results in deletion of the..., and we obtain as final result for $M(S)$: $\lambda\sigma.\{<\sigma,\lambda\bar{\sigma}.\{<\bar{\sigma}\{1/\underline{x}\},p_0>\}>\}$ which is, indeed, a (somewhat elaborate) way of setting $\underline{x}$ to 1.

Definition 7.4 owes a lot to the ideas of Milner [40,44]. Also, it is close to the approach to CSP semantics as described in [24]. The main difference lies in our use of processes as underlying mathematical structure rather than a denotational system with power domains (as in [40]) or with infinite trees (as in [24]). From the variety of operational approaches to CSP semantics we mention [18,25,34,49,53]. Applications of semantics to proof theory (in proving the soundness of a proof system) are studied in [1], cf. also [2].

We close our treatment of CSP with a few words on the definition of yield for $p \in P_7$. In fact, the same definitions both for a *path* for $<\sigma,p>$ and for p^+ can be used as in section 6. Observe, however, that this implies that *only* pairs $<\beta_{i+1},p_{i+1}> \in p_i(\beta_i)$ (for $\beta_i \in \Sigma$) contribute to such paths, whereas pairs $<v,p>$ or functions π do not appear in any path.

We now turn to the definition of Milner's CCS. Contrary to the previous languages, CCS is an expression based language. Synchronization and communication are very similar to CSP, but there is no notion of assignment or sequential composition as we had previously. Also, CCS features recursion rather than iteration. In the syntax we shall give for L_8 we have introduced a deviation from CCS in that we have separated λ-abstraction $(\lambda x\cdot\ldots)$ from synchronization $(<c,\ldots>,<\bar{c},\ldots>)$; in CCS, these notions are combined in the notation $\alpha x\cdot\ldots$ or $\bar{\alpha}v\cdot\ldots$. We first give a simple version of l_8, where recursive declarations are parameterless:

DEFINITION 7.5 (a version of CCS). The language L_8, with elements s,... is defined by

$$s ::= \underline{nil} \mid \langle e,s\rangle \mid \langle c,s\rangle \mid \langle \bar{c},s\rangle \mid s_1 \cup s_2 \mid s_1 \parallel s_2 \mid s\backslash c \mid \xi \mid \mu\xi[s] \mid \lambda x.s$$

where s in $\mu\xi[s]$ is restricted as stated below.

Remarks.

1. In the construct $\langle e,s\rangle$, e is a *simple* expression, defined for example by $e ::= \underline{x} \mid f(e_1,\ldots,e_n)$, for f an n-ary function symbol. We assume that evaluation of e always terminates, delivering a value $v \in V$.
2. Expressions s replace statements S; synchronization prefixes $\langle c,\ldots\rangle$, $\langle \bar{c},\ldots\rangle$ replace commands $C,\bar{C}$ as used above.
3. CCS's construct $\alpha x.\,B$ is written as $\langle c,\lambda x.s_B\rangle$, with s_B the construct in L_8 corresponding to B.
4. We have not taken the trouble to incorporate the relabelling feature of CCS.
5. The recursive construct $\mu\xi[s]$ corresponds to a "call" of some b defined by $b \Leftarrow B$ in CCS. Moreover, the s in $\mu\xi[s]$ is - for the moment - assumed to be of ground type (i.e., not of the form $\lambda x.s'$).

The process domain for L_8 is introduced in

DEFINITION 7.6. The process domain P_8 is defined by

$$P_8 = \{p_0\} \cup \mathcal{P}_c((\Gamma \cup V \cup \{\varepsilon\}) \times (P_8 \cup (V \to P_8))).$$

For the semantics of L_8 we need a class of environments $E = E_1 \times E_2$, where $E_1 = \mathit{Var} \to V$, $E_2 = X \to P_7$. (X is the set of variables ξ used in recursive definitions.) Thus, taking $\eta = \langle \eta_1,\eta_2\rangle \in E$, $\eta_1(x) = v$ and $\eta_2(\xi) = p$ are meaningful equations. As before, $\mathcal{V}$ is the valuation for simple expressions e, yielding results $\mathcal{V}(e)(\eta_1) = v$.

DEFINITION 7.7. The valuation $\mathcal{M}: L_8 \to (E \to P_8)$ is defined by

a. $\mathcal{M}(\underline{nil})(\eta) = p_0$

b. $\mathcal{M}(\langle e,s\rangle)(\eta) = \{\langle \mathcal{V}(e)(\eta_1),\mathcal{M}(s)(\eta)\rangle\}$

c. $\mathcal{M}(\langle c,s\rangle)(\eta) = \{\langle \gamma,\mathcal{M}(s)(\eta)\rangle\}$, where γ corresponds to c, and similarly for $\langle \bar{c},s\rangle$.

d. $M(s_1 \cup s_2)(\eta) = M(s_1)(\eta) \cup M(s_2)(\eta)$, $M(s_1 \| s_2)(\eta) = M(s_1)(\eta) \| M(s_2)(\eta)$, with "$\|$" to be defined below

e. $M(s\backslash c)(\eta) = M(s)(\eta)\backslash\gamma$

f. $M(\xi)(\eta) = \eta_2(\xi)$, $M(\mu\xi[s])(\eta) = \lim_i p_i$, where ($p_0 = p_0$ and) $p_{i+1} = \{<\varepsilon, M(s)(\eta\{p_i/\xi\})>\}$

g. $M(\lambda x.s)(\eta) = \{\lambda v.M(s)(\eta\{v/x\})\}$.

Here $p\backslash\gamma$ is as $p\backslash_1\gamma$ in definition 6.4 (in order to use $\backslash_2$, we would have to extend Γ with a dead symbol δ). Furthermore, the definition of "$\|$" is very similar to the one used in the CSP definition, as can be seen from DEFINITION 7.8. For $X,Y \in P_c(\cdot)$ of finite nonzero degree we put

$$X \| Y = \{x \| Y \mid x \in X\} \cup \{X \| y \mid y \in Y\} \cup$$
$$\{<\varepsilon, p' \| p''> \mid <\gamma,\pi> \in X, <\bar{\gamma}, \{<v,p''>\}> \in Y, \text{ where } \pi(v) = p'\}.$$

Here $<\beta,p> \| Y = <\beta, p \| Y>$, $\pi \| Y = (\lambda v.\pi(v)) \| Y = \lambda v.(\pi(v) \| Y)$, etc.

Example. For constructs b_1, b_2 in CCS defined by $b_1 \Leftarrow \alpha x.x+1.b_1$, and $b_2 \Leftarrow \bar{\alpha}\ y+3.b_2$, we have as corresponding $s_1, s_2 \in L_8$: $s_1 \equiv \mu\xi[<c,\lambda x.<x+1,\xi>>]$, $s_2 \equiv \mu\xi[<\bar{c},<y+3,\xi>>]$, and for $p_i = M(s_i)(\eta)$ we obtain
$p_1 = \lim_i p_1^{(i)}$, where $p_1^{(i+1)} = \{<\varepsilon,\{<\gamma,\{\lambda v.\{<v+1,p_1^{(i)}>\}\}>\}>\}$
$p_2 = \lim_i p_2^{(i)}$, where $p_2^{(i+1)} = \{<\varepsilon,\{<\bar{\gamma},\{<\eta_1(y)+3,p_2^{(i)}>\}>\}>\}$. Also, it can be shown that $(p_1 \| p_2)\backslash\gamma = \lim_i q^{(i)}$, where
$q^{(i+1)} = \{<\varepsilon,\{<\varepsilon,\{<\eta_1(y)+4,\ q^{(i)}>\}>\}>\}$.

Remarks.

1. The use of $<\varepsilon,\ldots>$ in the process theory corresponds to the unobservable action τ of CCS.
2. Processes p in P_8 are quite close to communication trees (Ch.6 of [44]). Important differences are
   (i) the collection of successors of a node in a communication tree is a multiset rather than a set
   (ii) the "mathematical sophistication we do not want to be bothered with" (a quotation from [44] referring to the case of infinite trees) is - if our attempts have been successful - present in our theory.
3. Recursive behaviour expressions *with* parameters - of the form $b(x) \Leftarrow B$ in CCS - can be dealt with very similarly to the above treatment of $\mu\xi[s]$. Without going into details, something along the following lines will have to be done: The syntax of L_8 is extended with the clause

$s ::= \ldots \mid \xi(s_1)\ldots(s_n)$. Moreover, the terms s - in particular the variables ξ - are now supposed to be *typed* as in, e.g., the typed lambda-calculus. We drop the requirement that s in $\mu\xi[s]$ be of ground type, and adapt the choice of p_0 - for the zero element of the CS converging to the meaning of $\mu\xi[s]$ - replacing it by $\underbrace{\lambda v\cdot\ldots\cdot\lambda v}_{n\times}.p_0$, where n is such that the type of ξ is $V^n \to V (n\geq 0)$.

4. The use of the $<\varepsilon,\ldots>$ prefix in definition 7.7 f could be avoided if we were to adopt Milner's requirement that "no behaviour may call itself recursively without passing a guard". Syntactically, this would amount to the requirement that, in a recursive construct $\mu\xi[s]$, ξ occurs in s only within subterms of the form $<c,\ldots\xi\ldots>$ or $<\bar{c},\ldots\xi\ldots>$ or $<e,\ldots\xi\ldots>$. In this way, the contraction property of $T' = \lambda p.M(s)(\eta\{p/\xi\})$ is guaranteed. In our treatment, the same result is obtained by using the CS of iterates $T^i(p_0)$ for T of the form $T = \lambda p.\{<\varepsilon, M(s)(\eta\{p/\xi\})>\}$. (As remarked already in section 3, we are not sure that this precaution is indeed necessary, but we do not know how to prove that $<T^i(p_0)>_i$ is a CS without it.)

This concludes our discussion of CCS semantics. We close with a remark on p^+ for $p \in P_8$. Analogously to what we did in previous sections (3.4), we can define p^+ over the alphabet $V \cup \Gamma$ - where, just as we did for CSP, *paths* are defined such that constituents π of p do not contribute to its paths. Also, we may again put $p \sim q \iff p^+ = q^+$, and investigate propertie of "$\sim$".

8. MISCELLANEOUS NOTIONS IN CONCURRENCY

There is an astounding variety of notions in concurrency, and only a few of them have been investigated in the preceding sections. In this section we briefly comment upon some additional topics. In most cases we provide some suggestions on how the theory of processes could be linked to the notion concerned. Sometimes, we provide no more than some pointers to problems still to be dealt with.

1. *Critical sections*. Let us extend the language L_3 (section 5) with the construct $[S]$. Thus, as syntax for L_3 we have

$$S ::= \underline{x} := s \mid \underline{\text{skip}} \mid b \mid S_1;S_2 \mid S_1 \cup S_2 \mid S_1 \,||\, S_2 \mid S^* \mid [S].$$

Here [S] has as intended meaning that execution of S is not interruptible (S is "locked".) Using P_3 as in section 5, we put $M([S]) = \lambda\sigma.$ $\{\langle\sigma', p_0\rangle \mid \sigma' \in M(S)^+(\sigma)\}$, where p^+ is the (usual) yield of p. This expresses that S is, by [...], turned into an elementary action execution of which cannot be merged at intermediate stages with execution of some parallel statement. Note that σ' in $M(S)^+(\sigma)$ may equal ⊥; strictly speaking, this requires appropriate adaptation of the definition of Σ and of P_3.

2. *Guarded commands.* In section 6 we encountered a guarded command such as $\underline{\text{if}}\ b_1 \rightarrow S_1 \ \square\ b_2 \rightarrow S_2\ \underline{\text{fi}}$, to be modelled by $(b_1;S_1) \cup (b_2;S_2) \cup (\neg b_1;\neg b_2;\Delta)$. This correspondence implies the following: Suppose that, e.g., in state σ it turns out that b_1 is true, and S_1 is selected for execution. Before starting execution of S_1, an interleaving action of some parallel S' may have changed σ to σ' for which b_1 is no longer true, and we see that we cannot be sure that the first action of S_1 is executed with b_1 true, even though the "branch" $b_1;S_1$ was chosen since b_1 was true for σ. A different interpretation of a guarded command is possible - and may even be the intended one - viz. one in which the first elementary action of S_1 is taken immediately after it was selected on the basis of b_1 being true. Let us write $b \Rightarrow S$ for a construct which, contrary to b;S, allows no interleaving actions between b and the first action of S. Including this construct in L_3 requires an extension of M with the clause $M(b \Rightarrow S) = \lambda\sigma.\ \underline{\text{if}}\ W(b)(\sigma)\ \underline{\text{then}}\ M(S)(\sigma)\ \underline{\text{else}}\ \emptyset\ \underline{\text{fi}}$.
(The reader should contrast this with $M(b;S) = M(S) \circ M(b) =$ $\lambda\sigma.\ \underline{\text{if}}\ W(b)(\sigma)\ \underline{\text{then}}\ \{\langle\sigma, M(S)\rangle\}\ \underline{\text{else}}\ \emptyset\ \underline{\text{fi}}$.)

3. *Await statement* [47]. Consider the await statement (*): $\underline{\text{await}}\ b\ \underline{\text{then}}\ S$. Operationally, when execution reaches (*), if b is true then S is executed as in divisible action, if b is false execution waits. Combining the ideas of 1,2 above, we can model (*) by $b \Rightarrow [S]$.

4. *Indivisible parameter passing.* Extend L_3 with the clause

$$S ::= \dots \mid (\lambda \underline{x}.S)(t)$$

where $(\lambda\underline{x}.S)(t)$ is equivalent to $\underline{x} := t;S$, but allows no concurrent action at the ";". We can deal with this by putting $M((\lambda\underline{x}.S)(t)) =$ $= \lambda\sigma.\ M(S)(\sigma\{V(t)(\sigma)/\underline{x}\})$.

5. *Histories on channels.* Extend L_5 with

$$S ::= \ldots \mid \text{read}\ (\underline{x}, C_k) \mid \text{write}\ (s, C_k).$$

Here $C_1,\ldots,C_n$ are channels (as before), but now may contain *sequences* of values. States are now pairs $<\sigma,\rho>$, σ as usual, $\rho = \rho_1,\ldots,\rho_n$, each ρ_i a – possibly infinite – sequence of values, the current contents of channel C_i. Let $\varepsilon(\rho_i)$ test whether the sequence ρ_i is empty, and let $\rho_i\uparrow$ be the last element of ρ_i. Also, $\rho_i\downarrow$ denotes ρ_i with its last element deleted. The central clauses in the definition are

$$\begin{aligned} &M(\text{read}(\underline{x}, C_k)) = \\ &\lambda\sigma\rho.\ \underline{\text{if}}\ \varepsilon(\rho_k)\ \underline{\text{then}}\ \emptyset\ \underline{\text{else}}\ \{<<\sigma\{\rho_k\uparrow/\underline{x}\}, \rho_k\downarrow>, p_0>\}\ \underline{\text{fi}} \\ &M(\text{write}\ (s, C_k)) = \\ &\lambda\sigma\rho.\{<<\sigma, V(s)(\sigma)\,\hat{}\,\rho_k>, p_0>\}. \end{aligned}$$

Here $\hat{}$ denotes concatenation (of sequences over $V^* \cup V^\omega$), and in the ρ-component we have not mentioned the channels which are not referred to (and remain unchanged). Observe that reading from an empty channel results in an empty output. As usual, this captures the operational notion of waiting.

6. *Linking channels.* Let p,q be processes in the domain $P = \{p_0\} \cup P_c((A \cup \Gamma \cup \{\varepsilon\}) \times P)$. Previously, synchronization of p,q was achieved through matching pairs $\gamma,\bar{\gamma}$ occurring in p and q respectively. Such matching can also be "programmed" by using the notation $(p \parallel q)[\gamma:\delta]$ which expresses that δ (in this paragraph standing for some element of Γ rather than for the dead state) now acts as $\bar{\gamma}$, i.e., we define

$$\begin{aligned} (p \parallel q)[\gamma:\delta] = (p \parallel q)\ (\text{"}\parallel\text{" as in section 3})\ \cup \\ \{<\varepsilon, p' \parallel q'> \mid <\gamma,p'> \in p,\ <\delta,q'> \in q\}. \end{aligned}$$

An operation such as $(p \parallel q)[\gamma:\delta]$ is reminiscent of the use of channel linking in Back & Mannila [8]. Also, it resembles the use of equalities $c_i.a = c_j.b$ in [52], which in a similar manner establish linking between "ports" of processes $p, p_1, \ldots, p_n$ occurring in their $\underline{\text{com}}\ldots\underline{\text{moc}}$ construct (albeit that their definition of "$\parallel$" differs from the one used throughout our paper).

7. *Logic*. Let α be a some formula of, e.g., predicate or temporal logic ([50]). We can distinguish a variety of ways of interpreting α in process p. Let, e.g., $p \in P_3$. We may choose $\alpha(p_0) = tt$ or $\alpha(p_0) = ff$, $\alpha(\lambda\sigma.X) = \lambda\sigma.\alpha(X)$, $\alpha(X) = \Lambda_{x \in X}\, \alpha(x)$ or $\alpha(X) = V_{x \in X}\, \alpha(x)$, $\alpha(<\sigma,p_0>) = \alpha(\sigma)$, and, for $p \neq p_0$, $\alpha(<\sigma,p>) = \alpha(\sigma) \vee \alpha(p(\sigma))$, or $\alpha(<\sigma,p>) = \alpha(\sigma) \wedge \alpha(p(\sigma))$. E.g., the combination of definitions $\alpha(X) = \Lambda_{x \in X}\, \alpha(x)$ with $\alpha(<\sigma,p>) = \alpha(\sigma) \vee \alpha(p(\sigma))$ states that $\alpha(p)$ is true in σ whenever α is true in at least one node along each path for $<\sigma,p>$. The implications of these definitions for the model theory of temporal logic deserve further study. We also would like to know whether the results of Emerson & Clarke [19] can be applied in the context of processes.

8. *ADA rendez-vous, distributed processes, data flow*. These notions are mentioned here for the sake of completeness. We have no semantic definitions for them at the moment of writing this. For the ADA rendez-vous this should not be too difficult, because of its close connection with CSP(cf.[26]). For DP([15,27]) and data flow ([14,16,23,35,36,37,38,51,58]) we need further study.

9. *Fairness*. There is a well-known correspondence between fairness and unbounded nondeterminacy (see,e.g., Apt & Olderog [3]). Since our processes allow a smooth treatment of the latter, the question arises as to their role for defining the former. We know how to do this, and we hope to describe it in a future publication (which is not along the lines of the approach sketched in the remark in [11]).

This concludes our discussion of some miscellaneous topics in concurrency, and brings us to the end of this paper.

REFERENCES

[1] APT, K.R., *Formal justification of a proof system for communicating sequential processes*, preprint, EUR, 1981.

[2] APT, K.R., N. FRANCEZ & W.P. DE ROEVER, *A proof system for communicating sequential processes*, ACM TOPLAS, $\underline{2}$ (1980) 359-385.

[3] APT, K.R. & E.R. OLDEROG, *Proof rules dealing with fairness*, Proc. Logic of Programs 1981 (D.Kozen, ed.), 1-9, Lecture Notes in Computer Science 131, Springer, 1982.

[4] APT, K.R. & G.D. PLOTKIN, *A Cook's tour of countable nondeterminism*, Proc. 8th ICALP (S. Even & O. Kariv, eds), 479-494, Lecture Notes in Computer Science, 115, Springer, 1981.

[5] ARNOLD, A. & M. NIVAT, *Metric interpretations of infinite trees and semantics of nondeterministic recursive programs*, Theoretical Computer Science, 11 (1980) 181-206.

[6] ARNOLD, A. & N. NIVAT, *The metric space of infinite trees. Algebraic and topological properties*, Fund. Inf. III, 4 (1980) 445-476.

[7] BACK, R.J., *Semantics of unbounded nondeterminism*, Proc. 7th ICALP (J.W. de Bakker & J. van Leeuwen, eds), 52-63, Lecture Notes in Computer Science, 85, Springer, 1980.

[8] BACK, R.J. & N. MANNILA, *A refinement of Kahn's semantics to handle nondeterminism and communication*, preprint, University of Helsinki, 1982.

[9] DE BAKKER, J.W., *Semantics of infinite processes using generalized trees*, Proc. 6th MFCS (J. Gruska, ed.), 240-252, Lecture Notes in Computer Science, 53, Springer, 1977.

[10] DE BAKKER, J.W., *Mathematical Theory of Program Correctness*, Prentice-Hall International, 1980.

[11] DE BAKKER, J.W. & J.I. ZUCKER, *Denotational semantics of concurrency*, Proc. 14th ACM Symp. on Theory of Computing, pp. 153-158, 1982.

[12] BEKIC, H., *Towards a mathematical theory of processes*, Tech. Report TR 25-125, IBM Lab., Vienna, 1971.

[13] BERGSTRA, J.A. & J.W. KLOP, *Fixed point semantics in process algebras*, Department of Computer Science Technical Report, Mathematisch Centrum, 1982.

[14] BOUSSINOT, F., *Proposition de sémantique dénotationelle pour des réseaux de processus avec opérateur de melange equitable*, Theoretical Computer Science, 18 (1982), 173-206.

[15] BRINCH-HANSEN, P., *Distributed processes: a concurrent programming concept*, Comm. ACM 21 (1978), 934-941.

[16] BROCK, J.D. & W.B. ACKERMANN, *Scenarios: a model of non-determinate computation*, Proc. Formalization of Programming concepts (J. Diaz & I. Ramos, eds.), 252-259, Lecture Notes in Computer Science, 107, Springer, 1981.

[17] DE BRUIN, A., *On the existence of Cook semantics*, Report IW 163/81, Mathematisch Centrum, 1981.

[18] COUSOT, P. & R. COUSOT, *Semantic analysis of communicating sequential processes*, Proc. 7th ICALP (J.W. de Bakker & J. van Leeuwen, eds.) 119-133, Lecture Notes in Computer Science, 85, Springer, 1980.

[19] DUGUNDJI, J., *Topology*, Allen & Bacon, 1966.

[20] DIJKSTRA, E.W., *A Discipline of Programming*, Prentice-Hall, 1976.

[21] EMERSON, E.A. & E.M. CLARKE, *Characterizing correctness properties of parallel programs using fixpoints*, Proc. 7th ICALP (J.W. de Bakker & J. van Leeuwen, eds.) 169-181, Lecture Notes in Computer Science, 85, Springer, 1980.

[22] ENGELKING, R., *General Topology*, Polish Scientific Publishers, 1977.

[23] FAUSTINI, A.A., *An operational semantics for pure data flow*, Proc. 9th ICALP(M.Nielsen & E.M.Schmidt,eds.), 212-224, Lecture Notes Computer Science 140, Springer, 1982.

[24] FRANCEZ, N., C.A.R. HOARE, D.J. LEHMANN & W.P. DE ROEVER, *Semantics of nondeterminism, concurrency and communication*, J. Comp. Syst. Sciences, 19 (1979), 290-308.

[25] FRANCEZ, N., D.J. LEHMANN & A. PNUELI, *Linear history semantics for distributed languages*, Proc. 21st Symp. Foundations of Computer Science, IEEE 1980, 143-151.

[26] GERTH, R., *A sound and complete Hoare axiomatization of the ADA-rendezvous*, Proc. 9th ICALP(M.Nielsen & E.M.Schmidt,eds,), 252-264, Lecture Notes in Computer Science 140, Springer, 1982.

[27] GERTH, R., W.P. DE ROEVER & M. RONCKEN, *Procedures and concurrency: a study in proof*, Proc. Int. Symp. on Programming (M. Dezani-Ciancaglini & U. Montanari, eds.), 132-163, Lecture Notes in Computer Science, 137, 1982.

[28] GORDON, M., *The Denotational Description of Programming Languages*, Springer, 1979.

[29] HAHN, H., *Reelle Funktionen*, Chelsea, 1948.

[30] HENNESSY, N. & R. MILNER, *On observing nondeterminism and concurrency*, Proc. 7th ICALP (J.W. de Bakker & J. van Leeuwen, eds.), 299-309, Lecture Notes in Computer Science, 85, 1980.

[31] HENNESSY, M. & G.D. PLOTKIN, *Full abstraction for a simple parallel programming language*, Proc. 8th MFCS (J. Bečvař, ed.), 108-120, Lecture Notes in Computer Science, 74, 1979.

[32] HITCHCOCK, P. & D. PARK, *Induction rules and termination proofs*, Proc. 1st ICALP (M. Nivat, ed.), 225-251, North-Holland, 1973.

[33] HOARE, C.A.R., *Communicating sequential processes*, Comm. ACM, 21 (1978), 666-677.

[34] HOARE, C.A.R., *A model for communicating sequential processes*, Technical Monograph PRG-22, Oxford University, 1981.

[35] KAHN, G., *The semantics of a simple language for parallel programming*, Proc. IFIP 74, North-Holland, 1974.

[36] KAHN, G. & D.B. MACQUEEN, *Coroutines and networks of parallel processes*, Proc. IFIP 77, 993-998, North-Holland, 1977.

[37] KELLER, R.M., *Denotational models for parallel programs with indeterminate operators*, Formal Description of Programming Concepts (E.J. Neuhold, ed.), 337-366, North-Holland, 1978.

[38] KOSINSKI, P.R., *A straightforward denotational semantics for nondeterminate data flow programs*, Conf. Record 5th ACM Symp. Principles of Programming Languages, 1978, 214-221.

[39] LEHMANN, D.J., *Categories for fixed point semantics*, Proc. 17th IEEE Symp. on Foundations of Computer Science, 122-126, 1976.

[40] MILNE, G. & R. MILNER, *Concurrent processes and their syntax*, J. ACM, 26 (1979), 302-321.

[41] MILNE, R. & C. STRACHEY, *A Theory of Programming Language Semantics*, Chapmann & Hall, 1977.

[42] MILNER, R., *Processes; a mathematical model of computing agents*, Proc. Logic Coll. 73, (Rose & Shepherdson, eds.), North-Holland, 1973.

[43] MILNER, R., *Flow graphs and flow algebras*, J. ACM, 26 (1979), 794-818.

[44] MILNER, R., *A Calculus for Communicating Systems*, Lecture Notes in Computer Science, 92, 1980.

[45] NIVAT, M., *Infinite words, infinite trees, infinite computations*, Foundations of Computer Science III. 2 (J.W. de Bakker & J. van Leeuwen, eds.) 3-52, Mathematical Centre Tracts, 109, 1979.

[46] NIVAT, M., *Synchronization of concurrent processes*, *Formal Language Theory* (R.V. Book, ed.), 429-454, Academic Press, 1980.

[47] OWICKI, S. & D. GRIES, *Verifying properties of parallel programs, an axiomatic approach*, Comm. ACM, 19 (1976), 279-285.

[48] PLOTKIN, G.D., *A power domain construction*, SIAM J. on Comp., 5 (1976), 452-487.

[49] PLOTKIN, G.D., *An operational semantics for CSP*, Proc. IFIP Working Conference on Formal Description of Programming Concepts II (D. Bjørner, ed.), North-Holland, to appear.

[50] PNUELI, A., *The temporal logic of programs*, Proc. 19th Ann. Symp. on Foundations of Comp. Science, IEEE, 46-57, 1977.

[51] PRATT, V.R., *On the composition of processes*, Proc. 9th ACM Symp. on Principles of Programming Languages, 213-223, 1982.

[52] REM, M. & J.L.A. VAN DE SNEPSCHEUT, *Some observations on partially ordered computations*, Preprint, Eindhoven University of Technology, 1981.

[53] ROUNDS, W.C. & S.D. BROOKES, *Possible futures, acceptances, refusals, and communicating processes*, Proc. 22nd Symp. Foundations of Computer Science, IEEE, 140-149, 1981.

[54] SCOTT, D., *Data types as lattices*, SIAM J. on Comp., 5 (1976), 522-587.

[55] SCOTT, D.S., *Domains for denotational semantics*, Proc. 9th ICALP (M.Nielsen & E.M. Schmidt,eds), 577-613, Lecture Notes in Computer Science 140, Springer, 1982.

[56] SMYTH, M.B., *Power domains*, J. Comp. Syst. Sciences, 16 (1978), 23-36.

[57] STOY, J., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.

[58] WADGE, W., *An extensional treatment of dataflow deadlock*, Theoretical Computer Science, 13 (1981), 3-16.

APPENDIX A. HAHN's THEOREM

Since the proof of Hahn's theorem (theorem 2.9) is not easily accessible, we present the proof in this appendix. We repeat the theorem as

THEOREM A (= theorem 2.9). If (M,d) is a complete metric space, then so is $(\mathcal{P}_c(M),d)$, where $\mathcal{P}_c(M)$ denotes the collection of all closed subsets of M, and the distance d for sets is the Hausdorff distance. Moreover, we have, for $\langle X_n\rangle_n$ a CS of closed sets,

$$\lim_n X_n = X \overset{\text{df.}}{=} \{x \mid x = \lim_n x_n,\ \langle x_n\rangle_n \text{ a CS in } M \text{ such that } x_n \in X_n\}.$$

Proof. Clearly, we may assume that $X_n \neq \emptyset$ for almost all n. We show that (i) X is closed, and (ii) $d(X_n,X) \to 0$.
Ad(i). Let $\langle y_n\rangle_n$ be a CS in X with $y_n \to y$. We show that $y \in X$. Let, for each n, $\langle x_{i,n}\rangle_i$ be a CS such that $x_{i,n} \in X_i$, and $x_{i,n} \to y_n$. Consider the diagonal sequence $\langle x_{n,n}\rangle_n$, $x_{n,n} \in X_n$. Then $\langle x_{n,n}\rangle_n$ is a CS, with $x_{n,n} \to y$. Therefore, by the definition of X, we have that $y \in X$.
Ad(ii). The proof of this fact is more involved. We have to show that $\forall\varepsilon\exists N\forall n \geq N\,[d(X_n,X)<\varepsilon]$, i.e.,

(A1) $\forall\varepsilon\exists N\forall n \geq N\forall x_n \in X_n[d(x_n,X)<\varepsilon]$

(A2) $\forall\varepsilon\exists N\forall n \geq N\forall x \in X[d(x,X_n)<\varepsilon]$

or, equivalently,

(A3) $\forall\varepsilon\exists N\forall n \geq N\forall x_n \in X_n\exists x \in X[d(x_n,x)<\varepsilon]$

(A4) $\forall\varepsilon\exists N\forall n \geq N\forall x \in X\exists x_n \in X_n[d(x_n,x)<\varepsilon]$.

We first prove (A3). Choose ε. Then $(*)$: $\exists N\forall m,n \geq N[d(X_m,X_n)<\varepsilon/2]$. Now take any $m \geq N$, and any $x_m \in X_m$. We show how to find $x \in X$ such that $d(x_m,x) < \varepsilon$. There exists a sequence

$$m = N_0 < N_1 < N_2 < \ldots$$

such that $(**)$: $n,n' \geq N_k \Rightarrow d(X_n,X_{n'}) < \varepsilon/2^{k+1}$. Now define a sequence $\langle x_n\rangle_n$ as follows: For $n < N_0$, x_n is arbitrary. For $n = N_0$, $x_n = x_{N_0}$ $(=x_m)$.

For $N_0 < n \leq N_1$: take any x_n such that $d(x_{N_0},x_n) < \varepsilon/2$ (by (*))
For $N_1 < n \leq N_2$: take any x_n such that $d(x_{N_1},x_n) < \varepsilon/4$ (by (**))

...

For $N_k < n \leq N_{k+1}$: take any x_n such that $d(x_{N_k},x_n) < \varepsilon/2^{k+1}$ (by (**))

...

Then $\langle x_n \rangle_n$ is a CS, since for, say, $N_k < n \leq N_{k+1}$, and any $m \geq n$, $d(x_m,x_n) \leq$ $d(x_n,x_{N_{k+1}})+d(x_{N_{k+1}},x_{N_{k+2}})+\ldots+ d(x_{N_{k+\ell}},x_m)$

$< \varepsilon/2^{k+1} + \varepsilon/2^{k+2} + \ldots < \varepsilon/2^{k}$.

So, by completeness of (M,d), $x_n \rightarrow x$ for some x. Thus, $x \in X$. Furthermore, we have $\forall n > m$, $d(x_m,x_n) < \varepsilon/2 + \varepsilon/4 + \ldots$ (by similar reasoning) $< \varepsilon$. Hence, $d(x_m,x) \leq \varepsilon$. Altogether, we have proved (A3). We now prove (A4). Choose some ε. As before, there exists N such that $\forall m,n \geq N[d(X_m,X_n) < \varepsilon/2]$. Let $x \in X$ and $m \geq N$. We show that $d(x,X_m) < \varepsilon$. There exists a CS $\langle x_n \rangle_n$ such that $x_n \rightarrow x$. We have, for $m \geq N$, $d(X_n,X_m) < \varepsilon/2$, so $d(x_n,X_m) < \varepsilon/2$ for all $n \geq N$. Hence (since $x_n \rightarrow x$) $d(x,X_m) \leq \varepsilon/2 < \varepsilon$, which proves (A4). □

APPENDIX B

In this appendix, we present a detailed proof of lemma 2.15. The main part consists in the justification of the definitions of $p \circ q$, $p \cup q$ and $p \| q$, as provided in theorems B7, B12, B14 and B16 and their corollaries. Preliminary to these theorems there are some general lemmas on the Hausdorff distance. Throughout the Appendix lhs and rhs stand for left-hand side and right-hand side, respectively.

Up to lemma B5 we assume X,Y,... are subsets of an arbitrary metric space (M,d), and assume, moreover:

$$x \in X,\ x' \in X',\ y \in Y,\ y' \in Y'.$$

LEMMA B1. *Given* $\ell > 0$ $d(X,X') \le \ell$ *if and only if:*

(B1) $\forall x \exists x'\ d(x,x') \le \ell$, and

(B2) $\forall x' \exists x\ d(x,x') \le \ell$

Proof. $d(X,X') \le \ell$

$\Longleftrightarrow \forall x\ d(x,X') \le \ell$ and $\forall x'\ d(X,x') \le \ell$

$\Longleftrightarrow$ (B1) and (B2). □

We often use a special case of this:

COROLLARY B2. *Suppose there are surjections* $f: Y \to X$, $f': Y \to X'$ *such that* $\forall y\ d(f(y),f'(y)) \le \ell$. *Then* $d(X,X') \le \ell$.

Proof. Clear from lemma B1. □

LEMMA B3. *If*

(B3) $\forall y \exists x \forall x' \exists y' [d(y,y') \le d(x,x')]$

(B4) $\forall y' \exists x' \forall x \exists y [d(y,y') \le d(x,x')]$

then $d(Y,Y') \le d(X,X')$

Proof. (B3) implies, successively,

$$\begin{aligned}\forall y\exists x\forall x' d(y,Y') &\le d(x,x')\\ \forall y\exists x d(y,Y') &\le d(x,X')\\ \forall y d(y,Y') &\le d(X,X')\\ \sup_y d(y,Y') &\le d(X,X')\end{aligned}$$

Similarly, (B4) implies $\sup_{y'}d(Y,y') \le d(X,X')$. The desired result now follows by taking the maximum of the lhs of the last 2 inequalities. □

Actually, we only need lemma B3 in the special case of

COROLLARY B4. *Suppose there are surjections* $f\colon X \to Y$ *and* $f'\colon X' \to Y'$ *such that* $\forall x,x'[d(f(x),f'(x')) \le d(x,x')]$. *Then* $d(Y,Y') \le d(X,X')$.

Proof. Clear from lemma B3. □

LEMMA B5.

$$d(X\cup Y, Y'\cup Y') \le \max(d(X,X'),dY,Y')).$$

Proof. $d(x,X'\cup Y') \le d(x,X') \le d(X,X') \le \text{rhs}$.

Hence, $\sup_x d(x,X'\cup Y') \le \text{rhs}$.
Similarly, $\sup_y d(Y,X'\cup Y') \le \text{rhs}$
$\sup_{x'} d(X\cup Y,x') \le \text{rhs}$
$\sup_{y'} d(X\cup Y,y') \le \text{rhs}$.
Now take the maximum of the lhs of the last 4 lines. □

From now on we consider uniform processes, solving equation (2.2). (See definition 2.10.) We let x,y,... range over elements of A×P, and define $deg(\langle a,p\rangle) = deg(p)$.

We give one more lemma.

LEMMA B6. *For finite* p,p',q,q':

if $d(q,q') \le d(p,p')$

then $d(\langle a,q\rangle,\langle a',q'\rangle) \le d(\langle a,p\rangle,\langle a',p'\rangle)$.

Proof. Clear. □

THEOREM B7. *For finite* q,q':

$$(B5)\qquad d(p\circ q,p\circ q') \le d(q,q')$$

Proof. We prove (B5) simultaneously with

$$(B6)\qquad d(p\circ x,p\circ x') \le d(x,x')$$

by induction on n, where $n = \max(deg(q),deg(q'))$ in de case of (B5), and $n = \max(deg(x),deg(x'))$ in the case of (B6),

If $q = p_0$ or $q' = p_0$ then (B5) is clear. Otherwhise (cf.definition 2.14a)

$$\text{lhs of (B5)} = d(\{p\circ x|x\in q\},\{p\circ x'|x'\in q'\}) \le d(q,q')$$

by the induction hypothesis for (B6) and corollary B4 (taking $f(x) = p\circ x$ and $f'(x') = p\circ x'$). This proves (B5) for the given n. Now (B6) follows for the same n:

$$\begin{aligned} d(p\circ\langle a,q\rangle,p\circ\langle a',q'\rangle) &= d(\langle a,p\circ q\rangle,\langle a',p\circ q'\rangle) \\ &\le d(\langle a,q\rangle,\langle a',q'\rangle) \end{aligned}$$

by (B5) and lemma B6. □

COROLLARY B8. *For finite* q_n, *if* $\langle q_n\rangle_n$ *is a* CS, *then so is* $\langle p\circ q_n\rangle_n$.

Proof. Clear from theorem B7. □

We observe that corollary B8 justifies the definition $p\circ q = \lim_n(p\circ q^{(n)})$.

COROLLARY B9. *Theorem* B7 *holds for all* q,q'.

Proof. For all n, $d(p\circ q^{(n)},p\circ q'^{(n)}) \le d(q^{(n)},q'^{(n)})$, by theorem B7. Now lhs $\to d(p\circ q,p\circ q')$, rhs $\to d(q,q')$, and we see that $d(p\circ q,p\circ q') \le d(q,q')$. □

COROLLARY B10. *Corollary* B8 *holds for all* q_n.

Proof. Clear from corollary B9. □

A more interesting consequence is (for *all* sequences $<q_n>_n$):

COROLLARY B11. *If* $q_n \to q$ *then* $p \circ q_n \to p \circ q$.

Proof. $d(p \circ q_n, p \circ q) \le d(q_n, q) \to 0$. □

Note. Corollary B11 states that "$\circ$" is continuous in its second argument.

THEOREM B12. *For finite* p, p', q, q',

$$d(p \cup q, p' \cup q') \le \max(d(p,p'), d(q,q')).$$

Proof. If any of p, p', q, q' equals p_0, the result is clear. Otherwise it follows immediately from lemma B5. □

Again, we have the corollaries

COROLLARY B13.

a. *For finite* p_n, q_n, *if* $<p_n>_n, <q_n>_n$ *are* CS *then so is* $<p_n \cup q_n>_n$.
   (This justifies the definition $p \cup q = \lim_n (p^{(n)} \cup q^{(n)})$.)
b. *Theorem* B12 *holds for all* p, p', q, q'.
c. *Part a holds for all* p_n, q_n.
d. *If* $p_n \to p, q_n \to q$ *then* $p_n \cup q_n \to p \cup q$ *(for all* p, q).
   Thus, "$\cup$" *is jointly continuous in both arguments.*

Proof. We only prove

b. For all n, $d(p^{(n)} \cup q^{(n)}, p'^{(n)} \cup q'^{(n)}) \le \max(d(p^{(n)}, p'^{(n)}), d(q^{(n)}, q'^{(n)}))$.
   Now let $n \to \infty$.
d. $d(p_n \cup q_n, p \cup q) \le \max(d(p_n, p), d(q_n, q)) \to 0$. □

THEOREM B14. *For finite* p, p', q, q',

(B7) $$d(p \| q, p' \| q') \le \max(d(p,p'), d(q,q')).$$

Proof. We first prove a special case of (B7), namely with $q = q'$:

(B8) $$d(p \| q, p' \| q) \le d(p,p').$$

This is proved simultaneously with

(B9) $d(p \| y, p' \| y) \leq d(p,p')$

(B10) $d(x \| q, x' \| q) \leq d(x,x')$

by induction on n, where $n = \max(deg(p), deg(p')) + deg(q)$ in (B8), $n = \max(deg(p), deg(p')) + deg(y)$ in (B9), and $n = \max(deg(x), deg(x')) + deg(q)$ in (B10).

Now if any of p, p', q equals p_0, then (B8) is clear. Otherwise (cf.definition 2.14c):

$$\begin{aligned} \text{lhs of (B8)} &= d(\{p \| y \mid y \in q\} \cup \{x \| q \mid x \in p\}, \\ &\qquad \{p' \| y \mid y \in q\} \cup \{x' \| q \mid x' \in p'\}) \\ &\leq \max(d_1, d_2) \end{aligned}$$

by lemma B5, where

$$d_1 = d(\{p \| y \mid y \in q\}, \{p' \| y \mid y \in q\})$$

$$d_2 = d(\{x \| q \mid x \in p\}, \{x' \| q \mid x' \in p'\}).$$

Now $d_1 \leq d(p,p')$ by the induction hypothesis for (B9) and corollary B2 (taking $f(y) = p \| y$, $f'(y) = p' \| y$ and $\ell = d(p,p')$). Also $d_2 \leq d(p,p')$ by the induction hypothesis for (B10) and corollary B4 (taking $f(x) = x \| q$ and $f'(x') = x' \| q$). This proves (B8) for the given n. Now (B9) and (B10) follow for the same n. For (B9):

$$\begin{aligned} & d(p \| \langle a,q \rangle, p' \| \langle a,q \rangle) \\ &= d(\langle a, p \| q \rangle, \langle a, p' \| q \rangle) \\ &= \tfrac{1}{2} d(p \| q, p' \| q) \\ &\leq \tfrac{1}{2} d(p,p') \qquad \text{by (B7)} \\ &\leq d(p,p'), \end{aligned}$$

and for (B10):

$$\begin{aligned} & d(\langle a,p\rangle \| q, \langle a',p'\rangle \| q) \\ = \ & d(\langle a, p\| q\rangle, \langle a', p' \| q\rangle) \\ \le \ & d(\langle a,p\rangle, \langle a',p'\rangle) \end{aligned}$$

by (B7) and lemma B6.

Thus we have proved (B8). Similarly (by a symmetrical argument) we can prove (for finite p,q,q'):

(B11) $\quad d(p \| q, p \| q') \le d(q,q')$.

Finally, from (B10) and (B11), and the strong triangle inequality (see the remark after lemma 2.8) we obtain

$$\begin{aligned} d(p\| q, p'\| q') &\le \max(d(p\| q, p'\| q), d(p'\| q, p'\| q')) \\ &\le \max(d(p,p'), d(q,q')). \quad \square \end{aligned}$$

As before, we have the corollaries

COROLLARY B15.

a. *For finite* p_n, q_n, *if* $\langle p_n\rangle_n, \langle q_n\rangle_n$ *are* CS *then so is* $\langle p_n \| q_n\rangle_n$.
   (This justifies the definition $p\| q = \lim_n (p^{(n)} \| q^{(n)})$.)
b. *Theorem* B14 *holds for all* p,p',q,q'.
c. *Part a holds for all* p_n, q_n.
d. *If* $p_n \to p, q_n \to q$ *then* $p_n \| q_n \to p\| q$ *(for all* p,q).
   Thus, "$\|$" *is jointly continuous in both arguments.*

Proof. Clear. $\square$

Now the properties of lemma 2.15 q, i.e., associativity of "$\circ$","$\cup$","$\|$", commentativity of "$\cup$","$\|$", are easily proved. E.g., for associativity of "$\circ$", prove $(p\circ q)\circ r = p\circ(q\circ r)$ first for finite r by induction on *deg*(r), and then for all r by taking $r = \lim_n r^{(n)}$, and using corollary B11.

We conclude this appendix with a proof that "$\circ$" is jointly continuous in both arguments (as yet, we only proved continuity in its second argument).

THEOREM B16. *For finite* q,

(B12) $$d(p\circ q,p'\circ q) \le d(p,p').$$

Proof. We prove (B12) simultaneously with

(B13) $$d(p\circ y,p'\circ y) \le d(p,p'),$$

by induction on *deg*(q) (in (B12)) and *deg*(y) (in (B13)). If $q = p_0$ then (B12) is clear. Otherwise

$$\begin{aligned} d(p\circ q,p'\circ q) &= d(\{p\circ y \mid y \in q\},\{p'\circ y \mid y \in q\}) \\ &\le d(p,p') \end{aligned}$$

by the induction hypothesis for (B13) and corollary B2. As for (B13):

$$\begin{aligned} & d(p\circ\langle a,q\rangle,p'\circ\langle a,q\rangle) \\ &= d(\langle a,p\circ q\rangle,\langle a,p'\circ q\rangle) \\ &= \tfrac{1}{2}d(p\circ q,p'\circ q) \\ &\le \tfrac{1}{2}d(p,p') && \text{by (B12)} \\ &\le d(p,p'). \quad \square \end{aligned}$$

Finally, we obtain the corollaries.

COROLLARY B17.

a. *Theorem* B16 *holds for all* q.

b. *If* $p_n \to p$ *then* $p_n\circ q \to p\circ q$

c. *If* $p_n \to p$ *and* $q_n \to q$ *then* $p_n\circ q_n \to p\circ q$
(*i.e.,* "$\circ$" *is jointly continuous in both arguments*).

Proof. We prove only part c. We have $d(p_n\circ q_n,p\circ q) \le \max(d(p_n\circ q_n,p_n\circ q),$ $d(p_n\circ q,p\circ q) \le \max(d(q_n,q),d(p_n,p)) \to 0$, by the strong triangle inequality and corollaries B9 and B17. $\square$

TEN YEARS OF HOARE'S LOGIC
A SURVEY - PART II: NONDETERMINISM

K.R. Apt
LITP, Paris, France

ABSTRACT

A survey of various results concerning the use of Hoare's logic in proving correctness of nondeterministic programs is presented. Various proof systems together with the example proofs are given and the corresponding soundness and completeness proofs of the systems are discussed. Programs allowing bounded and countable nondeterminism are studied. Proof systems deal with partial and total correctness, freedom of failure and the issue of fairness. The paper is a continuation of APT [1] where various results concerning Hoare's approach to proving correctness of sequential programs are presented.

1. INTRODUCTION

The purpose of this paper is to provide a systematic presentation of the use of Hoare's logic to prove correctness of nondeterministic programs. This paper is a continuation of APT [1] where we surveyed various results concerning the use of Hoare's logic in proving correctness of deterministic programs.

Hoare's method of proving programs correct was introduced in HOARE [14]. Even though it was originally proposed in a framework of sequential programs only, it soon turned out that the method can be perfectly well applied to other classes of programs, as well, in particular to the class of nondeterministic programs.

We discuss the issues in the framework of Dijkstra's nondeterministic programs introduced in DIJKSTRA [7] and concentrate on the issues of soundness and completeness of various proof systems.

This survey is divided into two parts dealing with bounded and countable nondeterminism in sections 3 and 4, respectively. A program allows bounded nondeterminism if at each moment in its execution at most a finite, fixed in advance number of possibilities can be pursued. If this number of possibilities can be countable then we say that the program allows countable nondeterminism.

In section 2 we introduce the basic definitions. In section 3 we discuss partial and total correctness of Dijkstra's programs. The methods used are straightforward generalizations of those which were introduced in the case of sequential programs and discussed in section 2 of APT [1]. This should be contrasted with the presentation in section 4 where total correctness of countably nondeterministic programs and total correctness of programs under the assumption of fairness is discussed. Even though the methods and techniques used there are appropriate generalizations of those used in section 3, various new insights are there needed. Finally, in section 5 bibliographical remarks are provided.

2. PRELIMINARIES

Throughout the paper we fix an arbitrary first order language L with equality containing two boolean constants true and false with obvious meaning. Its formulae are called *assertions* and denoted by letters p, q, r. Simple variables are denoted by letters a, b, x, y, z, expressions by letters s, t and quantifier-free formulae (*Boolean expressions*) by the letter e; p [t/x] stands for a *substitution* of t for all free occurrences of x in p.

All classes of programs considered in this paper contain the skip statement, the assignment statement x:=t and are closed under the composition of programs " ; ".

By a *correctness formula* we mean a construct of the form {p}S{q} where p, q are assertions and S is a program from a considered class. Correctness formulae are denoted by the letter ϕ.

An *interpretation* of L consists of a nonempty domain and assigns to each nonlogical symbol of L a relation or function over its domain of appropriate arity and kind. The letter J stands for an interpretation. Given an interpretation J by a *state* we mean a function assigning to all variables of L values from the domain of interpretation. States are denoted by letters σ, τ. The notions of a value of an expression t in a state σ (written as $\sigma(t)$) and truth of a formula p in a state σ (written as $\models_J p(\sigma)$) are defined in the usual way. A formula p is *true under* J (written as $\models_J p$) if $\models_J p(\sigma)$ holds for all states σ.

We allow two special states: $\perp$ reporting nontermination of a program and fail reporting a failure in execution of a program. We have by definition $\not\models_J p(\perp)$ $\not\models_J p(\underline{\mathrm{fail}})$ for all formulae p. We define $[p]_J$ to the set all states σ which *satisfy* p *under* J (i.e. such that $\models_J p(\sigma)$ holds). Thus by definition for any p and J $\perp \notin [p]_J$ and $\underline{\mathrm{fail}} \notin [p]_J$.

Finally, let Tr_J be the set of all assertions which are true under J.

3. BOUNDED NONDETERMINISM

Denote by S_n the least class of programs such that for all boolean expressions $e_1,\ldots e_m$ and $S_1,\ldots,S_m \in S_n$ $\underline{\mathrm{if}}\ e_1 \to S_1 \,\square \ldots \square\, e_m \to S_m\ \underline{\mathrm{fi}} \in S_n$ and $\underline{\mathrm{do}}\ e_1 \to S_1 \,\square \ldots \square\, e_m \to S_m\ \underline{\mathrm{od}} \in S_n$.

This class of programs was introduced in DIJKSTRA [7] and further extensively studied in DIJKSTRA [8] and various other papers. The boolean expressions e_i in the context of the if and do - constructs are called *guards*. An intuitive meaning of the program $\underline{\mathrm{if}}\ e_1 \to S_1 \,\square \ldots \square\, e_m \to S_m\ \underline{\mathrm{fi}}$ is: choose nondeterministically a guard e_i which evaluates to true and execute the program S_i. In the case when all guards $e_1,\ldots,e_m$ evaluate to false the program *fails*, i.e. its execution improperly terminates. An intuitive meaning of the program $\underline{\mathrm{do}}\ e_1 \to S_1 \,\square \ldots \square\, e_m \to S_m\ \underline{\mathrm{od}}$ is : as long as at least one guard evaluates to true repeatedly do the following: choose any guard e_i which evaluates to true and execute the program S_i. In the case of one guard only the construct $\underline{\mathrm{do}}\ e_1 \to S_1 \,\square \ldots \square\, e_m \to S_m\ \underline{\mathrm{od}}$ is thus equivalent to the usual construct $\underline{\mathrm{while}}\ e_1\ \underline{\mathrm{do}}\ S_1\ \underline{\mathrm{od}}$.

3.1. Semantics of nondeterministic programs

Before we dwell on the issue of correctness of the programs from S_n we define their semantics. We follow here the approach of HENNESSY & PLOTKIN [13] the advantage of which is that it can be easily adopted to several other classes of programs. This semantics is based on the consideration of a transition relation "→" between pairs <S, σ> consisting of a program S and a state σ. The intuitive meaning of the relation

$$<S_1, \sigma> \to <S_2, \tau>$$

is: executing S_1 one step in a state σ can lead (nondeterministically) to a state τ with S_2 being remainder of S_1 still to be executed. It is convenient to assume the empty program E. Then S_2 is E if S_1 terminates in τ. We assume that for any S $E;S = S;E = S$.

Given an interpretation we define the above relation by the following clauses:

(i) $<\underline{skip}, \sigma> \rightarrow <E,\sigma>$

(ii) $<x{:=}t, \sigma> \rightarrow <E,\tau>$

where $\tau(x) = \sigma(t)$ and $\tau(y) = \sigma(y)$ for $y \neq x$

(iii) $<\underline{if}\ e_1 \rightarrow S_1 \ \square \ldots \square\ e_m \rightarrow S_m \underline{fi}, \sigma> \rightarrow <S_i,\sigma>$ if $\models_J e_i(\sigma)$

(iv) $<\underline{if}\ e_1 \rightarrow S_1 \ \square \ldots \square\ e_m \rightarrow S_m \underline{fi}, \sigma> \rightarrow <E, \underline{fail}>$ if $\models_J \bigwedge_{j=1}^{m} \neg e_i(\sigma)$

(v) $<\underline{do}\ e_1 \rightarrow S_1 \ \square \ldots \square\ e_m \rightarrow S_m \underline{od}, \sigma> \rightarrow <S_i\ ;\ \underline{do}\ e_1 \rightarrow S_1 \ \square \ldots \square\ e_m \rightarrow S_m od,\sigma>$
$\underline{if}$ $\models_J e_i(\sigma)$

(vi) $<\underline{do}\ e_1 \rightarrow S_1 \ \square \ldots \square\ e_m \rightarrow S_m\ \underline{od}, \sigma> \rightarrow <E, \sigma>$ if $\models_J \bigwedge_{i=1}^{m} \neg e_i(\sigma)$

(vii) if $<S_1,\sigma> \rightarrow <S_2,\tau>$ then $<S_1\ ;\ S, \sigma> \rightarrow <S_2\ ;\ S, \tau>$.

Let $\rightarrow^*$ stand for the transitive, reflexive closure of $\rightarrow$.

We now introduce the following definitions.

DEFINITION

(i) S can *diverge from* σ if there exists an infinite sequence $<S_i,\sigma_i>$ $(i = 0,1,\ldots)$ such that $<S,\sigma> = <S_0,\sigma_0> \rightarrow <S_1,\sigma_1> \rightarrow \ldots$

(ii) S can *fail from* σ if
$<S,\sigma> \rightarrow^* <S_1,\underline{fail}>$ for some S_1.

(iii) A finite sequence $<S_i,\sigma_i>$ $(i=0,1,\ldots,k)$ such that $<S,\sigma> = <S,\sigma_0> \rightarrow <S_1,\sigma_1> \rightarrow \ldots \rightarrow <S_k,\sigma_k> = <E,\sigma_k>$ is called a *computation starting in* $<S,\sigma>$; k is the length of this computation.

The following lemma will be needed later.

LEMMA 1. *If* S *cannot diverge from* σ *then there exists a natural number* k *such that all computations starting in* $<S,\sigma>$ *are of length at most* k.

PROOF. Consider the set of all finite sequences $<S,\sigma> = <S_0,\sigma_0> \rightarrow \ldots \rightarrow <S_n,\sigma_n>$ ordered by the subsequence ordering. This set forms a finitely branching tree. If the desired k did not exist then this tree would be infinite. By König's Lemma it would then contain an infinite branch which contradicts the assumption. □

We define now two types of semantics for the programs from $\mathcal{S}_n$ by putting

$$\mathcal{M}[\![S]\!](\sigma) = \{\tau \mid \langle S,\sigma\rangle \to^{*} \langle E,\tau\rangle\}$$

and

$$\begin{aligned}\mathcal{M}_{tot}[\![S]\!](\sigma) = \mathcal{M}[\![S]\!](\sigma) \;&\cup\; \{\bot \mid S \text{ can diverge from } \sigma\}\\ &\cup\; \{\underline{fail} \mid S \text{ can fail from } \sigma\}\end{aligned}$$

Both semantics depend on the interpretation J but we do not mention this dependence hoping that no confusion will arise. The difference between these two semantics lies in the way the "negative" informations about the program are dealt with - either they are dropped or they are explicitly mentioned.

3.2. Partial and total correctness

While studying a correctness of programs we are interested in various properties namely

(a) whether all proper states generated (or produced) by the program satisfy a given post-condition,

(b) whether the program always terminates, and

(c) whether none of the executions of the program leads to a failure.

We are usually interested in executions starting in a state satisfying some initial pre-condition. The above properties lead to various possible interpretations of the correctness formulae {p} S {q}. Let

$$\mathcal{M}[\![S]\!]([p]_J) = \bigcup_{\sigma\in[p]_J} \mathcal{M}[\![S]\!](\sigma)$$

and

$$\mathcal{M}_{tot}[\![S]\!]([p]_J) = \bigcup_{\sigma\in[p]_J} \mathcal{M}_{tot}[\![S]\!](\sigma).$$

We define

$$\models_J \{p\}\, S\, \{q\} \text{ iff } \mathcal{M}[\![S]\!]([p]_J) \subseteq [q]_J,$$

$$\models_{J,\,tot} \{p\}\, S\, \{q\} \text{ iff } \mathcal{M}_{tot}[\![S]\!]([p]_J) \subseteq [q]_J.$$

Informally speaking, $\models_J \{p\}\, S\, \{q\}$ means that any properly terminating execution of S starting in a state satisfying p leads to a state satisfying q ;

$\models_{J,tot}$ {p} S {q} in addition guarantees that any execution of S starting in a state satisfying p properly terminates. If $\models_J$ {p} S {q} holds we say that the program S is *partially correct under* J (with respect to p and q). If $\models_{J,tot}$ {p} S {q} holds we say that the program S is *totally correct under* J (with respect to p and q).

3.3. A proof system for partial correctness

We now present a formal system allowing us to deduce formally partial correctness of programs from S_n. Its axioms and proof rules are the following

AXIOM 1 : skip axiom

$$\{p\}\ \underline{skip}\ \{p\}$$

AXIOM 2 : assignment axiom

$$\{p[t/x]\}\ x:=t\ \{p\}$$

RULE 3 : composition rule

$$\frac{\{p\}\ S_1\{r\},\{r\}\ S_2\{q\}}{\{p\}\ S_1\ ;\ S_2\ \{q\}}$$

RULE 4 : $\underline{if}$ - rule

$$\frac{\{p \wedge e_i\}\ S_i\ \{q\},\ i = 1,\dots,m}{\{p\}\ \underline{if}\ e_1 \rightarrow S_1\ \square \dots \square\ e_m \rightarrow S_m \underline{fi}\{q\}}$$

RULE 5 : $\underline{do}$ - rule

$$\frac{\{p \wedge e_i\}\ S_i\ \{p\},\ i = 1,\dots,m}{\{p\}\ \underline{do}\ e_1 \rightarrow S_1\ \square \dots \square e_m \rightarrow S_m \underline{od}\ \{p \wedge \bigwedge_{i=1}^{m} \neg e_i\}}$$

p is called a *loop invariant*.

RULE 6 : consequence rule

$$\frac{p \rightarrow p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$$

We call this proof system N. For A being a set of assertions and a correctness formula ϕ we write $A \vdash_N \phi$ to denote the fact that there exists a proof of ϕ in N which uses as assumptions for the consequence rule assertions from A.

3.4. An example of a proof in N

To illustrate the use of the proof system N we now provide the following example. Let S stand for the following program

```
do 2|x ∨ 3|x →
   if 2|x → x:=x/2 ; a:=b+1
     □3|x → x:=x/3 ; b:=b+1
     □4|x → x:=x/4 ; a:=a+2 fi
od
```

where x, a, b are integer variables. This program computes the greatest powers of 2 and 3 which divide x. We now present a formal proof of this fact. More precisely we prove

(1) $$Tr_{J_0} \vdash_{N} \{a = 0 \wedge b = 0 \wedge x = z\}\ S\ \{z = x.2^a.3^b \wedge \neg(2|x \vee 3|x)\}$$

where J_0 is the standard interpretation of the language of Peano arithmetic augmented with the division operation and divisibility relation.

We present the proof in a "top-down" fashion. We choose $p \equiv z = x \cdot 2^a \cdot 3^b$ to be the loop invariant. We now show

(2) $$a = 0 \wedge b = 0 \wedge x = z \rightarrow p,$$

(3) $$\{p \wedge (2|x \vee 3|x)\}\ S_1\ \{p\}$$

where S_1 is the loop body,

(4) $$p \wedge\neg (2|x \vee 3|x) \rightarrow z = x\cdot 2^a\cdot 3^b \wedge\neg (2|x \vee\neg (2|x \vee 3|x).$$

Note that (3) implies by the do-rule $\{p\}\ S\ \{p \wedge\neg (2|x \vee 3|x)\}$ which together with (2) and (4) implies by the consequence rule (1). Both (2) and (4) are obvious.

To show (3) we have to show

(5) $\{p \wedge (2|x \vee 3|x) \wedge 2|x\}$ x:=x/2 ; a:=a+1 $\{p\}$,

(6) $\{p \wedge (2|x \vee 3|x) \wedge 3|x\}$ x:=x/3 ; b:=b+1 $\{p\}$,

(7) $\{p \wedge (2|x \vee 3|x) \wedge 4|x\}$ x:=x/4 ; a:=a+2 $\{p\}$

and apply the if-rule.

We now prove (5). By the assignment axiom

$$\{z = x \cdot 2^{a+1} \cdot 3^b\} \text{ a:=a+1 } \{p\}$$

and

$$\{z = (x/2) \cdot 2^{a+1} \cdot 3^b\} \text{ x:=x/2 } \{z = x \cdot 2^{a+1} \cdot 3^b\}$$

so by the composition rule $\{z = (x/2) \cdot 2^{a+1} \cdot 3^b\}$ x:=x/2 ; a:=a+1 $\{p\}$ which by the consequence rule implies (5). Proofs of (6) and (7) are similar and left to the reader.

Note. To ensure that the application of the division operation does not result in producing non-integer values we should actually use here the following assignment rule in the case of division operation:

$$\frac{p[(a/b)/x] \rightarrow b|a}{\{p[(a/b)/x]\}x := a/b\{p\}} .$$

We leave it to the reader checking that the above proof remains correct when this assignment rule is used.

3.5. Soundness of N

To justify the proofs in the system N one has to prove its *soundness* in the sense of the following theorem which links provability of the correctness formulae with their truth.

THEOREM 1. *For every interpretation* J, *set of assertions* A *and correctness formula* ϕ *the following holds: if all assertions from* A *are true under* J *and* $A \vdash_N \phi$ *then* ϕ *is true under* J.

In other words if $Tr_J \vdash_N \phi$ *then* $\models_J \phi$.

We call correctness formula *valid* if it is true under all interpretations J and a proof rule *sound* if for all interpretations J it preserves the truth under J of correctness formulae (and in the case of the consequence rule, assertions).

To prove the soundness of N it is sufficient to show that all axioms of N are valid and all proof rules of N are sound since the desired conclusion follows then by the induction on the length of proofs. As an example proof we now show the soundness of the do-rule.

Let S stand for do $e_1 \rightarrow S_1 \square \ldots \square e_m \rightarrow S_m$ od. Fix an interpretation J and assume that all the premises of the do-rule are true under J, i.e. that

(8) $\quad M[\![S_i]\!]([p \wedge e_i]_J) \subseteq [p]_J$ for $i = 1,\ldots,m$.

Let $\tau \in M[\![S]\!]([p]_J)$. Then for some $\sigma \in [p]_J$ $\tau \in M[\![S]\!](\sigma)$. By the definition of M we have

$$<S,\sigma_0> \rightarrow^* <S_1,\sigma_1> \rightarrow^* \ldots \rightarrow^* <S_\ell,\sigma_\ell> \rightarrow <E,\sigma_\ell>$$

where $\sigma = \sigma_0$, $\tau = \sigma_\ell$ and for all $j = 0,\ldots, \ell - 1$
$\sigma_j \in [e_{k_j}]_J$ and $\sigma_{j+1} \in M[\![S_{k_j}]\!](\sigma_j)$ for some $k_j \in \{1,\ldots,m\}$ and
$\sigma_\ell \in [\bigwedge_{i=1}^{m} \neg e_i]_J$. We have $\sigma_0 \in [p]_J$ and if for some $j \in \{0,\ldots,\ell-1\}$
$\sigma_j \in [p]_J$ then by (8) $\sigma_{j+1} \in M[\![S_{k_j}]\!]([p \wedge e_{k_j}]_J) \subseteq [p]_J$, i.e. $\sigma_{j+1} \in [p]_j$.
Thus for all $j = 0,\ldots,\ell$ $\sigma_j \in [p]_J$. In particular $\sigma_\ell \in [p]_J$ which means that $\tau \in [p \wedge \bigwedge_{i=1}^{m} \neg e_i]_J$. This proves the truth under J of the conclusion of the do-rule and thereby concludes the proof of the soundness of the rule.

3.6. Completeness of N in the sense of Cook

A converse property to that of soundness of a proof system is completeness which links truth of the correctness formulae with their provability. Unfortunately a converse implication to this theorem 1 can be proved only for a special type of interpretations J. This issue is discussed at length in APT [1] in sections 2.7. and 2.8. where we refer the reader for the details. We restrict ourselves here to presenting the appropriately adopted definitions without entering into any discussion of the results.

Define

$$\mathrm{post}_J(p,S) = M[\![S]\!]([p]_J)$$
$$\mathrm{pre}_J\,(S,q) = \{\sigma : M[\![S]\!](\sigma) \subseteq [q]_J\}$$

Note that these sets are characterized by the following equivalences (the second of them is just a rewording of the definition):

$$\begin{aligned} \models_J \{p\}\ S\ \{q\} \ \ \text{iff} \quad & [p]_J \subseteq \mathrm{pre}_J(S,q) \\ \text{iff} \quad & \mathrm{post}_J(p,S) \subseteq [q]_J. \end{aligned} \tag{9}$$

Let $\mathcal{S}_0$ be a class of programs.

Call the language L *expressive relative to* J *and* $\mathcal{S}_0$ if for all assertions p and programs $S \in \mathcal{S}_0$ there exists an assertion q which defines $\mathrm{post}_J(p,S)$. If J is such that L is expressive relative to J and $\mathcal{S}_0$ we write $J \in \mathrm{Exp}(L,S_0)$. It is worthwhile to note that in the definition of expressiveness we can alternatively require definability of $\mathrm{pre}_J(S,q)$ instead of $\mathrm{post}_J(p,S)$ (see APT [1]).

Definition A proof system G for S_0 is complete *in the sense of Cook* if, for every interpretation $J \in \mathrm{Exp}(L,\mathcal{S}_0)$ and every asserted program ϕ if $\models_J \phi$, then $Tr_J \mid\!\overline{_G}\ \phi$.

This definition of completeness is, as the name indicates, due to COOK [6].

Now, the proof system N for $\mathcal{S}_n$ is complete in the sense of Cook. The proof proceeds by induction on the structure of the programs.

The only two nontrivial cases are these of composition and the do-construct.

If $\models_J \{p\}\ S_1;S_2\ \{q\}$ then clearly $\models_J \{p\}\ S_1\ \{r\}$ and $\models_J \{r\} S_2 \{q\}$ where r defines $\mathrm{pre}_J(S_2,q)$; so, by the induction hypothesis and the composition rule, $Tr_J \mid\!\overline{_N}\ \{p\}S_1\ ;\ S_2\{q\}$. If $\models_J \{p\}S\{q\}$, where $S \equiv \underline{\text{do}}\ e_1 \rightarrow S_1 \square \ldots \square e_m \rightarrow S_m\ \underline{\text{od}}$, then we must find a loop invariant r such that for $i = 1,\ldots,m$ $\models_J \{r \wedge e_i\}\ S_i\ \{r\}$, $\models_J p \rightarrow r$ and $\models_J (r \wedge \bigwedge_{i=1}^{m} \neg\, e_i) \rightarrow q$. Then by the induction hypothesis and the consequence rule $Tr_J \mid\!\overline{_N}\ \{p\}\ S\ \{q\}$.

We choose r to be an assertion defining $\mathrm{pre}_J(S,q)$. Then by (9) $\models_J \{r\}\ S\ \{q\}$ so also $\models_J \{r\}\ \underline{\text{if}}\ e_i \rightarrow S_i \ \square\ \neg\, e_i \rightarrow \underline{\text{skip}}\ \underline{\text{fi}}\ ;\ S\{q\}$ for all

$i = 1,\dots,m$ as for any σ $M[\![\underline{\text{if}}\ e_i \to S_i \ \square\ \neg\, e_i \to \underline{\text{skip}}\ \underline{\text{fi}}\ ;\ S]\!](\sigma) \subset M[\![S]\!](\sigma)$ clearly holds. Now, since r defines $\text{pre}_J(S,q)$, then as in the case treated above $\models_J \{r\}\ \underline{\text{if}}\ e_i \to S_i\ \square\ e_i \to \underline{\text{skip}}\ \underline{\text{fi}}\ \{r\}$ from which $\models_J \{r \wedge e_i\}\ S_i\ \{r\}$ follows. By (9) we have $\models_J p \to r$ and $\models_J (r \wedge \bigwedge_{i=1}^{m} \neg\, e_i) \to q$ follows from the definition of r. This concludes the proof.

3.7 A proof system for total correctness

To prove total correctness of programs from S_n we must provide proof rules ruling out possibility of failure and nontermination.

A possible failure in an execution of a program from S_n can be caused only by the if-construct. Clearly the if-rule does not rule out a possibility of failure. However, a small refinement of this rule suffices to prove the lack of failure. We only need to ensure that at each moment an if-statement is to be executed at least one of its guards evaluates to true. This is achieved by the following modification

RULE 7 : if-rule II

$$\frac{p \to \bigvee_{i=1}^{m} e_i,\ \{p \wedge e_i\}\ S_i\ \{q\}_{i=1,\dots,m}}{\{p\}\ \underline{\text{if}}\ e_1 \to S_1\ \square \dots \square\, e_m \to S_m\ \underline{\text{fi}}\ \{q\}}$$

A possible nontermination of an execution of a program from S_n can be caused only by the do-construct and clearly the present do-rule does not rule out such a possibility. The following modification of the do-rule suffices to prove termination of each do-construct. This rule is due to HAREL [11] where a different formalism is used.

RULE 8 : do-rule II

$$\frac{\begin{array}{l} p(n) \wedge n > 0 \to \bigvee_{i=1}^{m} e_i\ ,\ p(0) \to \bigwedge_{i=1}^{m} \neg\, e_i, \\ \{p(n) \wedge n > 0 \wedge e_i\}\ S_i\ \{\exists m < n\ p(m)\}_{i=1,\dots,m} \end{array}}{\{\exists n\ p(n)\}\ \underline{\text{do}}\ e_1 \to S_1\ \square \dots \square\, e_m \to S_m\ \underline{\text{od}}\ \{p(0)\}}$$

Here p(n) is an assertion with a free variable n which does not appear in the programs and ranges over natural numbers.

Let NT denote the proof system obtained from N by replacing the if and do-rules by their modified versions. This proof system is appropriate for proving total correctness of programs from S_n.

To illustrate the use of the system we now indicate how to modify the proof given in section 3.4. to demonstrate the total correctness of the program there considered, i.e. to prove (1) within NT.

We choose $p(n) \equiv p \wedge \exists a_1, b_1, x_1\ (x = 2^{a_1} \cdot 3^{b_1} \cdot x_1 \wedge \neg(2|x_1 \vee 3|x_1) \wedge n = a_1 + b_1)$.

The second component of p(n) states that n is the sum of powers of 2 and 3 which divide x.

We now have

(10) $a = 0 \wedge b = 0 \wedge x = z \rightarrow \exists n\ p(n)$,

(11) $p(n) \wedge n > 0 \rightarrow 2|x \vee 3|x$,

(12) $p(0) \rightarrow \neg(2|x \vee 3|x)$,

(13) $\{p(n) \wedge n > 0\}\ S_1\ \{\exists m < n\ p(m)\}$

where the last correctness formula can be proved using the <u>if</u>-rule II since $p(n) \wedge n > 0 \rightarrow 2|x \vee 3|x \vee 4|x$ holds. The proof of (13) is a small modification of the proof of (3) and is left to the reader. Now by the <u>do</u>-rule II, (10) and (12) we obtain (1) as desired.

3.8. Arithmetical soundness and completeness of NT

As explained in section 2.11 of APT [1] when trying to prove soundness of a proof for total correctness one has to revise appropriately the notion of soundness. We follow here the approach of HAREL [11] also adopted in APT [1]. We recall the introduced definitions.

Let L be an assertion language and let L^+ be the minimal extension of L containing the language L_p of Peano arithmetic and a unary relation nat(x). Call an interpretation J of L^+ *arithmetical* if its domain includes the set of natural numbers, J provides the standard interpretation for L_p, and nat(x), is interpreted as the relation "to be a natural number". Additionally, we require that there exists a formula of L^+ which, when interpreted under J, provides the ability to encode finite sequences of elements from the domain of J into on element. (The last requirement is needed only for the completeness proof.)

One of the examples of an arithmetical interpretation is of course J_0. It is important to note that any interpretation of an assertion language L with an infinite domain can be extended to an arithmetical interpretation of L^+. Clearly, the proof system NT is suitable only for assertion languages of the form L^+, and an expression such as $p(n)$ is actually a shorthand for $nat(n) \wedge p(n)$.

We now say that a proof system G for total correctness is *arithmetically sound* if, for all arithmetical interpretations J and asserted programs ϕ $Tr_J \vdash_G \phi$ implies $\models_{J,tot} \phi$.

It can be shown that the proof system NT is arithmetically sound. The case of the if-rule II is easily handled. The proof of soundness of the do-rule II for the case of arithmetical interpretations is in turn an easy modification of the proof of soundness of the do-rule where one simply parametrizes the invariant p. The proofs of other cases are the same as before.

We say that a proof system G is *arithmetically complete* if for all arithmetical interpretations J and asserted programs ϕ $\models_{J,tot} \phi$ implies $Tr_J \vdash_G \phi$.

To show the arithmetical completeness of the system NT we first introduce the following notion:

$$pret_J(S,q) = \{\sigma : \mathcal{M}_{tot}[\![S]\!](\sigma) \subseteq [q]_J\}.$$

pret stands in the same relation to total correctness as pre does to partial correctness : we have $\models_{J,tot} \{p\}\ S\ \{q\}$ iff $[p]_J \subseteq pret_J(S,q)$.

Thanks to the provision for coding of finite sequences it can be shown that for any arithmetical interpretation J there exists an assertion which defines $pret_J(S,q)$. This fact is not completely obvious as the definition of $pret_J(S,q)$ also mentions (the nonexistence of) infinite sequences. This difficulty can be however circumvented by making use of Lemma 1.

The completeness proof proceeds by induction on the structure of programs. The only cases different from the corresponding ones in the completeness proof of N are those of if and do-constructs. Let J be an arithmetical interpretation.

If $\models_{J,tot} \{p\}\ \underline{\text{if}}\ e_1 \rightarrow S_1 \ \square \ \ldots \ \square e_m \rightarrow S_m\ \underline{\text{fi}}\ \{q\}$ then by definition $\models_J p \rightarrow \bigvee_{i=1}^{m} e_i$ and $\models_{J,tot} \{p \wedge e_i\}\ S_i\ \{q\}$ for $i = 1,\ldots,m$. By the induction hypothesis $Tr_J \vdash_{NT} \{p \wedge e_i\}\ S_i\ \{q\}$ for $i = 1,\ldots,m$ so by the if-rule II $Tr_J \vdash_{NT} \{p\}\ \underline{\text{if}}\ e_1 \rightarrow S_1 \ \square \ \ldots \ \square e_m \rightarrow S_m \underline{\text{fi}}\ \{q\}$.

Assume now $\models_{J,tot} \{r\}\ S\{q\}$ where $S \equiv \underline{\text{do}}\ e_1 \rightarrow S_1 \ \square \ldots \square e_m \rightarrow S_m \underline{\text{od}}$. Let n be a fresh variable. Let now C be the following set of states:

$$\begin{aligned}\mathrm{pret}_J(S,q) \cap \{\sigma : \models_J \mathrm{nat}(n)(\sigma) \wedge {} & \text{the longest computation} \\ & \text{starting in } \langle S,\sigma\rangle \text{ is of length } k+1, \\ & \text{where } k = \sigma(n)\}.\end{aligned}$$

Thus $\sigma \in C$ iff $\sigma(n)$ is a natural number, say k, such that all computations starting in $\langle S,\sigma\rangle$ properly terminate in a state satisfying q and the longest of these computations is of length k+1. It can be shown that there exists an assertion p(n) which defines C.

By the definition of p(n) we now have $\models_J p(n) \wedge n > 0 \rightarrow \bigvee_{i=1}^{m} e_i$, $\models_J p(0) \rightarrow \bigwedge_{i=1}^{m} \neg e_i$. Also it can be easily shown that $\models_J \{p(n) \wedge n > 0 \wedge e_i\}\ S_i\ \{\exists m < n\ p(m)\}$. By the induction hypothesis and the do-rule II we get $Tr_J \mid\!\frac{}{NT} \{\exists n\ p(n)\}\ \underline{\text{do}}\ e_1 \rightarrow S_1 \ \square \ \ldots \ \square e_m \rightarrow S_m \underline{\text{od}}\ \{p(0)\}$.

We now have by assumption $[r]_J \subseteq \mathrm{pret}_J(S,q)$ and so by virture of Lemma 1 $\models_J r \rightarrow \exists n\ p(n)$. Also $\models_J p(0) \rightarrow q$ holds so by the consequence rule we get $Tr_J \mid\!\frac{}{NT} \{r\}\ \underline{\text{do}}\ e_1 \rightarrow S_1 \ \square \ \ldots \ \square e_m \rightarrow S_m\ \underline{\text{od}}\ \{q\}$.

This concludes the proof.

4. COUNTABLE NONDETERMINISM

4.1. Bounded nondeterminism versus finite and countable nondeterminism

Up till now we have considered programs which allowed *bounded nondeterminism* only. By this we mean that for each pair $\langle S,\sigma\rangle$ where $S \in \mathcal{S}_n$ the set $\{\langle S_1, \sigma_1\rangle : \langle S, \sigma\rangle \rightarrow \langle S_1, \sigma_1\rangle\}$ is finite and moreover its cardinality is *bounded* by a constant dependent on S only. Informally it means that each program $S \in \mathcal{S}_n$ gives rise in one computation step to at most k different continuations where k depends on S only.

This property should be contrasted with that of *finite nondeterminism* which means that the above set is always finite but its cardinality does not depend on S only. An example of an instruction which leads to finite nondeterminism is $x:=? \leq y$ which sets to x a value smaller or equal to y. Such an instruction has been considered in FLOYD [9]. (Of course, we assume here that the programs are interpreted under a standard interpretation in natural numbers.)

It should be however noted that finite nondeterminism can be reduced to a bounded nondeterminism in the sense that $x:=? \leq y$ is equivalent to a program from $\mathcal{S}_n$. To see this take for example the program $b:=\underline{\text{true}}$; $x:=0$; $\underline{\text{do}}\ b \wedge x < y \rightarrow x:=x+1 \ \square\ b \wedge x < y \rightarrow b:=\underline{\text{false}}\ \underline{\text{od}}$. Consequently the study of finite nondeterminism (in the above sense) can be reduced to the study of bounded nondeterminism.

This is not any more the case with *countable nondeterminism*. By countable nondeterminism we mean that the above defined set can be countably infinite. An example of an instruction which leads to countable nondeterminism is the *random assigment* x:=? which sets to *x* an arbitrary nonnegative integer.

It is obvious how to define the semantics $M_{tot}[\![x:=?]\!]$ of x:=?. We have $\perp \notin M_{tot}[\![x:=?]\!](\sigma)$ for any σ. We now claim that there is no program $S \in S_n$ such that $M_{tot}[\![x:=?]\!] = M_{tot}[\![S]\!]$. This follows immediately from the following corollary to Lemma 1.

Corollary 1. *For any* $S \in S_n$ *and* σ *if* $\perp \notin M_{tot}[\![S]\!](\sigma)$ *then* $M_{tot}[\![S]\!](\sigma)$ *is a finite set.* □

Thus countable nondeterminism cannot be reduced to bounded (or finite) nondeterminism. This indicates that to study total correctness of programs allowing countable nondeterminism we have to develop essentially new proof rules, i.e. proof rules which cannot be derived from those of the proof system NT.

Note that this is not the case when dealing with the partial correctness of programs allowing countable nondeterminism as clearly

$$M[\![x:=?]\!] = M[\![b:=\underline{true}\ ;\ x:=0\ ;\ \underline{do}\ b \rightarrow x:=x+1 \ \square\ b \rightarrow b:=\underline{false}\ \underline{od}]\!]$$

(In this and the above considerations we ignored the fact that the value of b has been changed. It is easy to remedy this problem.)

Before we enter the proof theoretic considerations of countable nondeterminism we should perhaps explain why it is useful to study countable nondeterminism in the first place. First, the instruction x:=? can be viewed as another version of a more familiar read (x) instruction. Secondly, this instruction is particularly useful when dealing with the assumption of *fairness*, which will be discussed later. Also it allows to provide various neat characterizations of objects discussed in mathematical logic (see e.g. HAREL & KOZEN [12]).

4.2. A proof system for total correctness of countably nondeterministic programs

Consider now the class S_{cn} of programs which differs from S_n in that additionnally the instruction x:=? is allowed. We now present a proof system which allows us to prove total correctness of programs from S_{cn}. We add to the proof system NT the following axiom

AXIOM 9: random assignment axiom

$\{p\}x:=?\ \{p\}$

provided x is not free in p

and replace the *do*-rule II by the following generalization of it:

RULE 10: *do*-rule III

$$\frac{p(\alpha) \wedge \alpha > 0 \rightarrow \bigvee_{i=1}^{m} e_i,\ p(0) \rightarrow \bigwedge_{i=1}^{m} \neg e_i,\quad \{p(\alpha) \wedge \alpha > 0 \wedge e_i\}\ S_i\ \{\exists\beta < \alpha\ p(\beta)\},\ i = 1,\dots,m}{\{\exists\alpha\ p(\alpha)\}\ \underline{do}\ e_1 \rightarrow S_1 \,\square\dots\square\, e_m \rightarrow S_m\ \underline{od}\ \{p(0)\}}$$

where p(α) is an assertion with a free variable α which does not appear in the programs and ranges over ordinals.

Call the resulting proof system CNT.

4.3. An example of a proof in CNT

As an example proof in CNT consider now the following program:

```
S ≡ do x=0 → y:=? ; x:=1
      □ x ≠ 0 ∧ y > 0 → y:=y-1
    do.
```

We now wish to prove in CNT that S always terminates. More precisely, we prove in CNT the correctness formula {*true*} S {y=0}.

To this end we first specify the assertion language L. We assume that L contains the language of Peano arithmetic and has two sorts: *data* (for program data - here *integer*) and *ord* for ordinals. We assume a constant 0 of sort *ord* and a binary predicate symbol < over *ord*. The variables α, β are of sort *ord*, all other variables are of sort *data*.

In the course of the proof we shall have to convert values of sort *data* into values of sort *ord*. To this purpose we assume a one-argument conversion function $\bar{\cdot}$ of sort (*data*, *ord*) converting integers into ordinals and a constant ω of sort *ord*. We have $\forall x\, (\bar{x} < \omega)$ as by convention x is of type *data*.

Define $p(\alpha)$ by

$$p(\alpha) \equiv (x = 0 \rightarrow \alpha = \omega) \wedge (x \neq 0 \rightarrow \alpha = \bar{y}).$$

Intuitively speaking, for a state σ, $p(\alpha)(\sigma)$ holds if α is the smallest ordinal greater than or equal to the number of possible iterations performed by the loop when started in σ.

We now show that $p(\alpha)$ satisfies the premises of the *do*-rule III, i.e. that $p(\alpha)$ is a loop invariant.

1. We have $p(\alpha) \wedge \alpha > 0 \rightarrow x = 0 \vee y > 0 \rightarrow x = 0 \vee (x \neq 0 \wedge y > 0)$
2. We have $p(0) \rightarrow x \neq 0 \wedge y = 0 \rightarrow \neg(x = 0 \vee (x \neq 0 \wedge y > 0))$
3. We first show $\{p(\alpha) \wedge \alpha > 0 \wedge x = 0\}$ y:=? ; x:=1 $\{\exists \beta < \alpha\ p(\beta)\}$.

By the assignment axiom we have

$$\{\exists \beta < \alpha\ p(\beta)\ [1/x]\}\ x:=1\ \{\exists \beta < \alpha\ p(\beta)\}$$

so by the consequence rule

$$\{\forall y\ \exists \beta < \alpha\ p(\beta)[1/x]\}\ x:=1\ \{\exists \beta < \alpha\ p(\beta)\}.$$

By the random assignment axiom and the composition rule we now get

$$\{\forall y\ \exists \beta < \alpha\, p(\beta)\ [1/x]\}\ y:=?\ ;\ x:=1\ \{\exists \beta < \alpha\ p(\beta)\}$$

To complete the proof it now suffices to show that $p(\alpha) \wedge \alpha > 0 \wedge x = 0 \rightarrow \forall y \exists\, \beta < \alpha\ p(\beta)\ [1/x]$ is true. $p(\alpha) \wedge x = 0$ implies $\alpha = \omega$. So for any y put $\beta = \bar{y}$: then $\beta < \alpha$ and $p(\beta)[1/x]$ holds.

Next we show $\{p(\alpha) \wedge \alpha > 0 \wedge x \neq 0 \wedge y > 0\}$ y:=y-1 $\{\exists \beta < \alpha\ p(\beta)\}$. By the assignment axiom and the consequence rule it suffices to show that $p(\alpha) \wedge \alpha > 0 \wedge x \neq 0 \wedge y > 0 \rightarrow \exists \beta < \alpha\ p(\beta)\ [y-1/y]$ is true. We have

$$\begin{aligned} p(\alpha) \wedge \alpha > 0 \wedge x \neq 0 \wedge y > 0 &\rightarrow \alpha = \bar{y} \wedge y > 0 \wedge x \neq 0 \\ &\rightarrow \alpha = \bar{y} \wedge y > 0 \wedge p(\overline{y-1})[y-1/y] \\ &\rightarrow \exists \beta < \alpha\ p(\beta)[y-1/y] \end{aligned}$$

By the do-rule III we now get

$$\{\exists\alpha\ p(\alpha)\}\ S\ \{p(0)\}.$$

Clearly both $\exists\alpha\ p(\alpha)$ and $p(0) \rightarrow y = 0$ hold, so by the consequence rule $\{\underline{\text{true}}\}\ S\ \{y=0\}$ holds.

To be precise we actually proved $Tr_{J_1} \vdash_{CNT} \{\underline{\text{true}}\}\ S\ \{y=0\}$ where J_1 is a standard interpretation of the assertion language L.

4.4. Soundness and completeness of CNT

Before we dwell on the issue of soundness and completeness of CNT we have to specify for which assertion languages and their interpretations CNT is an appropriate proof system.

As in the previous section we assume that the assertion language L contains two sorts : data and ord. As before we have a constant 0 of type ord and a binary predicate symbol < over ord. Additionally we assume that L includes second order variables of arbitrary arity and sort. The second order variables can be bound only by the *least fixed point operator* μ provided the bound variable occurs positively in the considered formula. (Here a variable occurs *positively* in a formula if none of its occurrences in a disjunctive normal form of the formula is in the scope of a negation sign). Thus if the set variable a occurs positively in p(a) then μa.p is a well formed formula. The free variables of μa.p are those of p other than a.

An interpretation J for this type of assertion language is an ordinary two-sorted second order structure subject to the following five conditions

1. The domain $J_{\underline{data}}$ of sort data is countable (to ensure countable non-determinism,
2. The domain $J_{\underline{ord}}$ of sort ord is an initial segment of ordinals (to ensure a proper interpretation of the do-rule III),
3. The domain $J_{\underline{ord}}$ contains all countable ordinals (needed for the completeness proof),
4. The constant 0 denotes the least ordinal and the predicate symbol < denotes the strict ordering of the ordinals, restricted to $J_{\underline{ord}}$,
5. The domains of each of the set sorts contain all sets of the appropriate kind (to ensure the existence of the fixed points considered below).

The truth of the formulae of L under an interpretation J is defined in a standard way. The only nonstandard case is when a formula is of the form $\mu a.p$. We put then $\models_J \mu a.p$ iff $\models_J p[R/a]$ where R is the least fixed point of an operator naturally induced by p. Having defined the truth of the formulae of L we define the truth of the correctness formulae in the usual way.

The following theorem due to APT & PLOTKIN [3] explains why this type of assertion languages and their interpretation is of interest.

Theorem 2. *Let the assertion language* L *and its interpretation* J *satisfy the above stated conditions. Then for every correctness formula* ϕ $Tr_J \vdash_{CNT} \phi$ *iff* $\models_J \phi$.

This theorem states soundness and completeness of the proof system CNT. The soundness proof should hold for any reasonable assertion language ; it is the completeness proof which dictated the specific choice of the assertion language. The arguments used in the proofs are appropriate generalizations of those used in the soundness and completeness proofs of the system NT.

The use of ordinals in assertions requires perhaps a word of comment. It can be shown that ordinals are indeed necessary, i.e. the do-rule II is not sufficient here. For example we cannot prove the correctness formula considered in section 3.11 in a proof system in which the do-rule III is replaced by the do-rule II. In case when the assertion language L contains the language of Peano arithmetic and the domain of data values J_{data} is N, the set of natural numbers, we can exactly estimate which ordinals are needed for proofs in CNT. It turns out that exactly all *recursive ordinals* are needed. (By a recursive ordinal we mean here an ordinal attached in a natural way to a tree which can be coded by a recursive set. For equivalent characterizations see ROGERS [21].)

4.5. The issue of fairness

According to the usual semantics M_{tot} the program b:=true ; do b → skip □ b:=false od does not always terminate because the computation in which always the first guard is chosen is infinite. We can however imagine restricted forms of interpretation of programs from S_n under which the above program will always terminate.

One of such interpretations is the one under the assumption of *fairness*. In the context of programs from S_n this assumption states that in every infinite computation each guard which is infinitely often true is eventually chosen. Here a guard is true if it evaluates to true at the moment the control in the program is just before it.

This type of assumptions is particularly important when studying the behaviour of parallel programs in the context of which fairness is a most general modeling of the fact that the ratio of speeds between concurrent processors may be arbitrarily large and varying but always finite. Study of the hypothesis of fairness in the context of nondeterministic programs is partially motivated by the fact that parallel programs can be modelled by nondeterministic programs.

We now formally define the semantics of programs from $\mathcal{S}_n$ under the assumption of fairness. Let $\xi = \langle S_0, \sigma_0\rangle \rightarrow \langle S_1, \sigma_1\rangle \rightarrow \dots$ be an infinite computation starting in $\langle S_0, \sigma_0\rangle$. We say that ξ is *fair* if it fullfils the following two conditions:

i) for each program $S \equiv \underline{\text{if}}\ e_1 \rightarrow S_1 \ \square \dots \square\, e_m \rightarrow S_m\, \underline{\text{fi}}\ ;\ S'$ and each $i = 1,\dots,m$ if there are infinitely many j's for which $\langle S, \sigma_j\rangle$ appears in ξ and $\models_J e_i(\sigma_j)$, then there are infinitely many j's among them such that the transition $\langle S, \sigma_j\rangle \rightarrow \langle S_i\ ;\ S', \sigma_{j+1}\rangle$ appears in ξ,

ii) for each program $S \equiv \underline{\text{do}}\ e_1 \rightarrow S_1 \ \square \dots \square\, e_m \rightarrow S_m\, \underline{\text{od}}\ ;\ S'$ and each $i = 1,\dots,m$ if there are infinitely many j's for which $\langle S, \sigma_j\rangle$ appears in ξ and $\models_J e_i(\sigma_j)$, then there are infinitely many j's among them such that the transition $\langle S, \sigma_j\rangle \rightarrow \langle S_i\ ;\ S\ ;\ S', \sigma_{j+1}\rangle$ appears in ξ.

To avoid confusion resulting from the fact that various occurrences of S in ξ do not need to correspond with the same program, we should actually label each statement with a unique label. It is clear how to perform this process and we leave it to the reader.

We define the fair semantics for the programs from $\mathcal{S}_n$ by putting

$$\begin{aligned}\mathcal{M}_{fair}[\![S]\!](\sigma) = \mathcal{M}[\![S]\!](\sigma) &\cup \{\bot \mid \text{there exists a fair infinite computation} \\ &\qquad\quad \text{starting in } \langle S, \sigma\rangle\} \\ &\cup \{\underline{\text{fail}} \mid S \text{ can fail from } \sigma\}.\end{aligned}$$

Thus the difference between the semantics $\mathcal{M}_{tot}$ and $\mathcal{M}_{fair}$ lies in the treatment of infinite unfair computations. We assume that all finite computations are fair.

We now define the notion of total correctness of the programs considered under the assumption of fairness by putting

$$\models_{J,fair} \{p\}\ S\ \{q\} \text{ iff } \mathcal{M}_{fair}[\![S]\!]([p]_J) \subseteq [q]_J$$

where of course

$$\mathcal{M}_{fair}[\![S]\!]([p]_J) = \bigcup_{\sigma\in[p]_J} \mathcal{M}_{fair}[\![S]\!](\sigma).$$

If $\models_{J,fair} \{p\}\ S\ \{q\}$ holds then we say that $\{p\}\ S\ \{q\}$ holds under the assumption of fairness (w.r.t. J). Thus $\models_{J,fair} \{p\}\ S\ \{q\}$ holds iff each fair computation sequence of S starting in a state satisfying p successfully terminates and the terminating state satisfies q.

4.6. A transformation realizing fairness

We now wish to present a proof system in which total correctness under the assumption of fairness can be proved. For didactic resons instead of presenting the proof rules immediately, we rather explain how to derive them. To this purpose we first provide a transformation of a program $S \in \mathcal{S}_n$ into a program $S_{fair} \in \mathcal{S}_{cn}$ which realizes the assumption of fairness in the sense that S_{fair} generates exactly all fair computations of S. We proceed by the following successive steps:

1. replace each subprogram $\underline{do}\ e_1 \rightarrow S_1 \square \dots \square e_m \rightarrow S_m\ \underline{od}$ of S by

$$\underline{do} \bigvee_{i=1}^{m} e_i \rightarrow \underline{if}\ e_1 \rightarrow S_1 \square \dots \square e_m \rightarrow S_m\ \underline{fi}\ \underline{od},$$

2. replace each subprogram $\underline{if}\ e_1 \rightarrow S_1 \square \dots \square e_m \rightarrow S_m\ \underline{fi}$ of S by the following subprogram

$$\underline{for}\ j:=1\ \underline{to}\ m\ \underline{if}\ e_j\ \underline{then}\ z_j:=z_j-1\ ;$$

$$\underline{if}\ e_1 \wedge z_1 = 0 \wedge \forall_j z_j \geq 0 \rightarrow z_1:=?\ ;\ S_1$$
$$\square \dots \square e_m \wedge z_m = 0 \wedge \forall_j z_j \geq 0 \rightarrow z_m:=?\ ;\ S_m \underline{fi},$$

3. Rename all variables $z_1,\dots,z_m$ appropriately so that each $\underline{if}$-construct has its "own" set of these variables.

Strictly speaking the program S_{fair} does not belong to $\mathcal{S}_{cn}$ as the $\underline{if}$-$\underline{then}$ and the $\underline{for}$-constructs are not assumed in the syntax. It is however clear how to change it here into a sequence of the $\underline{if}$-constructs. Note that in step 1 we replaced each subprogram of S of the form of a $\underline{do}$-loop by another subprogram which is equivalent to the original one *in the sense* of the M_{fair} semantics.

Let us call the subprograms introduced in step 2 the $\underline{if}_{fair}$-constructs. The above transformation boils down to building into all $\underline{if}$-constructs of S a fair scheduler in which the auxiliary variables z_i count down to a moment when the corresponding guard is selected.

The following lemma relates S to S_{fair}.

Lemma 2.

a) *If* ξ *is a fair non failing computation of* S *then an extension* ξ' *of* ξ *dealing with the auxiliary variables of* S_{fair} *is a non failing computation of* S_{fair}.

b) *If* ξ *is a non failing computation of* S_{fair} *then its restriction to the computation steps dealing with* S *is a fair non failing computation of* S.

Proof

a) We annotate the states in ξ by assigning in each of them values to all variables z_i. Given a state σ_j there are two cases.

Case I. For no state $\sigma_k (k > j)$ the guard corresponding with z_i has been chosen.

Then by the assumption of fairness this guard has been only finitely many times enabled in case the control was there. We put $\sigma_j(z_i)$ to be equal 1 + the number of times the guard will still be enabled whenever the control will be there.

Case II. For some state $\sigma_k (k > j)$ the guard corresponding with z_i has been chosen. We put $\sigma_j(z_i)$ to be equal 1 + the number of times the guard will still be enabled and not chosen whenever the control will be there.

b) By the construction of S_{fair} the restriction of ξ to the computation steps dealing with S is a computation sequence for S. Suppose that this restriction is not a fair computation sequence. Then behind some point in this computation a guard would be infinitely many times enabled at the moment a control is there and yet never chosen. By the construction of S_{fair} the variable z_i corresponding with this guard would become arbitrarily small. This is however impossible because as soon as it becomes negative a failure will arise. □

Corollary 2. *Suppose that none of the auxiliary variables introduced in* S_{fair} *occurs free in the assertions* p *and* q. *Then*

$$\models_{J,fair} \{p\}\ S\ \{q\} \text{ iff } \forall\sigma\ [\models_J p(\sigma) \rightarrow S \text{ cannot fail from } \sigma]$$

$$\text{and } \models_{J,weak}\{p\}\ S_{fair}\ \{q\}. \quad \Box$$

Here $\models_{J,weak}\{p\}\ S_{fair}\{q\}$ holds if in the definition of the semantics $M_{tot}[\![S_{fair}]\!]$ of S we drop any mentioning of failure. We then say that S_{fair} is *weakly totally correct under* J with respect to p and q.

4.7. A proof system dealing with fairness

The above corollary indicates that in order to prove total correctness of S under the assumption of fairness it is sufficient to prove weak total correctness of S_{fair} provided the absence of failure in S can be established. To prove weak total correctness of S_{fair} we can use the proof system CNT defined in section 4.2 in which the if-rule II is replaced by the original if-rule in order to ignore the possibility of failures. Call this system CWT.

Assume now for a moment that only deterministic do-loops are allowed, i.e. do-loops of the form do e → S od. Then the first step in the transformition discussed in the previous section is not needed and can be deleted. Now, due to the form of S_{fair} any proof of its weak total correctness can be transformed into a direct proof of S provided we use the following transformed version of the if-rule:

$$\frac{\{p\}\ \underline{\text{for}}\ j:=1\ \underline{\text{to}}\ m\ \underline{\text{if}}\ e_j\ \underline{\text{then}}\ z_j:=z_j-1\ \{p'\}, \qquad \{\ p' \wedge e_i \wedge z_i = 0 \wedge \bar{z} \geq 0\}\ z_i:=?\ ;\ S_i\{q\}\quad i = 1,\dots,m}{\{p\}\ \underline{\text{if}}\ e_1 \rightarrow S_1 \ \Box\ \dots\ \Box e_m \rightarrow S_m\ \underline{\text{fi}}\ \{q\}}$$

Indeed, by applying the above rule we replace systematically each $\underline{\text{if}}_{fair}$-subprogram of S_{fair} by the original if-subprogram of S; thus, in effect we obtain a direct proof of S. The above rule can be simplified if we "absorb" all assignments to auxiliary variables into the assertion p. In such a way we obtain a proof rule dealing exclusively with the if-construct and its components.

The last issue to be dealt with is that of freedom of failure which has to be dealt with according to Corollary 2. This problem can be taken care of in the same way as in section 3.7 of by simply adding to the premises of the fair if-rule the assertion $p \rightarrow \bigvee_{i=1}^{m} e_i$.

Summerizing, the final version of the rule has the following form

RULE 11: fair if-rule

$$\frac{\begin{array}{l} p \rightarrow \bigvee_{i=1}^{m} e_i, \\ \{p\ [\underline{\text{if}}\ e_j\ \underline{\text{then}}\ z_j+1\ \underline{\text{else}}\ z_j/z_j]_{j \neq i}[1/z_i] \wedge e_i \wedge \bar{z} \geq 0\}\ S_i\ \{q\}_{i=1,\dots,m} \end{array}}{\{p\}\ \underline{\text{if}}\ e_1 \rightarrow S_1 \ \square \ \dots \ \square e_m \rightarrow S_m\ \underline{\text{fi}}\ \{q\}}$$

We still have to deal with the problem of do-loops as we assumed above that only deterministic loops are allowed. For this purpose we have to go back to the transformation from the previous section. In step 1 we replaced each do-loop by a program equivalent to it in the sense of the M_{fair} semantics. Therefore a proof of total correctness under the assumption of fairness of the latter program constitutes a proof of total correctness under the assumption of fairness of the former one. Thanks to this observation we can derive the fair do-rule. It has the following form after some simplifications:

RULE 12 : fair do-rule

$$\frac{\begin{array}{l} \qquad p(\alpha) \wedge \alpha > 0 \rightarrow \bigvee_{i=1}^{m} e_i,\ p(0) \rightarrow \bigwedge_{i=1}^{m} \neg e_i, \\ \{p(\alpha)\ [\text{if}\ e_j\ \underline{\text{then}}\ z_j+1\ \underline{\text{else}}\ z_j/z_j]_{j \neq i}[1/z_i] \wedge \alpha > 0 \wedge e_i \wedge \bar{z} \geq 0\} \\ S_i \\ \{\exists \beta < \alpha\ p(\beta)\}_{i=1,\dots,m} \end{array}}{\{\exists \alpha\ p(\alpha)\}\ \underline{\text{do}}\ e_1 \rightarrow S_1 \ \square \ \dots \ \square e_m \rightarrow S_m\ \underline{\text{od}}\ \{p(0)\}}$$

The assertion $p(\alpha)$ satisfies the same condition as in rule 10.

Summarizing, the proof system FN for total correctness of programs from $\mathcal{S}_n$ under the assumption of fairness is obtained from the proof system N by replacing the if and do-rule by the proof rules introduced above. Note that the random assignment axiom is not needed – we used it only to derive the final form of the new rules.

The only pupose of intoducing the transformation S into S_{fair} was to derive the new rules in a straightforward way. These rules deal with the *original* programs and not their transformed versions.

4.8. Soundness and completeness of FN

The following lemma provides a proof theoretic counterpart of Corollary 2.

Lemma 3. *Suppose that none of the auxiliary variables introduced in* S_{fair} *occurs free in the assertions* p *and* q. *Then*

$$\mathrm{Tr}_J \vdash_{\overline{FN}} \{p\}\ S\ \{q\} \text{ iff } \mathrm{Tr}_J \vdash_{\overline{CWT}} \{p\}\ S_{fair}\ \{q\} \text{ and}$$

$$\forall\sigma\ [\models_J p(\sigma) \rightarrow S \text{ cannot fail from } \sigma]. \quad \Box$$

This lemma can be easily justified on the basis of remarks provided in the previous section while introducing the new proof rules.

Lemma 3 together with Corollary 2 reduces the question of soundness and completeness of FN to that of CWT. But the latter system is clearly sound and complete in the sense of section 4.4. This shows that the proof system FN is also sound and complete in the same sense. We have only to restrict additionally the class of allowed structures to those which in their data domain contain natural numbers.

4.9. An example of a proof in FN

We conclude the discussion of fairness by presenting an example proof in FN. Consider the following program S:

```
do x > 0 → if true → if b → x:=x-1
                     □ b → b:=false
                     □¬b → skip fi
           □ true → b:=true fi
od
```

We want to prove $\models_{J_0,fair}$ {true} S {true}, i.e. that S always terminates under the assumption of fairness.

To this purpose we have to find an assertion $p(\alpha)$ such that

(14) $\quad p(\alpha) \wedge \alpha > 0 \rightarrow x > 0$

(15) $\quad p(0) \rightarrow x \leq 0$

(16) $\quad \exists\, \alpha\ p(\alpha)$

and

(17) $\quad \{p(\alpha) \wedge \alpha > 0 \wedge x > 0\}\ S'\ \{\exists\, \beta < \alpha\ p(\beta)\}$

where S' is the body of the do-loop. (Note that we use here the original do-rule (rule 10) as the do-loop in question is deterministic. It is easy to see that the do-rules 10 and 12 are equivalent in the case of deterministic do-loops.)

Let $\rho(a,b,c,d) = \omega^3.a + \omega^2.b + \omega.c + d$ for any integers a,b,c,d where $a > 0$. Then $\rho(a,b,c,d)$ is an ordinal. We define

$$p(\alpha) \equiv \alpha = \underline{\text{if}}\ x > 0\ \underline{\text{then}}\ \rho(x, z_3, 1-b,\ b \rightarrow z_1, z_2)$$
$$\underline{\text{else}}\ 0.$$

In the expression $1 - b$, true is interpreted as 1, false as 0 ; $b \rightarrow z_1, z_2$ stands for if b then z_1 else z_2 ; the auxiliary variables z_1 and z_2 are associated with the outer guards and z_3, z_4 and z_5 with the inner guards, respectively.

It is clear that (14) - (16) hold. To prove (17) we have to insure that in a fair computation the value of ρ decreases on each iteration of the loop. More formally we wish to apply the fair if-rule so we have first to prove the premises

(18) $\quad \{(p(\alpha) \wedge \alpha > 0 \wedge x > 0)\ [z_2+1/z_2][1/z_1] \wedge z_1, z_2 \geq 0\}\ S_1\ \{\exists \beta < \alpha\ p(\beta)\}$

and

(19) $\quad \{p(\alpha) \wedge \alpha > 0 \wedge x > 0)\ [z_1+1/z_1][1/z_2] \wedge z_1, z_2 \geq 0\}\ b := \underline{\text{true}}\ \{\exists \beta < \alpha\ p(\beta)\}$

as the first premise of the fair if-rule is obviously satisfied. Here

$$
\begin{aligned}
S_1 \equiv \underline{\text{if}}\ & b \to x:=x-1\\
\square\ & b \to b:=\underline{\text{false}}\\
\square\ & \neg b \to \underline{\text{skip}}\ \underline{\text{fi}}.
\end{aligned}
$$

To prove (18) we once again wish to apply the fair if-rule. The premises to prove are

$$\{p_1[b \to z_4+1, z_4/z_4][\neg b \to z_5+1, z_5/z_5][1/z_3] \wedge b \wedge z_3, z_4, z_5 \geq 0\}\ x:=x-1\ \{]\ \beta \to \alpha\ p(\beta)\} \tag{20}$$

$$\{p_1[b \to z_3+1, z_3/z_3][\neg b \to z_5+1, z_5/z_5][1/z_4] \wedge b \wedge z_3, z_4, z_5 \geq 0\}\ b:=\underline{\text{false}}\ \{]\ \beta \to \alpha\ p(\beta)\} \tag{21}$$

and

$$\{p_1[b \to z_i+1, z_i/z_i]_{i=3,4}[1/z_5] \wedge \neg b \wedge z_3, z_4, z_5 \geq 0\}\ \underline{\text{skip}}\ \{]\beta < \alpha\ p(\beta)\} \tag{22}$$

where

$$p_1 \equiv (p(\alpha) \wedge \alpha > 0 \wedge x > 0)[z_2+1/z_2][1/z_1] \wedge z_1, z_2 \geq 0.$$

Note that the pre-assertion of (20) is equivalent to $\rho(x,1,0,1) = \alpha \wedge b \wedge x > 0 \wedge \bar{z} \geq 0$.

We have by the assignment axiom

$$
\begin{aligned}
&\{\rho(x,1,0,1) = \alpha \wedge b \wedge x > 0 \wedge \bar{z} \geq 0\}\\
&x:=x-1\\
&\{(\rho(x+1,1,0,1) = \alpha \wedge b \wedge x > 0 \wedge \bar{z} \geq 0) \vee \rho(0)\}
\end{aligned}
$$

which implies by the consequence rule (20) as the necessary implication is clearly true.

To prove (21) note that the pre-assertion of (21) is equivalent to

$$\rho(x, z_3+1, 0, 1) = \alpha \wedge \alpha > 0 \wedge b \wedge \bar{z} \geq 0 \wedge x > 0$$

which in turn implies the assertion

$$q \equiv \exists\beta < \alpha(x > 0 \wedge \bar{z} \geq 0 \wedge \beta = \rho(x, z_3, 1, z_2)).$$

Now by the assignment axiom and the consequence rule

$$\{q\}\ b{:=}\underline{\text{false}}\{\exists\beta < \alpha\ p(\beta)\}$$

so (21) by the consequence rule.

Finally, to prove (22) we note that

$$p_1[b \rightarrow z_1+1,\ z_i/z_i]_{i=3,4}[1/z_5] \wedge \neg b \wedge z_3, z_4, z_5 \geq 0$$

implies

$$\rho(x, z_3, 1, z_2+1) = \alpha \wedge \neg b \wedge \bar{z} \geq 0 \wedge x > 0$$

which in turn implies $\exists\beta < \alpha\ p(\beta)$. Hence (22) holds by the skip axiom.

Now, from (20) - (22) we get (18) by the fair if-rule.

To prove (17) note that the pre-assertion of (19) is equivalent to

$$\rho(x, z_3, 1-b, b \rightarrow z_1+1, 1) = \alpha \wedge \alpha > 0 \wedge x > 0 \wedge \bar{z} \geq 0$$

which in turn implies the assertion

$$r \equiv \exists\beta < \alpha(\rho(x, z_3, 0, z_1+1) = \beta \wedge x > 0 \wedge \bar{z} \geq 0).$$

Now by the assignment axiom and the consequence rule $\{r\}\ b{:=}\underline{\text{true}}\ \{\exists\beta < \alpha\ p(\beta)\}$ so (19) by the consequence rule.

We now proved both (18) and (19) and we get (17) by the fair if-rule. (14) - (17) imply by the do-rule $\{\underline{\text{true}}\}\ S\ \{\underline{\text{true}}\}$ so by virtue of the soundness of the system FNT we get $\models_{J_0,\text{fair}}\{\underline{\text{true}}\}\ S\ \{\underline{\text{true}}\}$. This concludes the proof.

4.10 The issue of justice

Another possible restricted interpretation of nondeterministic programs is the one under the assumption of justice. In the context of programs from $\mathcal{S}_n$ this assumption states that in every infinite computation each guard which is true from some moment on is eventually chosen. Here, as before, a guard is true if it evaluates to true at the moment the control in the program is just before it.

The assumption of *justice* can be treated in an analogous way as that of fairness. To obtain a transformation realizing justice we only need to replace in the transformation from section 4.6. the program from the first line in step 2 by

$$\underline{\text{for}}\ j:=1\ \underline{\text{to}}\ m\ \underline{\text{if}}\ e_j \rightarrow z_j:=z_j-1 \ \square\ \neg e_j \rightarrow z_j:=?\ \underline{\text{fi}}.$$

All other steps in the development of the proof rules for justice are the same as before and left to the reader.

As a final remark we would like to indicate that in the transformation from section 4.6 we can omit the conditions $z_i=0$ from all of the guards, both for the case of fairness and justice. Clearly various other transformation also satisfy lemma 2. We chose here a transformation which leads to simplest proof rules dealing with fairness or justice.

5. BIBLIOGRAPHICAL REMARKS

The first treatment of nondeterminism in the framework of Hoare's logic is due to LAUER [15] where a proof rule dealing with the or-construct (the meaning of the construct S_1 or S_2 is execute either S_1 or S_2) is introduced. Correctness of nondeterministic programs introduced in section 3 is extensively studied in DIJKSTRA [8] using a different approach. Axioms 1,2 and proof rules 3,6 are from HOARE [14]. Rules 4,5 are obvious modifications of the appropriate rules dealing with the deterministic versions of the constructs and introduced in LAUER [15] and HOARE [14], respectively. They appear for example in DE BAKKER [5] (p. 292).

Soundness and completeness proofs from sections 3.5 and 3.6 are straightforward generalizations of the corresponding proofs dealing with deterministic versions of the programs and presented for example in DE BAKKER [5] (section 3). Rule 7 is inspired by the discussion of clean behaviour of programs in PNUELI [19]. The completeness proof from section 3.8 is an appropriate modification of a corresponding proof from HAREL [11].

The notion of bounded nondeterminism is introduced in DIJKSTRA [8]. Countable nondeterminism is extensively studied in APT & PLOTKIN [3] and several related references can be found there. Corollary 1 is implicit in DIJKSTRA [8]. Axiom 9 is from HAREL [11] and rule 10 from APT & PLOTKIN [3] where a slightly different syntax is used. Sections 4.3 and 4.4 are based on APT & PLOTKIN [3], as well. The program from section 4.3 is from DIJKSTRA [8].

The issue of fairness is discussed in several papers (see for example PNUELI [19]). First proof rules dealing with fairness were proposed in GRÜMBERG et al. [10], LEHMANN et al. [17] and APT & OLDEROG [2]. LEHMANN [16] contains a simplified completeness proof of a rule introduced in GRÜMBERG et al. [10]. Sections 4.7 - 4.10 are based on APT et al. [4]. Transformations realizing fairness were first introduced in APT & OLDEROG [2]. Simplified versions of such transformations are given and discussed in PARK [18].

The program studied in section 4.9 is due to S. Katz. First proof rules dealing with justice were proposed in APT & OLDEROG [3] and LEHMANN et al. [17]. LEHMANN [16] contains another proof rule for justice. In LEHMANN et al. [17] arguments for introducing the hypotheses of justice and fairness when studying parallel programs are given. QUEILLE & SIFAKIS [20] contains a thorough discussion of various possible formalizations of the assumption of fairness.

REFERENCES

[1] APT, K.R., *Ten Years of Hoare's Logic, a survey, part I,* TOPLAS, vol. 3, N° 4, pp. 431-483 (1981).

[2] APT, K.R. & E.-R. OLDEROG, *Proof rules dealing with fairness,* Bericht Nr. 8104, Inst. Inf. Prakt. Math., University of Kiel, (1981). (Extended abstract appeared in: Logic of Programs, Lecture Notes in Computer Science, 131, pp. 1-8, Springer-Verlag, New York, (1982).)

[3] APT, K.R. & G.D. PLOTKIN, *Countable nondeterminism and random assignment,* Technical Report 82-7, LITP, Université Paris 7 (1982). (Extended abstract appeared as : A Cook's tour of countable nondeterminism, in : Proceedings ICALP'81, Lecture Notes in Computer Science, 115, pp. 479-494, Springer Verlag, Berlin, (1981).)

[4] APT, K.R., A. PNUELI & J. STAVI, *Fair termination revisited - with delay,* in: Proc. 2nd Conference FST and TCS, Bangalore, India, pp. 146-170 (1982).

[5] DE BAKKER, J.W., *Mathematical theory of program correctness,* Prentice-Hall, Englewood Cliffs, (1980).

[6] COOK, S.A., *Soundness and completeness of an axiom system for program verification*, SIAM J. Comput., vol. 7, n° 1, pp. 70-90 (1978).

[7] DIJKSTRA, E.W., *Guarded commands, nondeterminacy and formal derivation of programs*, Communications ACM, vol. 18, N° 8 (1975).

[8] DIJKSTRA, E.W., *A discipline of programming*, Prentice-Hall Englewood Cliffs, (1976).

[9] FLOYD, R.W., *Nondeterministic algorithms*, Journal ACM, vol. 14, N° 4, pp. 636-644 (1967).

[10] GRÜMBERG, O., N. FRANCEZ, J.A. MAKOWSKY & W.P. de ROEVER, *A proof rule for fair termination of guarded commands, in : Algorithmic languages* (Eds, J.W. de BAKKER, J.C. van VLIET), pp. 399-416, IFIP, North Holland, Amsterdam (1981).

[11] HAREL, D., *First-order dynamic logic*, Lecture Notes in Computer Science, 68, Springer Verlag, New York (1979).

[12] HAREL, D. & D. KOZEN, *A programming language for the inductive sets, and applications*, in : Proceedings ICALP'82, Lecture Notes in Computer Science, 140, Springer-Verlag, Berlin

[13] HENNESSY, M.C.B. & G.D. PLOTKIN, *Full abstraction for a simple programming language*, in : Proc. 8th Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, 74, pp. 108-120, Springer Verlag, New-York (1979).

[14] HOARE, C.A.R., *An axiomatic basis for computer programming*, Communications ACM, vol. 12, N° 10, pp. 576-580, 583 (1969).

[15] LAUER, P.E., *Consistent formal theories of the semantics of programming languages*, Technical Report TR. 25.121, IBM Lab. Vienna, (1971).

[16] LEHMANN, D., *Another proof for the completeness of a rule for the fair termination of guarded commands and another rule for their just termination*, Technical Report IW 178/81, Mathematisch Centrum, (1981).

[17] LEHMANN, D., PNUELI, A. & J. STAVI, *Impartiality, justice and fairness: the ethics of concurrent termination*, Proceedings ICALP'81, Lecture Notes in Computer Science, 115, pp. 264-277, Springer-Verlag, Berlin, (1981).

[18] PARK, D., *A predicate transformer for weak fair iteration, in :* Proceedings 6th IBM Symposium on Mathematical Foundations of Computer Science, Hakone, Japan (1981).

[19] PNUELI, A., *The temporal semantics of concurrent programs,* Theoretical Computer Science, vol. 13, N° 1, pp. 45-60, (1981).

[20] QUEILLE, J.P. & J. SIFAKIS, *Fairness and related properties in transition systems - a time logic to deal with fairness,* Technical Report - RR N° 292, University of Grenoble (1982).

[21] ROGERS, H. Jr, *Theory of recursive functions and effective computability,* McGraw-Hill, New York, (1967).

THE "FAIRNESS" PROBLEM AND NONDETERMINISTIC COMPUTING NETWORKS

D. Park
Warwick University, Coventry, England

The existence of a "fairness problem" has been recognised for some years now. It accounts for the most important respect in which the formal description of programming languages involving parallelism can be expected to be inadequate. In its simplest form the problem arises in trying to give a reasonable mathematical account of what can be expected of computing networks. A 'node' in such a network can be expected to produce a sequence of outputs determined by the sequences of inputs which it receives. If the node operates asynchronously, its output can be expected to be a function (in the mathematical sense) of its input -- *but with more than one input it is a function also of the relative orders in which these inputs arrive.* A simple example of such a node is the 'merge' node with two inputs, which outputs any message it receives as soon as it arrives, into one output line. Naively, the implementation of such a node seems trivial -- connect all three lines together (as electronics,of course, there is more to it than this). What is computed is "nondeterministic" unless the information on relative timing is known -- so that networks involving such components should be regarded as computing relations rather than functions. Actually, if we can give a reasonable account of networks whose only nondeterminism arises from such merge nodes, we will have solved the more general problem-- so long as the nondeterminism arises only from relative timing of input sequences, there is an equivalent network whose only nondeterministic nodes are merges (for each node R with n inputs, produce a corresponding node T(R) of one input, consisting of inputs to R "tagged" with the input line on which they occur; replace R by the composition of T(R) with a merge of n inputs -- using n - 1 merge nodes, say; the merged inputs are outputs from n nodes, each of which "tags" its inputs with the line on which it acts. The resulting network of 2n nodes is equivalent to R, given an appropriate design for T(R)).

The nondeterminism arises from what we are accustomed to leaving unspecified -- the precise times which computations and transmissions of messages take. The "fairness" difficulty arises from an extreme position as regards this convention -- that the times taken are not only unknown but also unbounded. Computations of nodes all terminate, where specified, but nothing is known about relative speeds. There is an inherent connection with the notion of "unbounded nondeterminism", which we will have discuss.

It will become clear that a mathematical analysis of such networks must introduce some "time-like" features, and that this will detract from the abstractness of the model to be used. We look at some examples which force this point home. The most important can be regarded as serious anomalies, which prevent a "naive" extension of ideas which work well for deterministic networks. We look at what seems a reasonable way to cope with these problems, while remaining as abstract as possible. The result is an extended "Kahn principle" for networks composed of nondeterministic 'merge' nodes as well as arbitrary deterministic nodes. This treatment is related to that of Boussinot [1], though it has been simplified by adopting the "hiatonization" notion suggested by Wadge following his [7]. The emphasis on "angelicness" is largely inherited from Broy [3]. From the remark above, the extended principle has wide application -- to arbitrary networks subject to nondeterminism because of unspecified computation/ transmission times.

1. DETERMINISTIC BACKGROUND

The analysis for deterministic networks of Kahn [4] is an elegant application of elementary domain theory. The notions of approximation, monotonicity, continuity, fixpointing provide elegant proof rules when networks composed only of deterministic nodes are analyzed using them. Recall the basic notions:

SEQUENCES: for any Σ, Σ^∞ consists of all finite and infinite sequences from Σ. The null sequence if written

$$\perp$$

APPROXIMATION: given u, v $\in \Sigma^\infty$,

$$u \sqsubseteq v$$

if u is an initial segment of v. For vectors of sequences

$$<u_1,\dots,u_n> \sqsubseteq <v_1,\dots,v_n>$$

if $u_i \sqsubseteq v_i$, each i. These specify orderings on each $(\Sigma^\infty)^n$.

LIMITS: the partial orderings are "sequentially complete". Any sequence

$$u_1 \sqsubseteq u_2 \sqsubseteq \dots$$

has a least upper bound $\bigsqcup_i u_i$.

MONOTONICITY: f(u) is monotone if

$$u \sqsubseteq v \Rightarrow f(u) \sqsubseteq f(v)$$

CONTINUITY: f(u) is continuous if, whenever

$$u_1 \sqsubseteq u_2 \sqsubseteq \dots$$

then

$$f(\bigsqcup_i u_i) = \bigsqcup_i f(u_i)$$

(This is a weaker constraint than is required for some purposes -- but strong enough to guarantee that the usual fixpoint formula

$$Y(f) = \bigsqcup_i f^i(\perp)$$

is meaningful, and satisfies $f(Y(f)) = Y(f)$, for continuous f.)

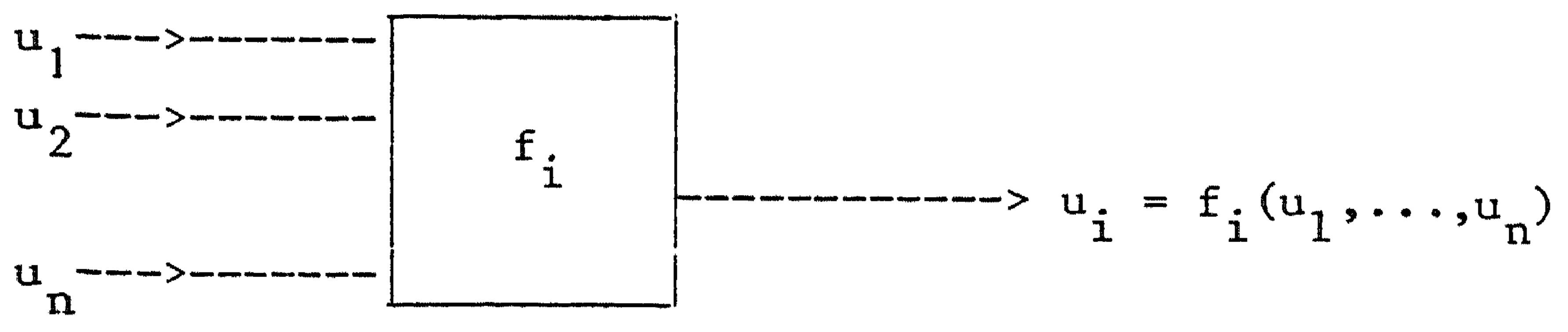
KAHN PRINCIPLE: The behaviour of a deterministic network with n communication lines can be obtained as the minimal fixpoint Y(f) of an appropriate continuous

$$f\colon (\Sigma^\infty)^n \to (\Sigma^\infty)^n$$

where Σ is an appropriate set of communication "tokens".

$$Y(f) = \bigsqcup_i f^i(\perp,\dots,\perp)$$

The appropriate function f is obtained by combining together the n continuous functions f_i, each of which specifies how the history of u_i, considered as the output from some node, depends on histories of inputs to that node.



It is important, and intuitively plausible, that the functions f_i which correspond to actual computing components should be monotone and continuous. The "behaviour" of the whole network is an n-tuple of histories, and this can be characterized neatly as the minimal fixpoint (in the lattice theory sense) of the function f, which combines the specifications of all nodes of the network.

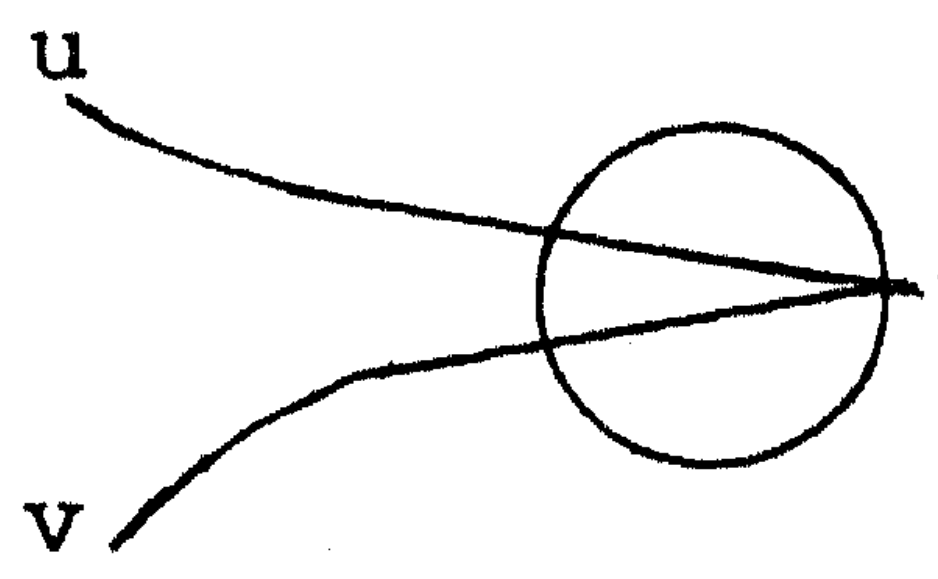
2. FAIRNESS, ANGELICNESS, ORACLES

While "nondeterministic" nodes can be analyzed in other terms, they are made most immediately tractable by the introduction of "oracles". Nondeterminism, in this sense, is determinism, but with some missing parameter -- its "oracle". Such a node has a history function, which now produces a set of possible output histories, indexed by an "oracle set" Δ:

$$R_i(u_1,\ldots,u_n) = \{f_i(u_1,\ldots,u_n,\delta) \mid \delta \in \Delta\}$$

Remembering the deterministic analysis, we of course expect the functions f_i to be monotone continuous on the sequence domains (and on the oracle domain, too, given appropriate structure on Δ).

Almost all problems arise as soon as one introduces the 'merge' operation, whose behaviour is to transmit communications from two input lines onto one output line -- interleaving them nondeterministically.



$$\text{merge}(u,v) = \{\text{dmerge}(u,v,\delta) \mid \delta \in \Delta\}$$

Intuitively, the requirement of FAIRNESS is just the requirement that every communication in u and v is eventually transmitted through the node, and appears in the output. Actually, our analysis of the notion will depend on a further distinction:
∞-FAIRNESS: if u is infinite, all of v is transmitted (and vice versa).
ANGELICNESS: if u is finite, all of v is transmitted (and vice versa).

Both are varieties of the general "fairness" requirement -- whatever u looks like, all of v should get through. But they are actually independent requirements. The reader might like to reflect on the 4 variants of merge which correspond to varieties of "scheduler" occupying the merge node; we expect the ∞-fair, angelic variety; but scheduling could be angelic and not ∞-fair -- by guaranteeing to inspect one input line only when starved of input on the other; and it could be ∞-fair but not angelic -- by allowing itself to be starved indefinitely by absence of input on only one of its input lines. (This insight can be made rigorous in terms of the concepts introduced below.)

To see the implications of these constraints, look first at the details of dmerge(u,v,δ) -- the merge of u, v with oracle δ. δ is to be a sequence of 0's and 1's, specifying the order in which lines are to be inspected. Using a 'dot' notation a.u for prefixing and/or concatenation, the following are (implicitly) recursion equations for dmerge:

$$\mathrm{dmerge}(\bot,v,0.\delta) = \mathrm{dmerge}(u,\bot,1.\delta) = \bot$$

$$\mathrm{dmerge}(a.u,v,0.\delta) = \mathrm{dmerge}(u,a.v,1.\delta) = a.\mathrm{dmerge}(u,v,\delta)$$

COPING WITH ∞-FAIRNESS: take the range Δ for the oracle δ to be the set of "fair oracles" -- all sequences with an infinite number of both 0's and 1's.

3. IS MERGE MONOTONE?

The 'dmerge' above is, apparently, not angelic. We would need dmerge($\bot$,u,δ) = dmerge(u,$\bot$,δ) = u for angelicness. But the dmerge would be non-monotone. For example

$$\bot \sqsubseteq a$$

but dmerge($\bot$,b...,0.δ) = b... $\not\sqsubseteq$ dmerge(a,b...,0.δ) = a.b... (assume a, b distinct).

COPING WITH ANGELICNESS I: Distinguish "terminated" from "unterminated" finite strings, by adjoining a special token #, occurring only as a final symbol. A terminated finite sequence is to be maximal:

$$u\# \sqsubseteq v \text{ iff } u\# = v$$

With this distinction, angelicness is satisfied -- but with "terminated finite" replacing "finite". 'dmerge' needs more definition, of course,

$$\text{dmerge}(\#,u,\delta) = \text{dmerge}(u,\#,\delta) = u$$

With this addition, and the above assumption about the approximation relation on terminated sequences $u\#$, 'dmerge' can be seen to be monotone continuous.

4. IS MERGE COMPUTABLE?

Actually, the necessity to use the set Δ of fair oracles is a source of embarrassment. The set is "constructive" or "computable" only in an unconventional sense. Neither Δ not its complement can be generated or recognized in any satisfactory "effective" way. But it is obvious that Δ is essential to a treatment of fairness. Even if it is not used to specify the ∞-fair merge, it can be obtained from the specification by

$$\Delta = \text{merge}(0^\omega,1^\omega) \quad (a^\omega \text{ is the infinite sequence of a's})$$

But this must throw as much doubt on the "computability" of 'merge' as on the computability of Δ.

To resolve these doubts it is necessary to reflect on the notions of "computability" which are appropriate, recalling that 'merge' is inherently nondeterministic, and therefore "relational" rather than "functional". While computability of functions between domains is well-understood, the general notion appropriate to relations between domains is elusive. One can agree that a relation between domains D and E is to be understood as a function from D to a "powerdomain" of E, and that the computable relations are then just those which turn into computable functions -- but there is a variety of powerdomain costructions, and in all those that are known the distinctions necessary to fairness etc.

disappear. Specifically, with $E = \{0,1\}^{\omega}$, the element of the powerdomain of E which corresponds to the fair oracle set Δ is the same as that which corresponds to the set E of all oracles (fair and unfair). As regards "computability" it seems reasonable to regard the ∞-fair merge as either "uncomputable" or, worse, as indistinguishable from the ∞-unfair merge.

There is a fundamental difficulty here, very similar to the difficulty that arises over devices with "unbounded nondeterminism", and closely related to a variety of difficulties facing a "constructive" development of mathematics. The fair set Δ has the same questionable status as many similarly defined sets of real numbers. To the strict constructivist such sets should play no part in reasoned argument or specification. Most computer scientists would sympathise with such a position. But its implication must be recognised -- that fairness, unbounded nondeterminism are "unspecifiable". They are constraints (on descriptions of languages or of devices) that may not, in their general form, be expressed -- or, if expressed, may not be used in reasoned argument. Since it does seem possible to reason usefully with such constraints, this is not the position to be taken here. But the plausibility and tidiness of the strict position must be recognised.

Obviously there is a sense in which 'merge' is computable. The following definition attempts to capture it:

IMPLEMENTABILITY: Given domains D, E, and a relation $R \subseteq D \times E$, R is implementable if there is a computable $f:D \to E$, with

$$(d,f(d)) \in R$$

for all $d \in D$.

(Note: computability is relative to indexings of bases

$$\{d_n \mid n \geq 0\}, \{e_n \mid n \geq 0\}$$

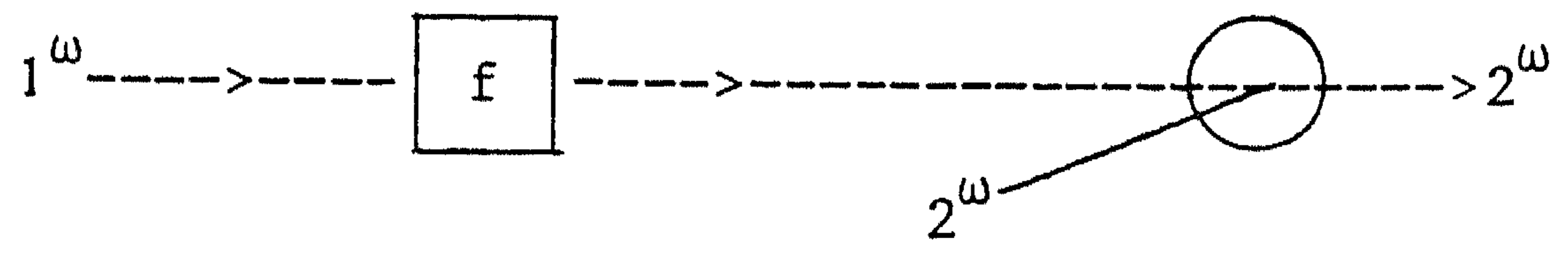
for D, E. If D, E are algebraic, $\{(m,n) \mid e_n \sqsubseteq f(d_m)\}$ should be recursively enumerable in the standard sense, for f to be computable from D to E.) Clearly merge is "implementable", as is any relation derived from a computable function using a non-empty oracle set. One appropriate computable f corresponding to merge is

$$f(u,v) = \mathrm{dmerge}(u,v,(0.1)^{\omega}).$$

For further discussion, see [5], [6].

5. ARE DETERMINISTIC NODES CONTINUOUS?

Consider the following network:



Here f is intended to characterize a function which filters its input, transmitting only the 0's which occur in it.

$$f(\perp) = \perp \qquad f(\#) = \#$$

$$f(0.u) = 0.f(u) \qquad f(x.u) = f(u), \quad x \neq 0$$

The input to f is to be 1^{ω}, the infinite sequence of 1's. Since this input contains no 0's, f has no output. Intuitively, therefore, the resulting output from the merge node is

$$\mathrm{dmerge}(f(1^{\omega}),2^{\omega},\delta)$$

which is just its second argument, 2^{ω}. Our intuition on this point is one of "angelicness". In the terms of the preceding section, we want

$$f(1^{\omega}) = \#$$

to be the "terminated" null sequence. But this must be reconciled with the continuity of f. We have

$$1^{\omega} = \bigsqcup_{n} 1^{n}$$

so

$$f(1^{\omega}) = \bigsqcup_{n} f(1^{n})$$

Now for each finite n, we can consider the possibility that the merge box waits at least n steps for some output from f. With input $1^{n}0^{\omega}$ to f this would make the output start with 0. To allow for this, we must have

A: for each n there is a δ such that $\mathrm{dmerge}(f(1^n),2^\omega,\delta)$ represents (in some form) the null sequence $\perp$.

B: for every δ, $\mathrm{dmerge}(f(1^\omega),2^\omega,\delta)$ represents (in some form) the sequence 2^ω.

It is clearly absurd that something represent both the null sequence and the infinite sequence 2^ω. Since 'dmerge' is a function, it follows from (A) and (B) that

$$f(1^\omega) \neq f(1^n), \text{ for all } n$$

while, from continuity of f

$$f(1^\omega) = \bigsqcup_n f(1^n).$$

So the limit $\bigsqcup f(1^n)$ must be non-trivial -- there are infinitely many distinct $f(1^n)$ -- all of which represent the (non-terminated) null-sequence.

COPING WITH ANGELICNESS II: Adjoin a new token τ to Σ and identify the terminator # with the infinite sequence τ^ω. Consider a sequence with occurrences of τ to represent the sequence which is obtained by deleting them. (So each sequence over the original Σ has many representations.)

This device is displeasing, since it does not seem sufficiently "abstract". But the example, together with the continuity assumption, forces some such situation on us.

We will need to show how to extend functions to act on sequences containing τ. It is clear what happens to 'dmerge' -- τ is transmitted from input to output just as the other tokens:

$$\mathrm{dmerge}(\tau.u,v,0.\delta) = \mathrm{dmerge}(u,\tau.v,1.\delta) = \tau.\mathrm{dmerge}(u,v,\delta)$$

The analogy between # and τ^ω is reasonable, given that output occurrences of τ are discounted; $\mathrm{dmerge}(\tau^\omega,v,\delta)$ consists of v with additional occurrences of τ, since δ is a fair oracle.

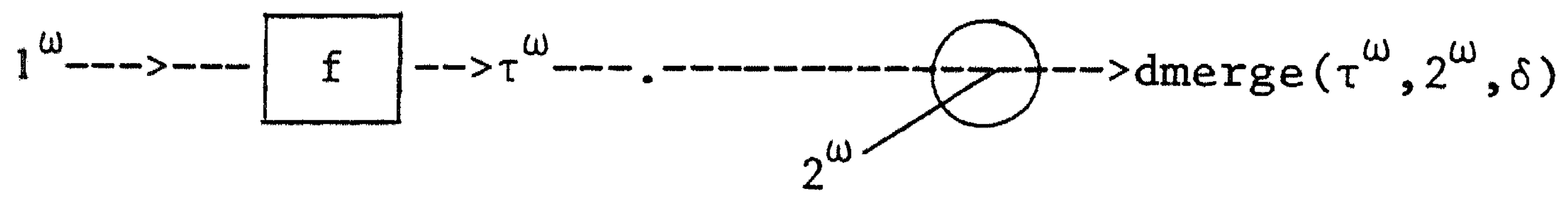
The example we have been considering is mended by setting

$$f(1^n) = \tau^n$$

which gives the final result

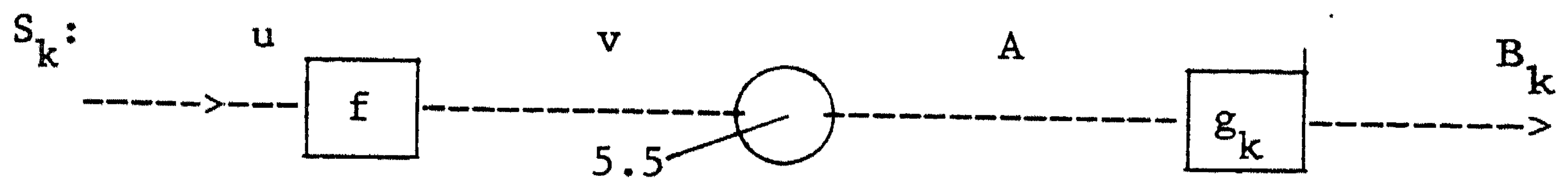
$$\mathrm{dmerge}(\tau^\omega, 2^\omega, \delta)$$

a sequence with infinitely many 2's and τ's.



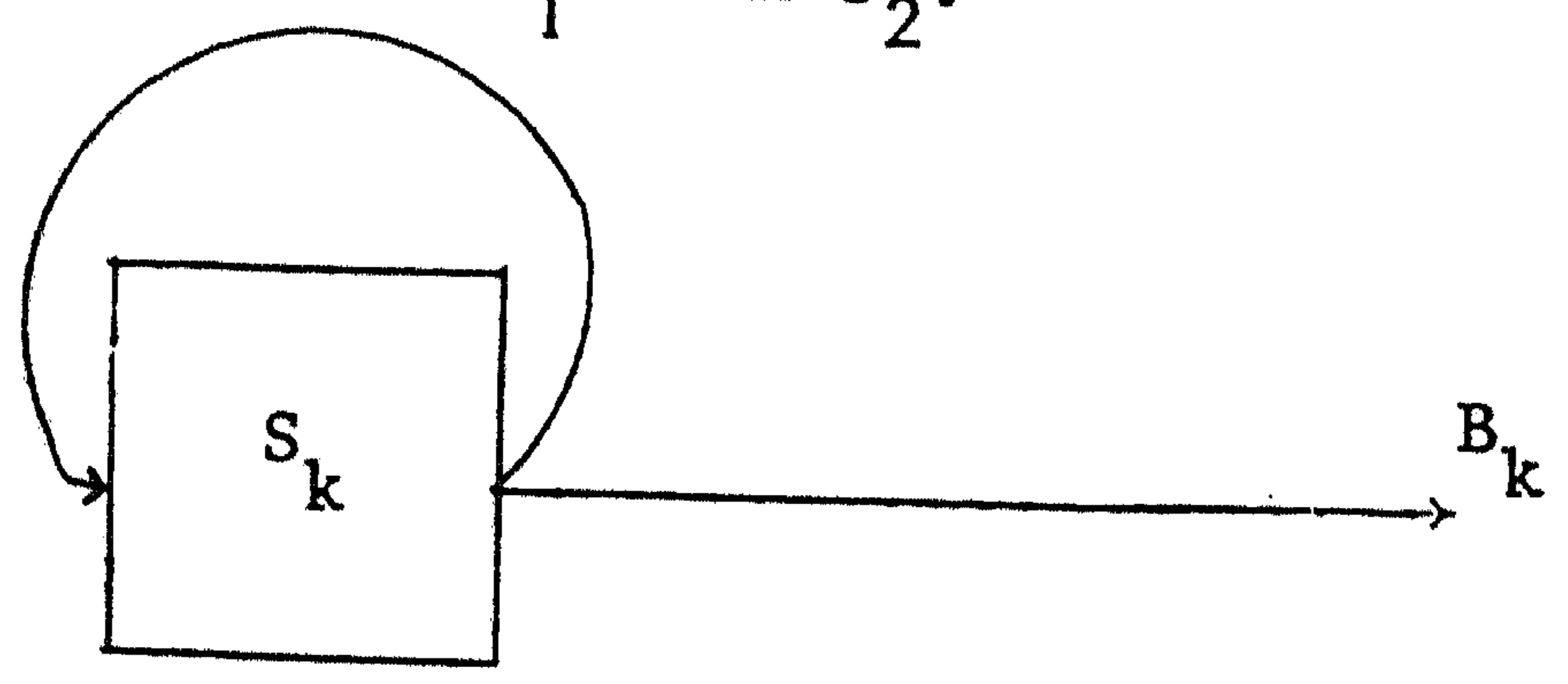
6. THE BROCK-ACKERMANN ANOMALY

Brock and Ackermann [2] have used a variety of ingenious examples to demonstrate that the straightforward history relation (without τ device) is not enough to characterise nondeterministic networks. The version in [2] hinges on two similar networks:



Tokens are digits. The node f is to select the first token from u, and transmit its successor. The output is merged with the sequence 5.5 of length 2. g_1, g_2 transmit the first two tokens they receive. The only distinction is that g_2 delays transmission of its first token until both have been received.

Neglecting τ, the two networks have the same history relations. No matter what u is, two tokens are bound to arrive at g_k eventually, since the identity of these 2 tokens is independent of k. But there is a context which distinguishes S_1 from S_2:



We have $B_1 = \{5.5,5.6\}_s B_2 = \{5.5\}$. The result 5.6 becomes impossible for S_2, because the first 5 can only be emitted after the second has been received by g_2, ahead of the 6.

What is neglected in the simple history functions is an account of input-output causality. Using τ, this can ve rectified. We will need some definitions:

GENERALIZED LENGTH: For $u \in \Sigma^\infty$, $|u| \leq \infty$ denotes the length of u (in the conventional sense). For tuples of sequences, define

$$|\langle u_1, u_2, \ldots, u_n \rangle| = \min\{|u_i|, 1 \leq i \leq n\}$$

$\succ$-FUNCTIONS: a monotone continuous f: $(\Sigma^\infty)^n \to (\Sigma^\infty)^m$ is a >-function if, whenever $|u|$ is finite,

$$|f(u)| > |u|$$

(similarly, define $\geq$-FUNCTIONS.)

The >-functions induce precisely the "contraction maps" on (tuples of) infinite sequences, and therefore seem to be the appropriate domain-theory counterpart of this notion. They are interesting for the following reasons:

1. >-functions have unique fixpoints. $|Y(f)|$ cannot be finite, or we would have $|Y(f)| = |f(Y(f))| > |Y(f)|$. But if $|Y(f)|$ is infinite, it is maximal under $\sqsubseteq$ -- therefore Y(f) is both the minimal and the maximal fixpoint of f.
2. Because of (1), applying the Kahn principle to a >-function obtains the behaviour of a network as a tuple consisting only of infinite sequences. Finite histories appear as sequences of the form $u.\tau^\omega$, for u finite.
3. Because of the "angelicness" problem, it is essential that finite histories are terminated by τ^ω, when they are inputs to merge nodes or to other nodes with angelic expectations. And this property is guaranteed, by (2).

KAHN'S PRINCIPLE REVISED: The behaviour of a nondeterministic network can be obtained, to within occurrences of τ, as a "fixpoint set"

$$\{u \mid u \in R(u)\}$$

where $R(u) = \{f(u,\delta) \mid \delta \in \Delta\}$, for a suitable (generalized) oracle set Δ and where $f(u,\delta)$ is a $>$-function on some $(\Sigma^{\infty})^{n}$ for each $\delta \in \Delta$.

The Brock-Ackermann example can be analyzed this way. We get

$$S_1(\tau^{\omega}) = \{\tau^{j}.5.\tau^{k}.5.\tau^{\omega} \mid j,\ k \geq 0\}$$

$$\begin{aligned} S_1(\tau^{i}.x.w) = {} & \{\tau^{j}.5.\tau^{k}.5.\tau^{\omega} \mid j + k \leq i\} \\ & \cup \{\tau^{j}.5.\tau^{k}.(x+1).\tau^{\omega} \mid j + k = i\} \\ & \cup \{\tau^{i+1}.(x+1).\tau^{j}.5.\tau^{\omega} \mid j \geq 0\} \end{aligned}$$

$$S_2(\tau^{\omega}) = \{\tau^{j}.5.5.\tau^{\omega} \mid j \geq 0\}$$

$$\begin{aligned} & S_2(\tau^{j}.5.5.\tau^{\omega} \mid j \leq i\} \\ & \quad \cup \{\tau^{i+1}.5.(x+1).\tau^{\omega}\} \\ & \quad \cup \{\tau^{i+1}.(x+1).5.\tau^{\omega}\} \end{aligned}$$

The reader will want to check that these are implied by the constraint that each S_k be a union of $>$-functions. The crucial term is

$$\tau^{i+1}.5.(x+1).\tau^{\omega}$$

in the definition of S_2. The initial sequence of τ's must have length $> i$, since it must have a prefix τ^{j} in $S_2(\tau^{i})$ with $j > i$ since $|\tau^{j}| > |\tau^{i}|$.

In the looping context, the behaviours B_k are just the fixpoints of S_k:

$$B_k = \{u \mid u \in S_k\}$$

These are

$$B_1 = \{\tau^{i}.5.5.\tau^{\omega},\ \tau^{i}.5.6.\tau^{\omega} \mid i \geq 0\}$$

$$B_2 = \{\tau^{i}.5.5.\tau^{\omega} \mid i \geq 0\}$$

Neglecting occurrences of τ, they are the expected behaviours. The 5.6 case is excluded from B_2, from constraint on B_2 which we just observed.

7. MAKING THE REVISED PRINCIPLE WORK

We now specify the way in which appropriate history relations, composed of >-functions, can be constructed for networks involving just deterministic nodes and merges.

A. If f is a ≥ -function into Σ^∞, then $f'(u) = \tau.f(u)$ is a >-function.

B. For each δ, the function $\mathrm{dmerge}(u,v,\delta)$ is a >-function.

C. Compositions and products of ≥-functions are also ≥-functions.

D. Let $f: (\Sigma^\infty)^n \to \Sigma^\infty$ be a continuous n-ary function on sequences. Construct a corresponding ≥-function $f'(u)$ by considering successive truncations of a components of equal length, as follows: let $u[i]$ denote the tuple obtained by truncating each component of u to length i. For $i \leq |u|$, define $f'(u[i])$ by induction on i. Put $f'(u[0]) = f(\bot,\ldots,\bot)$; given $f'(u[i])$, choose $f'(u[i+1])$ as the minimal element above $f'(u[i])$ which agrees with $f(u[i+1])$ to within occurrences of τ, and with length $\geq i + 1$ (note this can always be done). Finally, put

$$f'(u) = f'(u[|u|]), \text{ if } |u| \text{ is finite}$$

or

$$f'(u) = \bigsqcup_i f'(u[i]), \text{ if } |u| \text{ is infinite.}$$

Notice that $f'(u)$ is bound to agree with $f(u)$ modulo τ only when u is a tuple of sequences of equal length -- i.e. only when $u = u[|u|]$. (An example which displays the awkwardnesses is a "parallel or"

$$f(1.u,v) = f(u,1.v) = 1 \quad f(0.u,0.v) = 0$$

There is no $f'(u,v)$ which agrees with $f(u,v)$ modulo τ for arbitrary u, v. The proof of this fact is left to the interested reader.) It is clear that application of (A) - (D) allow the "massaging" of specifications of deterministic nodes, and of nondeterministic nodes analyzable in terms of 'merge', into >-functions.

A proof that the result of these manipulations satisfies the revised Kahn principle is given in an Appendix to these notes. It turns out to be quite elaborate -- for a largely technical reason. It should be clear that every behaviour in the fixpoint set is a possible behaviour for the network -- since each of these is obtained by an infinite iteration of the

>-function corresponding to a particular oracle choice. It is the converse direction that provides the difficulty. Given a behaviour u for the network, one wants to find a δ such that

$$u = f(u,\delta)$$

But all that appears to be given is an oracle δ' such that

$$u \sim f(u,\delta')$$

i.e. such that u, $f(u,\delta')$ are identical modulo τ.
The difficulty derives in passing from the second of these assertions to the first -- which turns out to be possible only when the second assertion is rephrased and analyzed carefully.

8. SUMMARY AND CONCLUDING REMARKS

The modelling method can be viewed as proceeding as follows:

(1) Nondeterminism is firstly eliminated by introducing extra "oracle" parameters; infinite fairness is ensured by constraining oracles to be "fair"; angelic fairness is ensured by introducing the "hiaton" τ as a possible communication. The reception of hiatons affects those operations which use oracles, but no others (except possibly in causing the propagation of further hiatons).

(2) The operation of each component of a network can be specified by a function

$$f(u,\delta)$$

on (tuples of) hiatonized sequences u and on fair oracles δ, which is continuous in u. The value $f(u,\delta)$ is to be the sequence of outputs from the component which are available as a result of u, up to some moment (if any) after reception of all of u, and before reception of further inputs. The stepwise effect $f(u,\delta)$ need not contain all of the ultimate effect $f(u.\tau^{\omega},\delta)$, for finite sequences u.

(3) Such a function $f(u,\delta)$ is to be manipulated, if need be, by introducing hiatons into output sequences, so as to obtain a function

$$f^{\wedge}(u,\delta)$$

which is continuous in u, such that

$$f^{\wedge}(u,\delta) \sim f(u,\delta) \qquad \text{for } |u| = \infty,$$

and

$$|f^{\wedge}(u,\delta)| > |u| \qquad \text{for } |u| < \infty.$$

(4) Once the function $f^{\wedge}(u,\delta)$ has been obtained, its values on finite sequences may be disregarded, and oracles may be abstracted out of consideration. It is sufficient to regard a component or network as specified by its ULTIMATE BEHAVIOUR RELATION

$$R(u) = \{f^{\wedge}(u,\delta) \mid \delta \in \Delta\}, \qquad |u| = \infty$$

which maps (tuples of) *infinite* hiatonized sequences to sets of (tuples of) infinite hiatonized sequences.

(5) Ultimate behaviour relations can be combined in elementary set-theoretic ways to model combinations of components and of networks:

parallel composition of networks corresponds to the cartesian product operation on sets

$$(R \times S)(u) = (R(u), S(u)).$$

serial composition of networks corresponds to (serial) composition of relations

$$(R;\, S)(u) = \{w \mid w \in S(v) \text{ and } v \in R(u), \text{ for some } v\}.$$

feedback of outputs to (some subset of) inputs corresponds to an operation on relations which generalizes the "fixpoint" notion

$$(Y v.R(u,v))(u) = \{v \mid v \in R(u,v)\}.$$

(Perhaps this should be called the set of "containpoints").

(6) These and other manipulations of behaviours can be justified by reference to the Main Theorem proved in the Appendix, which characterizes possible behaviours of any closed network (i.e. of any network without

input parameters). In the Main Theorem such a network was regarded as a single n-ary feedback

$$(Yu.R(u)) = \{u \mid u \in R(u)\}$$

where

$$R(u) = (R_1 \times R_2 \times \ldots\ldots \times R_n)(u)$$

corresponds to the parallel composition of the n network components, each regarded as a component with n inputs. Replacement of a set of component relations by its composition, where appropriate, does not essentially alter the solution set

$$\{u \mid u \in R(u)\}.$$

(7) BEHAVIOURAL EQUIVALENCE of networks with ultimate behaviours R, S can be defined by

$$R \sim S \text{ iff } \varepsilon(R(u)) = \varepsilon(S(u)) \text{ for all } |u| = \infty.$$

CONTEXTUAL EQUIVALENCE is then

$$R \underline{=} S \text{ iff } T[R] = T[S]$$

for any "context" T[X] -- the behaviour of a complete network as a function of a varying component X.

The Brock-Ackermann example shows that

$$R \sim S \text{ does } \textit{not} \text{ imply } R \underline{=} S.$$

Remarkably, we do have

$$R = S \text{ implies } R \underline{=} S \text{ (which implies } R \sim S)$$

from the Main Theorem of the Appendix.

(8) While we have achieved a model which distinguishes networks which are contextually inequivalent, we have not arrived at a completely satisfactory "fully abstract" semantics for networks, since it is clear that

$$R \mathrel{\underline{=}} S \text{ does not imply } R = S$$

(e.g. take $S(u) = \{\tau.v \mid v \in R(u)\}$, for any R). We could, of course, take "denotations" for networks to be equivalence classes of ultimate behaviours under '$\underline{=}$'; but the structure of such equivalence classes is obscure. Making the notion more logically tractable appears to be both hard and worthy of further effort.

(9) The methods of these notes appear to generalise to other domains over which there are interesting "non-deterministic" recursion equations of the same elementary form

$$\begin{aligned} x_1 &= R_1(x_1,\ldots,x_n) \\ x_2 &= R_2(x_1,\ldots,x_n) \\ &\vdots \\ x_n &= R_n(x_1,\ldots,x_n) \end{aligned}$$

in which the R_i involve nondeterministic operators -- most immediately to nondeterministic recursive definitions of list structures (i.e. of trees). But there is a more general problem to be treated before such ideas can be extended to the denotational semantics of programming languages involving fair nondeterministic constructs. This would require a treatment, at least, of recursive "function" definitions

$$\begin{aligned} f_1(x_1,\ldots,x_{n_1}) &= R_1(f_1,\ldots,f_m,x_1,\ldots,x_{n_1}) \\ &\vdots \\ f_m(x_1,\ldots,x_{n_m}) &= R_m(f_1,\ldots,f_m,x_1,\ldots,x_{n_m}) \end{aligned}$$

in which the R_i involve fair nondeterminism. We look forward to such a treatment developing from the ideas described in these notes -- though there are further conceptual difficulties to be dealt with before this could be done adequately.

APPENDIX: JUSTIFICATION OF THE "REVISED KAHN PRINCIPLE"

This appendix outlines a formal proof of the principle enunciated in the main notes.

Assume an alphabet Σ of tokens, contianing the "hiaton" τ. Let Δ be the fair oracle set $(0^*11^*0)^\omega$.

1. DEFINITIONS

ESSENCE: (x) is the function which deletes occurrences of τ from x, with recursion equations

$$\varepsilon(\bot) = \bot;\ \varepsilon(\tau.x) = \varepsilon(x);\ \varepsilon(a.x) = a.\varepsilon(x);\ a \neq \tau;$$

Consider $\varepsilon(x)$ as extended to tuples of sequences by

$$\varepsilon(x_1,\ldots x_n) = (\varepsilon(x_1),\ldots.,\varepsilon(x_n)).$$

EQUIVALENCE: $x \sim y$ iff $\varepsilon(x) = \varepsilon(y)$.

ASYNCHRONEITY: a function f is asynchronous if

$$f(x) = f(\varepsilon(x))$$

(i.e. if f is insensitive to occurrences of τ).

ASYNCHRONOUS EXTENSION: any function f on $((\Sigma-\{\tau\})^\infty)^n$ extends naturally to an asynchronous function on $(\Sigma^\infty)^n$, by setting

$$f(x) = f(\varepsilon(x))$$

(using the same symbol for f -- without confusion, we hope).

MERGE-COMBINATION: a function

$$f\colon (\Sigma^\infty)^n \times \Delta^m \to (\Sigma^\infty)^m$$

is a MERGE-COMBINATION, if each component

$$f(u_1,\ldots,u_n,\delta_1,\ldots,\delta_m)_i$$

is either an instance of dmerge

$$= \text{dmerge}(u_{j_i},u_{k_i},\delta_i),\ \text{for some } j_i,\ k_i$$

or is asynchronous and independent of δ

$$= f_i(\varepsilon(u_1),\ldots,\varepsilon(u_n)),\ \text{for some continuous } f_i.$$

Note that a merge-combination, as defined above, is intended to model a parallel composition of m nodes, each producing one output, sharing up to n inputlines. Each node is either a fair merge or is deterministic (i.e. independent of relative timing of inputs). We can write such a combination as

$$f(u,\delta), \quad u \in (\Sigma^{\infty})^n, \quad \delta \in \Delta^m$$

thinking of δ as a "generalized oracle". The component δ_i is only relevant it the i-th node is a merge.

A network of such nodes is describable by a merge-combination with $m = n$. The following then asserts the validity of the extended Kahn principle for such networks.

2. MAIN THEOREM

2.0: Let $f\colon (\Sigma^{\infty})^n \times \Delta^n \to (\Sigma^{\infty})^n$ be a merge-combination (with $m = n$). There is a function $f\hat{}$, with the same functionality as f, such that

$$f\hat{}(u,\delta)$$

is a >-function, for each $\delta \in \Delta^n$, and such that the following are equivalent, for each $w \in (\Sigma^{\infty})^n$:

(i) There is a $\delta \in \Delta^n$ and $v_i \in (\Sigma^{\infty})^n$, $i = 0, 1, \ldots$ such that $(\bot,\ldots,\bot) = v_0 \sqsubseteq v_1 \sqsubseteq \ldots\ldots$, $w \sim v = \sqcup_i v_i$, $|v| = \infty$, and $\varepsilon(v_{i+1}) \sqsubseteq \varepsilon(f(v_i,\delta)) \sqsubseteq \varepsilon(v)$ for each $i \geq 0$.

(ii) There is a $\delta' \in \Delta^n$ and a $v \sim w$, with $v = f\hat{}(v,\delta')$

It should be clear that (ii) produces what is needed. With

$$R(u) = \{f\hat{}(u,\delta) \mid \delta \in \Delta^n\},$$

the set w we are interested in is

$$\{\varepsilon(u) \mid u \in R(u)\}.$$

The Main Theorem then asserts that this is the set of behaviours of the corresponding network. It takes a certain amount of reflection to see that (i) is the property of w which expresses this. The key point is the nature of the correspondence between the sequence v_0, v_1, ... and an intended behaviour. Writing

$$v_{i+1} = v_i . w_i$$

the corresponding behaviour is intended to be one in which the symbols emitted at the i-th step by the n nodes of the network are the n components of w_i.
We can now return to the development which leads to the Main Theorem.

3. ELEMENTARY PROPERTIES

3.0: ε is continuous and monotone (if u is a prefix of v, $\varepsilon(u)$ is a prefix of $\varepsilon(v)$).
3.1: the asynchronous extension of a continuous (monotone) function on $(\Sigma-\{\tau\})^{\infty}$ is continuous (monotone) on Σ^{∞}.
3.2: if $x \sqsubseteq y \sqsubseteq z$ and $x \sim x' \sqsubseteq z' \sim z$, there is a y' with $x' \sqsubseteq y' \sqsubseteq z'$, and $y' \sim y$. (Construct a suitable continuous function on z such that

$$f(x) = x', \ f(z) \sqsubseteq z', \text{ and } w \sim f(w) \text{ for all } w.$$

Then take $y' = f(y)$)

$$\begin{tikzcd}
z \arrow[d, dash] & \sim & z' \\
y \arrow[d, dash] & \sim & y' \ . \\
x & \sim & x'
\end{tikzcd}$$

4. MANIPULATIONS ON ASYNCHRONOUS FUNCTIONS

>-VARIANT: $f^{\wedge}$ is a >-variant of f if

(i) $\varepsilon(f^{\wedge}(x) \sqsubseteq \varepsilon(f(x))$ for all x
(ii) if $|x| = \infty$ then $\varepsilon(f^{\wedge}(x)) = \varepsilon(f(x))$
(iii) $f^{\wedge}$ is a >-function.

LEMMA 4.0. *Every continuous asynchronous* f *has a >-variant* $f^{\wedge}$.

PROOF. As in the main text. Define $f^{\wedge}(x[i])$ by induction on i, where $x[i] \sqsubseteq x$ is the tuple of components of x truncated to lenght i; if $|x| = \infty$, put $f^{\wedge}(x) = \bigsqcup_i f^{\wedge}(x[i])$; otherwise $f^{\wedge}(x) = f^{\wedge}(x[|x|])$.

LEMMA 4.1. *If* $f^{\wedge}$ *is a >-variant of a continuous* f, $v' \sqsubseteq v$ *and* $|v| = \infty$, *then*
(i) there is a $w' \sqsubseteq f^{\wedge}(v)$ *with* $w' \sim f(v')$
(ii) if $|v'| < \infty$ *then* $|w'| > |v'|$.

PROOF. Use Elementary Property 3.2 above with $x' = x = \bot$ to get $\hat{f}(v') \sim y \sqsubseteq f(v')$. Since $v \sqsubseteq v'$, we have $f(v') \sqsubseteq f(v)$ and $\hat{f}(v') \sqsubseteq \hat{f}(v)$ by monotonicity. Choose w' by Elementary Property 3.2, then (i) of the Lemma holds, and if $|v'| < \infty$, then $|w'| \geq |\hat{f}(v')| > |v'|$

$$\begin{tikzcd}
& f(v) \arrow[r, phantom, "\sim"] \arrow[d, dash] & \hat{f}(v) \arrow[d, dash] \\
\varepsilon(f(v')) \arrow[r, phantom, "\sim"] \arrow[d, dash] & f(v') \arrow[r, phantom, "\sim"] \arrow[d, dash] & w' \arrow[d, dash] \\
\varepsilon(\hat{f}(v')) \arrow[r, phantom, "\sim"] \arrow[d, dash] & y \arrow[r, phantom, "\sim"] \arrow[d, dash] & \hat{f}(v') \\
\bot \arrow[r, phantom, "\sim"] & \bot &
\end{tikzcd} .$$

5. ORACLE TRANSFORMS

It is necessary to analyze the trade-off between the structure of oracles to nondeterministic nodes, and the way in which sequences are "diluted" by occurrences of τ. If the dilution changes, it is possible to compensate by choosing a different oracle. It turns out to be important that this compensation can be done "continuously". In fact the only transform in question here is a transform for dmerge, but the property is important enough to be worth stating for arbitrary oracle-dependent functions.

Assume oracles are chosen from Δ. We will need to regard Δ as embedded in the domain $\Omega = \{0,1\}^\infty$, so that the oracle-dependent function

$$f\colon (\Sigma^\infty)^n \times \Delta \to \Sigma^\infty$$

is induced by a function

$$f\colon (\Sigma^\infty)^n \times \Omega \to \Sigma^\infty$$

which is continuous on Ω as well as on $(\Sigma^\infty)^n$. An *oracle transform* for f is then a continuous function

$$\phi\colon (\Sigma^\infty)^n \times (\Sigma^\infty)^n \times \Omega \to \Omega$$

which produces an oracle to compensate for inessential variations in the argument u to

$$f(u,\delta)$$

DEFINITIONS

CONSISTENCY: u is consistent with v if, for some v',

$$u \sim v' \sqsubseteq v$$

τ-MAXIMALITY: u is a τ-maximal prefix of v -- written

$$u \sqsubseteq_\tau v$$

if (i) $u \sqsubseteq v$

(ii) $u \sim u' \sqsubseteq v$ implies $u' \sqsubseteq u$ for all u'

BASIC PROPERTIES:

5.1: every $w \sqsubseteq v$ can be extended to a $w' \sqsubseteq_\tau v$, with $w \sqsubseteq w'$, $w \sim w'$.

5.2: $a.u \sqsubseteq_\tau a.v$ if and only if $u \sqsubseteq_\tau v$.

5.3: if $u_1 \sqsubseteq_\tau u_2 \sqsubseteq_\tau \ldots$ then each $u_i \sqsubseteq_\tau \bigsqcup_n u_n$.

ORACLE TRANSFORMS: ϕ is an oracle transform for f if ϕ is continuous, and, whenever $\delta \in \Delta$, $|u| = |v| = \infty$

(5.4.1) if $u' \sim v'$, $u' \sqsubseteq_\tau u$, $v' \sqsubseteq_\tau v$, then $f(v',\phi(u,v,\delta)) \sim f(u',\delta)$

(5.4.2) if v' is consistent with u, then $|f(v',\phi(u,v',\delta))| \geq |v'|$

(5.4.3) if $u \sim v$ then $\phi(u,v,\delta) \in \Delta$.

NOTE. The τ-maximality condition in 5.4.1 seems to be essential. This can be seen before we look in detail at a transform for f = dmerge by reflecting just on the case that u = v; it is reasonable that

$$\phi(u,u,\delta) = \delta$$

but 5.4.1 then fails with $u' = u = v = (\tau^\omega, x)$, $v' = (\perp, x)$, and x any infinite sequence. If δ is a fair oracle, the left-hand of 5.4.1 denotes some finite initial segment of x, whereas the right-hand of 5.4.1 denotes a sequence equivalent to all of x.

6. A TRANSFORM FOR DMERGE

The function dmerge is easily extended to a continuous function on all of Ω by adding equations for the empty oracle $\delta = \perp$. It is then the (unique) function satisfying:

(6.0.0) $\text{dmerge}\ (u_0,u_1,\perp) = \perp$

($6.0.1_0$) $\text{dmerge}\ (\perp,u_1,0.\delta) = \perp$

($6.0.1_1$) $\text{dmerge}\ (u_0,\perp,1.\delta) = \perp$

$(6.0.2_0)$ $\mathrm{dmerge}\,(a.u_0,u_1,0.\delta) = a.\mathrm{dmerge}\,(u_0,u_1,\delta)$
$(6.0.2_1)$ $\mathrm{dmerge}\,(u_0,a.u_1,1.\delta) = a.\mathrm{dmerge}\,(u_0,u_1,\delta)$.

The corresponding transform $\phi(u_0,u_1,v_0,v_1,\delta)$ can then be defined by the following recursion equations:

$(6.0.3)$ $\phi(u_0,u_1,v_0,v_1,\perp) = \perp$
$(6.0.4_0)$ $\phi(\perp,u_1,v_0,v_1,0.\delta) = \phi(u_0,u_1,\perp,v_1,0.\delta) = \perp$
$(6.0.4_1)$ $\phi(u_0,\perp,v_0,v_1,1.\delta) = \phi(u_0,u_1,v_0,\perp,1.\delta) = \perp$

for each $a \in \Sigma$:

$(6.0.5_0)$ $\phi(a.u_0,u_1,a.v_0,v_1,0.\delta) = 0.\phi(u_0,u_1,v_0,v_1,\delta)$
$(6.0.5_1)$ $\phi(u_0,a.u_1,v_0,a.v_1,1.\delta) = 1.\phi(u_0,u_1,v_0,v_1,\delta)$

for each $a \in \Sigma$, $a \neq \tau$:

$(6.0.6_0)$ $\phi(\tau.u_0,u_1,a.v_0,v_1,0.\delta) = \phi(u_0,u_1,a.v_0,v_1,\delta)$
$(6.0.6_1)$ $\phi(u_0,\tau.u_1,v_0,a.v_1,1.\delta) = \phi(u_0,u_1,v_0,a.v_1,\delta)$
$(6.0.7_0)$ $\phi(a.u_0,u_1,\tau.v_0,v_1,0.\delta) = 0.\phi(a.u_0,u_1,v_0,v_1,0.\delta)$
$(6.0.7_1)$ $\phi(u_0,a.u_1,v_0,\tau.v_1,1.\delta) = 1.\phi(u_0,a.u_1,v_0,v_1,1.\delta)$

for each $a, b \in \Sigma$, $a \neq b$, $a, b \neq \tau$:

$(6.0.8_0)$ $\phi(a.u_0,u_1,b.v_0,v_1,0.\delta) = \perp$
$(6.0.8_1)$ $\phi(u_0,a.u_1,v_0,b.v_1,1.\delta) = \perp$

<u>NOTE</u>. These equations do not uniquely determine ϕ, because of the two equations $6.0.6_i$, we must specify, as usual, that ϕ is the minimal solution to $(6.0.3)$-$(6.0.8_1)$, in the lattice theoretic sense. Thus

$$\phi(\tau^\omega,\tau^\omega,0.x,0.x,\delta) = \perp$$

for all infinite δ and all x, since $\perp$ is the minimal solution to the relevant equations, all of which are instances of $6.0.6_0$ or $6.0.6_1$.

<u>LEMMA 6.1</u>. *ϕ, as specified above, is an oracle transform for dmerge.*

<u>PROOF</u>. (ad 5.4.1) In fact the required equivalence holds for the finite oracles $\delta \in \Omega$. First check that this is so, proceeding by induction on $|\delta|$, as follows:
Abbreviate the two sides of the equivalence by

$$\begin{aligned} x &= \mathrm{dmerge}(u'_0,u'_1,\delta) \\ \delta' &= \phi(u_0,u_1,v_0,v_1,\delta) \\ y &= \mathrm{dmerge}(v'_0,v'_1,\delta'). \end{aligned}$$

We want to show that $x \sim y$, given that $u'_i \sim v'_i$, $u'_i \sqsubseteq u_i$, $v'_i \sqsubseteq_\tau v_i$, $|u_i| = |v_i| = \infty$, $i = 0, 1$. The base of the induction is just the case $\delta = \bot$, in which $\delta' = \bot$, from 6.0.3, and $x = y = \bot$, from 6.0.0. Suppose now that the equivalence holds whenever $|\delta| < n$, and that $|\delta| = n$. Assume that 0 is the initial symbol of δ -- i.e. that $0 \sqsubseteq \delta$, there are four cases to consider, depending on the initial symbols of u_0, v_0 (the cases in which $1 \sqsubseteq \delta$ are analogous):

(a) $\tau \sqsubseteq u_0$ and $\tau \sqsubseteq v_0$: then $\tau \sqsubseteq u'_0$ and $\tau \sqsubseteq v'_0$, since u'_0 and v'_0 are τ-maximal prefixes; obtain x', y' from $6.0.2_0$, $6.0.5_0$, such that $x = \tau.x'$, $y = \tau.y'$, and $x' \sim y'$ from the induction hypothesis.

(b) $\tau \sqsubseteq u_0$ and $a \sqsubseteq v_0$, some $a \neq \tau$: then $x = \tau.x'$ as in (a), and $x' \sim y$ from the induction hypothesis and $6.0.6_0$.

(c) $a \sqsubseteq u_0$ for some $a \neq \tau$, and $u'_0 = \bot$, then $v'_0 = \tau^m$, for some $m \leq \omega$; if m is finite, then $\tau^m.b \sqsubseteq v_0$, for some $b \neq \tau$, since $v'_0 \sqsubseteq_\tau v_0$; apply $6.0.7_0$ m times to obtain $0^m \sqsubseteq \delta'$, and then either $6.0.5_0$ or $6.0.8_0$ to obtain $0^m = \delta'$, or $0^{m+1} \sqsubseteq \delta'$; in either case $y = \tau^m \sim x$.

(d) $a \sqsubseteq u_0$ for some $a \neq \tau$, and $a \sqsubseteq u'_0$: then $\tau^m.a \sqsubseteq v'_0$, for some finite m, since $u'_0 \sim v'_0$. Obtain $y = \tau^m.a.y'$, by m applications of $6.0.7_0$, $6.0.2_0$ and one application of $6.0.5_0$, $6.0.2_0$. Then $x = a.x'$, with $x' \sim y'$ by the induction hypothesis.

All possibilities are covered by (a) - (d) and by the analogues with $1 \sqsubseteq \delta$. 5.4.1 therefore holds for all finite δ. The extension to infinite oracles is straightforward, from the continuity of dmerge and ϕ on oracles δ, and of the essence map $\varepsilon(x)$ on sequences x. Thus

$$\varepsilon(\mathrm{dmerge}(u'_0,u'_1,\delta)) = \bigsqcup\{\varepsilon(\mathrm{dmerge}(u'_0,u'_1,\delta''))\,|\,\delta'' \leq \delta,\ |\delta''| < \infty\}$$

etc. .

(ad 5.4.2) A similar argument shows, by induction on $|v'_0| + |v'_1|$, that

$$|\phi(u,v',\delta)| \geq |v'|$$

provided v' is consistent with u, and $|u| = |\delta| = \infty$; and, by induction on $|\delta'|$, that

$$|\mathrm{dmerge}(v',\delta')| \geq \min\{|v'|, |\delta'|\}$$

5.4.2 results from combining these two inequalities.
(ad 5.4.3) By induction on $|\delta'|$, every finite prefix $\delta' \sqsubseteq \phi(u,v,\delta)$ determines a decomposition $\phi(u,v,\delta) = \delta'.\phi(u'',v'',\delta'')$ for some infinite u'' and $\delta'' \in \Delta$. It is just sufficient therefore to exclude the possibilities $\phi(u,v,\delta) = \bot, 0^{\omega}$, or 1^{ω}.

(a) if $\phi(u,v,\delta) = \bot$ then, for $i = 0$ or 1, no nontrivial $v'_1 \sqsubseteq v_i$ is consistent with u_i; but then $u_i \not\sim v_i$.
(b) $\phi(u,v,\delta) = i^{\omega}$ only when $v_i = x.\tau^{\omega}$, $y.a \sqsubseteq u_i$, with $x \sim y$, $a \neq \tau$, $i^{|y|+1} \sqsubseteq \delta$-- given that $\delta \in \Delta$; but then $u_i \not\sim v_i$ again.

This concludes the proof of 6.1.

7. THE MAIN RESULTS

A network is specified by continuous functions

$$f: (\Sigma^{\infty})^n \times \Delta^n \to (\Sigma^{\infty})^n$$

with

$$f(u_1,\ldots,u_n,\delta_1,\ldots,\delta_n) = (f_1(u_1,\ldots,u_n,\delta_1),\ldots, f_n(u_1,\ldots,u_n,\delta_n)).$$

Each f_i is either asynchronous, and has a >-variant $f_i^{\wedge}$, from 4.0, or is an instance of dmerge:

$$f_i(u_1,\ldots,u_n,\delta_i) = \mathrm{dmerge}(u_{k_i},u_{l_i},\delta_i).$$

If f_i is an instance of dmerge, it can be made into a >-function by setting

$$f_i^{\wedge}(u_1,\ldots,u_n,\delta_i) = \tau.f_i(u_1,\ldots,u_n,\delta_i).$$

Since dmerge is already a ≥-function. Combine these >-functions into a single function

$$f^{\wedge}(u,\delta) = (f_1^{\wedge}(u,\delta_1),\ldots,f_n^{\wedge}(u,\delta_n))$$

$$u = (u_1,\ldots,u_n)$$

f can be regarded as having a combined oracle transform

$$\phi^{\wedge}(u,v,\delta) = (\phi_1^{\wedge}(u,v,\delta_1),\ldots,\phi_n^{\wedge}(u,v,\delta_n))$$

with

$$\phi_i^{\wedge}(u,v,\delta_i) = \delta_i, \text{ say, if } f_i \text{ is asynchronous}$$

and

$$\phi_i^{\wedge}(u,v,\delta_i) = \phi(u_{k_i},u_{l_i},v_{k_i},v_{l_i},\delta_i),$$

when f_i is an instance of dmerge -- where ϕ is as specified in the preceding section.

LEMMA 7.1. *If* $|u| = |v| = \infty$, $v' \sqsubseteq_\tau v$, $u' \sqsubseteq_\tau u$, $u' \sim v'$, $\delta \in \Delta^n$, *then*

(i) *there exists* $w' \sim f(u',\delta)$, *such that*

$$f^{\wedge}(v',\phi^{\wedge}(u,v',\delta)) \sqsubseteq w' \sqsubseteq f^{\wedge}(v,\phi^{\wedge}(u,v,\delta))$$

(ii) *if* $|v'| < \infty$ *then* $|w'| \geq |f^{\wedge}(v',\phi^{\wedge}(u,v',\delta))| > |v'|$.

(iii) *if* $u \sim v$ *then* $\phi^{\wedge}(u,v,\delta) \in \Delta^n$.

PROOF. The Lemma amalgamates 4.1, 6.1, and follows by applying them appropriately to projections $f_i(u,\delta)$:

(a) for asynchronous f_i, choose $w' \sim f_i(v',\ldots)$, as in 4.1.

(b) when f_i is an instance of dmerge, take

$$w' = f_i^{\wedge}(v',\phi_i^{\wedge}(u,v,\delta_i))$$

as in 6.1. (iii) is immediate. (i), (ii) follow from monotonicity of f_i and ϕ_i.

We are now in position to conclude the proof of the main result.

PROOF OF MAIN THEOREM 2.0. It will be convenient to show (i), (ii) in 2.0 are equivalent to (i'): There is a $\delta \in \Delta^n$ and $u_i \in (\Sigma^\infty)^n$, $i = 0, 1,\ldots$ such that $(\bot,\ldots,\bot) \sim u_0 \sqsubseteq_\tau u_1 \sqsubseteq_\tau \cdots$ with $u_{i+1} \sim f(u_i,\delta)$ for each $i \geq 0$, and $w \sim \bigsqcup_i u_i$. The proof proceeds in three parts:

(ii) ⇒ (i): given $v = f^{\wedge}(v,\delta')$, define v_i inductively by

$$v_0 = (\perp,\ldots,\perp)$$
$$v_{i+1} = f\hat{\ }(v_i,\delta').$$

If $|v_i| < \infty$ then $|v_{i+1}| > |v_i|$, from construction of f^; so $|\sqcup v_i| = \infty$ and $\sqcup v_i = v$; moreover

$$\begin{aligned}\varepsilon(v_{i+1}) &= \varepsilon(f\hat{\ }(v_i,\delta')) \sqsubseteq \varepsilon(f(v_i,\delta')) \\ &\sqsubseteq \varepsilon(f(v,\delta')) = \varepsilon(f\hat{\ }(v,\delta')) = \varepsilon(v).\end{aligned}$$

<u>(i) ⇒ (i')</u>: given a nested sequence v_i, with $v = \sqcup v_i$, $v_0 = (\perp,\ldots,\perp)$, $\varepsilon(v_{i+1}) \sqsubseteq \varepsilon(f(v_i,\delta)) \le \varepsilon(v)$, note first that $v \sim f(v,\delta)$, since

$$\varepsilon(v) = \varepsilon(\sqcup_i v_i) \le \varepsilon(\sqcup_i f(v_i,\delta)) = \varepsilon(f(v,\delta))$$

and

$$\varepsilon(f(v,\delta)) = \varepsilon(\sqcup_i f(v_i,\delta)) \sqsubseteq \varepsilon(v)$$

(from given assumptions, plus monotonicity and continuity). Now choose u_i as in (i'), to satisfy

(a) $\quad v_i \sqsubseteq u_i \sqsubseteq_\tau v$

(b) $\quad$ each u_i is consistent with $f(u_i,\delta)$,

by induction on i. For $i = 0$, choose $\perp \sim u_0 \sqsubseteq_\tau v$. Given u_i satisfying (a), (b), choose $u_{i+1} \sqsubseteq_\tau v$, with $u_{i+1} \sim f(u_i,\delta)$. Since $u_i \sqsubseteq v$, $f(u_i,\delta) \sqsubseteq f(v,\delta) \sim v$, so u_{i+1} exists; and $u_i \sqsubseteq_\tau u_{i+1}$, since u_i is consistent with $f(u_i,\delta) \sim u_{i+1}$ and $u_i \sqsubseteq_\tau v$; moreover, since $v_i \sqsubseteq u_i$,

$$\varepsilon(v_{i+1}) \sqsubseteq \varepsilon(f(v_i,\delta)) \sqsubseteq \varepsilon(f(u_i,\delta)) = \varepsilon(u_{i+1}),$$

so $v_{i+1} \sqsubseteq u_{i+1}$, since u_{i+1} is τ-maximal in v. Finally, since $u_{i+1} \sim f(u_i,\delta)$, u_{i+1} is consistent with $f(u_{i+1},\delta)$, from monotonicity of f and nestedness of u_i. From (a), $v = \sqcup_i v_i \sqsubseteq \sqcup_i u_i \sqsubseteq v$; so $\sqcup_i u_i = v$, and $|v| = \infty$; so (i') is satisfied.

<u>(i') ⇒ (ii)</u>: given $u = \sqcup_i u_i$, and δ as in (i'), consider the following function:

$$g(v) = f\hat{\ }(v,\phi\hat{\ }(u,v,\delta))$$

g is a composition of continuous functions, and therefore continuous; it therefore has a minimal fixpoint

$$v = f^{\wedge}(v,\phi^{\wedge}(u,v,\delta)).$$

It will be sufficient to show that $v \sim u$ and that $\phi^{\wedge}(u,v,\delta) \in \Delta^n$.

(a) First check that $|v| = \infty$. Take $v_0 = (\bot,\ldots,\bot)$, $v_{i+1} = f^{\wedge}(v_i,\phi^{\wedge}(u,v_i,\delta)) \sqsubseteq v$. Then v_0 is consistent with u, and if v_i is consistent with u, so is v_{i+1}, from 7.1 (i) (replacing v there by some v'' with $v_i \sqsubseteq_\tau v''$, and noting that $w' \sim f(u',\delta) \sqsubseteq f(u,\delta) \sim u$). And $|v_{i+1}| > |v_i|$, from 7.1 (ii); so $|v| = \infty$.

(b) Now use the Main Lemma 7.1 to compare v with u. For each i, choose w_i with

$$u_i \sim w \sqsubseteq_\tau v.$$

$w_0 \sim \bot$ clearly exists; and given $w_i \sqsubseteq_\tau v$, we have $w_i \sim u_i \sqsubseteq_\tau u$; so the hypotheses of 4.1 (i) hold, producing $w_{i+1} \sqsubseteq_\tau f^{\wedge}(v,\phi^{\wedge}(u,v,\delta)) = v$, with $w_{i+1} \sim f(u_i,\delta) \sim u_{i+1}$; and $|w_{i+1}| > |w_i|$. So $\sqcup_i w_i$ is infinite; so $v = \sqcup_i w_i$. Finally, since each $u_i \sim w_i$, and ε is continuous,

$$\varepsilon(v) = \varepsilon(\sqcup_i w_i) = \varepsilon(\sqcup_i u_i) = \varepsilon(u)$$

i.e. $v \sim u$, as required.

Since $v \sim u$, and $|v| = |u| = \infty$, $\phi^{\wedge}(u,v,\delta) \in \Delta^n$, as required. The proof is now complete.

REFERENCES

[1] BOUSSINOT, F., *Proposition de semantique denotationnelle pour des reseaux de processus avec operateur de melange equitable.* TCS 18(2), 173-206 (1982).

[2] BROCK, J.D. & W.B. ACKERMANN, *Scenarios: A model of non-determinate computation*, Proc. Peniscola Colloquium, 252-259. Springer LNCS 107 (1981).

[3] BROY, M., *Fixed Point Theory for Communication and Concurrency*, IFIP TC2 Working Conference, Garmisch-Partenkirchen, 1982.

[4] KAHN, G., *The Semantics of a Simple Language for Parallel Processing,* Proc. IFIP Congress 74, 471-475 (1974).

[5] PARK, D., *On the Semantics of Fair Parallelism,* Proc. Copenhagen Winter School. 504-526, Springer LNCS 86(1980).

[6] PARK, D., *Concurrency and Automata on Infinite Sequences*, in Proc. GI Conference on Theoretical Computer Science, Karlsruhe. Springer LNCS 104 (1981).

[7] WADGE, W.W., *An Extensional Treatment of Dataflow Deadlock,* Proc Evian Symposium, 285-299. Springer LNCS 70 (1979).

VERIFICATION OF CONCURRENT PROGRAMS: A TEMPORAL PROOF SYSTEM

by

ZOHAR MANNA
Computer Science Department
Stanford University
Stanford, CA
and
Applied Mathematics Department
The Weizmann Institute of Science
Rehovot, Israel

AMIR PNUELI
Applied Mathematics Department
The Weizmann Institute of science
Rehovot, Israel

ABSTRACT

A proof system based on temporal logic is presented for proving properties of concurrent programs based on the shared-variables computation model. The system consists of three parts: the general uninterpreted part, the domain dependent part and the program dependent part. In the general part we give a complete proof system for first-order temporal logic with detailed proofs of useful theorems. This logic enables reasoning about general time sequences. The domain dependent part characterizes the special properties of the domain over which the program operates. The program dependent part introduces program axioms which restrict the time sequences considered to be execution sequences of a given program.

The utility of the full system is demonstrated by proving invariance, liveness and precedence properties of several concurrent programs. Derived proof principles for these classes of properties, are obtained and lead to a compact representation of proofs.

This research was supported in part by the National Science Foundation under grants MCS79-09495 and MCS80-06930, by DARPA under Contract N00039-82-C-0250, by the United States Air Force Office of Scientific Research under Grant AFOSR-81-0014, and by the Basic Research Foundation of the Israeli Academy of Sciences.

A. INTRODUCTION

In this work we present a proof system based on temporal logic for proving the properties of concurrent programs. We refer the reader to [MP1] for a more detailed discussion of the computational model of concurrent programs, and the advantages offered by the language of temporal logic in formulating properties of concurrent programs.

1. THE TEMPORAL LANGUAGE: SYNTAX AND SEMANTICS

We first describe the temporal language we are going to use. This language contains special constructs that are suitable for reasoning about programs.

The language uses a set of basic symbols consisting of individual variables and constants, propositions, and function and predicate symbols. The set is partitioned into two subsets: global and local symbols. Intuitively speaking, the *global symbols* denote entities that do not change during a program execution. The *local symbols*, on the other hand, may change their meanings and values in different states throughout the execution. For our purpose, the only local symbols that interest us are local individual variables and propositions. We will have global symbols of all types.

We use the usual set of boolean connectives: $\wedge$, $\vee$, $\supset$, $\equiv$, and $\sim$ together with the equality predicate $=$ and the first-order quantifiers $\forall$ and $\exists$. These operators are referred to as the *classical operators.* The quantifiers $\forall$ and $\exists$ are applied only to global individual variables..

The *modal operators* used are: $\Box$, $\Diamond$, $\bigcirc$, and $\mathcal{U}$, which are called respectively the *always*, *sometime*, *next* and *until* operators. The first three operators are unary while the $\mathcal{U}$ operator is binary. We use the *next* operator, $\bigcirc$, in two different ways — as a temporal operator applied to formulas and as a temporal operator applied to terms.

A model (I, α, σ) for our language consists of a (global) interpretation I, a (global) assignment α and a sequence of states σ.

- The *interpretation* I specifies a nonempty domain D and assigns concrete elements, functions and predicates to the (global) individual constants, function and predicate symbols.
- *The assignment* α assigns a value over the appropriate domain to each of the global individual variables.
- *The sequence* $\sigma = s_0, s_1, \ldots$ is an infinite sequence of states. Each *state* s_i assigns values to the local individual variables and propositions.

For a sequence

$$\sigma = s_0, s_1, \ldots$$

we denote by

$$\sigma^{(i)} = s_i, s_{i+1}, \ldots$$

the i-truncated suffix of σ.

Given a temporal formula w, we present below an inductive definition of the truth value of w in a model (I, α, σ). The value of a subformula or term τ under (I, α, σ) is denoted by $\tau|^{\alpha}_{\sigma}$, with I being implicitly understood.

Consider first the evaluation of terms:

- For a local individual variable or local proposition y:
  $$y|^{\alpha}_{\sigma} = s_0[y],$$
  i.e., the value assigned to y in s_0, the first state of σ.
- For a global individual variable u:
  $$u|^{\alpha}_{\sigma} = \alpha[u],$$
  i.e., the value assigned to u by α.
- For an individual constant the evaluation is given by I:
  $$c|^{\alpha}_{\sigma} = I[c].$$
- For a k-ary function f:
  $$f(t_1, \ldots, t_k)|^{\alpha}_{\sigma} = I[f](t_1|^{\alpha}_{\sigma}, \ldots, t_k|^{\alpha}_{\sigma}),$$
  i.e., the value is given by the application of the interpreted function $I[f]$ to the values of $t_1, \ldots, t_k$ evaluated in the model (I,α,σ).
- For a term t:
  $$(\bigcirc\, t)|^{\alpha}_{\sigma} = t|^{\alpha}_{\sigma^{(1)}},$$
  i.e., the value of $\bigcirc t$ in $\sigma = s_0, s_1, \ldots$ is given by the value of t in the 1-truncated suffix $\sigma^{(1)} = s_1, s_2, \ldots$.

Consider now the evaluation of formulas:

- For a k-ary predicate p (including equality):
  $$p(t_1, \ldots, t_k)|^{\alpha}_{\sigma} = I[p](t_1|^{\alpha}_{\sigma}, \ldots, t_k|^{\alpha}_{\sigma}).$$
  Here again, we first evaluate the arguments in the model and then test $I[p]$ on them.
- For a disjunction:
  $$(w_1 \lor w_2)|^{\alpha}_{\sigma} = \textit{true} \quad \textit{if and only if} \quad w_1|^{\alpha}_{\sigma} = \textit{true} \ \textit{or} \ w_2|^{\alpha}_{\sigma} = \textit{true}.$$
  And similarly for the other binary boolean connectives $\lor$, $\supset$, and $\equiv$.

- For a negation:

  $$(\sim w)|^{\alpha}_{\sigma} = \textit{true} \quad \textit{if and only if} \quad w|^{\alpha}_{\sigma} = \textit{false}.$$

- For a next-time application:

  $$(\bigcirc w)|^{\alpha}_{\sigma} = w|^{\alpha}_{\sigma^{(1)}}.$$

  Thus $\bigcirc w$ means: w will be true in the *next* instant — read "next w".

- For an all-times application:

  $$(\Box w)|^{\alpha}_{\sigma} = \textit{true} \quad \textit{if and only if} \quad \text{for every } k \geq 0,\ w|^{\alpha}_{\sigma^{(k)}} = \textit{true},$$

  i.e., w is true for all suffix sequences of σ. Thus $\Box w$ means: w is true for *all* future instants (including the present) — read "always w" or "henceforth w".

- For a some-time application:

  $$(\Diamond w)|^{\alpha}_{\sigma} = \textit{true} \quad \textit{if and only if} \quad \text{there exists a } k \geq 0 \text{ such that } w|^{\alpha}_{\sigma^{(k)}} = \textit{true},$$

  i.e., w is *true* on at least one suffix of σ. Thus $\Diamond w$ means: w will be true for *some* future instant (possibly the present) — read "sometime w" or "eventually w".

- For an until application:

  $$w_1 \mathcal{U} w_2|^{\alpha}_{\sigma} = \textit{true} \quad \textit{if and only if} \quad \text{for some } k \geq 0,\ w_2|^{\alpha}_{\sigma^{(k)}} = \textit{true} \text{ and for all } i,\ 0 \leq i < k,\ w_1|^{\alpha}_{\sigma^{(i)}} = \textit{true}.$$

  Thus $w_1 \mathcal{U} w_2$ means: there is a future instant in which w_2 holds, and such that *until* that instant w_1 continuously holds — read "w_1 until w_2" ([KAM], [GPSS]).

- For a universal quantification:

  $$(\forall u.w)|^{\alpha}_{\sigma} = \textit{true} \quad \textit{if and only if} \quad \text{for every } d \in D,\ w|^{\alpha'}_{\sigma} = \textit{true},$$

  where $\alpha' = \alpha \circ [u \leftarrow d]$ is the assignment obtained from α by assigning d to u.

- For an existential quantification:

  $$(\exists u.w)|^{\alpha}_{\sigma} = \textit{true} \quad \textit{if and only if} \quad \text{for some } d \in D,\ w|^{\alpha'}_{\sigma} = \textit{true},$$

  where $\alpha' = \alpha \circ [u \leftarrow d]$.

Following are some examples of temporal expressions and their intuitive interpretations:

$u \supset \Diamond v$	If u is presently true, v will eventually become true.
$\Box(u \supset \Diamond v)$	Whenever u becomes true it will eventually be followed by v.
$\Diamond \Box w$	At some future instant w will become permanently true.
$\Diamond(w \wedge \bigcirc \sim w)$	There will be a future instant such that w is true at that instant and false at the next.
$\Box \Diamond w$	Every future instant is followed by a later one in which w is true,

thus w is true infinitely often.

$\Box(u \supset \Box v)$	If u ever becomes true, then v is true at that instant and ever after.
$\Box u \vee (u \mathcal{U} v)$	Either u holds continuously or it holds until an occurrence of v. This is the weak form of the *until* operator that states that u will hold continuously until the first occurrence of v if v ever happens or indefinitely otherwise.
$\Diamond v \supset ((\sim v) \mathcal{U} u)$	If v ever happens, its first occurrence is preceded by (or coincides with) u.

If w is true under the model (I, α, σ), we say that (I, α, σ) ***satisfies*** w or that (I, α, σ) is a ***(satisfying) model for*** w. We denote this by

$$(I, \alpha, \sigma) \models w.$$

A formula w is ***satisfiable*** if there exists a satisfying model for it.

A formula w is ***valid*** if it is true in every model; in this case we write

$$\models w.$$

Sometimes we are interested in a restricted class of models C. A formula w which is true for every model in C is said to be ***C-valid***, denoted by

$$C \models w.$$

Example:

The formula $\Diamond(w_1 \wedge w_2) \supset (\Diamond w_1 \wedge \Diamond w_2)$ is valid, i.e.,

$$\models \Diamond(w_1 \wedge w_2) \supset (\Diamond w_1 \wedge \Diamond w_2).$$

It says that if there exists an instant in which both w_1 and w_2 are true then there exists an instant in which w_1 is true and there exists an instant in which w_2 is true.

Reversing the implication does not yield a valid formula, i.e.,

$$\not\models (\Diamond w_1 \wedge \Diamond w_2) \supset \Diamond(w_1 \wedge w_2).$$

For, consider an interpretation consisting of a sequence of states:

$$\sigma : s_0, s_1, \ldots$$

such that w_1 is true on all odd numbered states and false elsewhere, and w_2 is true on all the even numbered states and false on the odd ones. Then certainly both $\Diamond w_1$ and $\Diamond w_2$ are true on σ, hence $\Diamond w_1 \wedge \Diamond w_2$ is true. On the other hand, there is no state on which both w_1 and w_2 are true simultaneously. Hence $\Diamond(w_1 \wedge w_2)$ is false. Consequently the implication is false under the interpretation σ. ┛

2. THE PROOF SYSTEM

Having defined valid formulas, we naturally look for a deductive system in which validity can be proved. In such a system we take some of the valid formulas as axioms and provide a set of sound inference rules by which we hope to be able to prove the other valid formulas as theorems. A formula w is a ***theorem*** of the system either if it is an axiom of the system or has a ***proof*** in which it is derived from the axioms using the inference rules of the system. We denote the fact that w is a theorem is ***provable*** within the system by $\vdash w$.

Our interest in the temporal logic formalism is mainly motivated by the applicability of this logic to proving properties of concurrent programs. Therefore, apart from developing the general basic logical properties of the operators and their interrelations, we will mostly be interested in properties that are valid over computations of a given concurrent program P. Thus, the notion of validity our system will try to capture is that of a formula being true for all possible computations of the given program, and not necessarily over an arbitrary model. This corresponds to the concept of $\mathfrak{A}(P)$-validity where $\mathfrak{A}(P)$ is the class of all models corresponding to computations of P.

We structure our proof system into three main layers dependent on the universal validity of the theorems that can be derived in each layer. In the first layer, called the *general part*, we deal with the general temporal properties of discrete linear sequences (arbitrary models). Theorems proved in that part are valid for all sequences over arbitrary domains. They universally hold for arbitrary computations of all programs over such domains, as well as for sequences which cannot even be derived as the computations of a program. In the next layer the *domain part*, we restrict our attention to a particular domain D and provide tools for proving validity over models all of which are interpreted over D. The third, most restrictive layer is the *program part*. Here we restrict our attention to a particular program P and develop tools for proving validity only over models whose sequences are legal computations of P.

In [MP5], an extended version of this paper, the program dependent part is proved to be complete relative to the general temporal theory over the data domain. We also show that its dependence on the particular computation model studied is modular, by presenting a similar system for proving properties of CSP programs.

B. GENERAL PART

We start the general part by describing first the axiomatic system for propositional temporal logic in which we do not admit predicates or quantification.

3. THE PROPOSITIONAL TEMPORAL SYSTEM ($\Box, \Diamond, \bigcirc$ AND $\mathcal{U}$)

The proof system for the propositional part consists of the following axioms:

AXIOMS:

A1. $\vdash \sim\Diamond w \equiv \Box\sim w$

A2. $\vdash \Box(w_1 \supset w_2) \supset (\Box w_1 \supset \Box w_2)$

A3. $\vdash \Box w \supset w$

A4. $\vdash \bigcirc\sim w \equiv \sim\bigcirc w$

A5. $\vdash \bigcirc(w_1 \supset w_2) \supset (\bigcirc w_1 \supset \bigcirc w_2)$

A6. $\vdash \Box w \supset \bigcirc w$

A7. $\vdash \Box w \supset \bigcirc\Box w$

A8. $\vdash \Box(w \supset \bigcirc w) \supset (w \supset \Box w)$

A9. $\vdash (w_1 \mathcal{U} w_2) \equiv [w_2 \vee (w_1 \wedge \bigcirc(w_1 \mathcal{U} w_2))]$

A10. $\vdash (w_1 \mathcal{U} w_2) \supset \Diamond w_2.$

Axiom A1 defines $\Diamond$ as the dual of $\Box$; it states that at all times w is false *if and only if* it is not the case that sometimes w holds. Axiom A2 states that if universally w_1 implies w_2 then if at all times w_1 is true then so is w_2. Axiom A3 establishes the present as part of the future by stating that if w is true at all future instants it must be true at the present. Axiom A4 establishes $\bigcirc$ as self-dual. Consequently it implies that the next instant exists and is unique, and restricts our models to linear sequences (no branching). Axiom A5 is the analogue of A2 for the $\bigcirc$ operator. Axiom A6 states that the next instant is one of the future states. Axiom A7 states that if w holds in all future instants it also holds in all instants which lie in the future of the next instant. Axiom A8 is the "computational induction" axiom; it states that if a property is inherited over one step transitions, it is invariant over any suffix sequence whose first state satisfies w. Axiom A9 characterizes the ***until*** operator by distributing its effect into what is implied for the present and what is implied for the next instant. Axiom A10 simply states that "w_1 until w_2" implies that w_2 will eventually happen.

INFERENCE RULES:

R1. *Propositional Tautology* — PT

If u is an instance of a propositional tautology then $\vdash u$

R2. *Modus Ponens* — MP

If $\vdash u \supset v$ and $\vdash u$ then $\vdash v$

R3. $\Box$ *Insertion* — $\Box$I

If $\vdash u$ then $\vdash \Box u$

All these rules are sound. The soundness of R1 and R2 is obvious. Note that in R1 we also include temporal instances of tautologies; we may substitute an arbitrary temporal formula for a proposition letter in obtaining an instance. For example, the formula $\Box w \supset \Box w$ is a temporal instance of the tautology $p \supset p$. To justify R3, we recall that validity of w means that w is true in *all* models, hence $\Box w$ is also valid.

DERIVED RULES AND THEOREMS:

Before giving some theorems that can be proved in this system, we develop several useful derived rules:

Propositional Reasoning — PR

$$\frac{\vdash (u_1 \wedge u_2 \wedge \ \ldots \wedge u_n) \supset v \qquad \vdash u_1,\ \vdash u_2,\ \ldots,\ \text{and}\ \vdash u_n}{\vdash v}$$

The notation above is used to describe inference rules. It has the general form

$$\frac{\vdash \varphi_1,\ \vdash \varphi_2,\ \ldots,\ \vdash \varphi_m}{\vdash \psi}$$

and means that if we have already proved $\varphi_1, \ldots, \varphi_m$ (the *assumptions* or *premises* of the rule), we are allowed by this rule to infer ψ (the *conclusion* or *consequent* of the rule).

Proof:

The rule PR follows from the propositional tautology (Rule R1)

$$\vdash [(u_1 \wedge u_2 \wedge \ \ldots \ \wedge u_n) \supset v] \supset [u_1 \supset (u_2 \supset (\ldots(u_n \supset v)\ldots))]$$

by applying MP (Rule R2) $n+1$ times. ■

Whenever we apply this derived rule without explicitly indicating the premise

$$\vdash (u_1 \wedge u_2 \wedge \ldots \wedge u_n) \supset v,$$

it means that the premise is an instance of a propositional tautology.

○ *Insertion* — ○I

$$\frac{\vdash u}{\vdash \bigcirc u}$$

Proof:

1. $\vdash u$ given
2. $\vdash \Box u$ by □I
3. $\vdash \bigcirc u$ by A6 and MP ∎

The first theorem that we derive in the system is:

T1. $\vdash w \supset \Diamond w$

Proof:

1. $\vdash (\Box \sim w) \supset \sim w$ by A3
2. $\vdash w \supset (\sim \Box \sim w)$ by PR
3. $\vdash w \supset \Diamond w$ by A1 and PR ∎

The theorem implies (by MP) the derived rule

◇ *Insertion* — ◇I

$$\frac{\vdash u}{\vdash \Diamond u}$$

T2. $\vdash \bigcirc w \supset \Diamond w$

Proof:

1. $\vdash (\Box \sim w) \supset (\bigcirc \sim w)$ by A6

2. $\vdash (\sim \bigcirc \sim w) \supset (\sim \Box \sim w)$ by PR

3. $\vdash \bigcirc w \supset \Diamond w$ by A1, A4, and PR ∎

The following three rules (and a similar rule for the *until* operator presented later) show that all the temporal operators are monotonic in the sense that an argument may be replaced by a weaker statement yielding a weaker expression.

$\Box\Box$ *Rules*

$$\text{(a)}\ \frac{\vdash u \supset v}{\vdash \Box u \supset \Box v} \qquad \text{(b)}\ \frac{\vdash u \equiv v}{\vdash \Box u \equiv \Box v}$$

Proof of (a):

1. $\vdash u \supset v$ given
2. $\vdash \Box(u \supset v)$ by $\Box$I
3. $\vdash \Box(u \supset v) \supset (\Box u \supset \Box v)$ by A2
4. $\vdash \Box u \supset \Box v$ by 2, 3 and MP

Rule (b) then follows by propositional reasoning by using the tautology

$$[(u \supset v) \wedge (v \supset u)] \equiv (u \equiv v).$$ ∎

$\Diamond\Diamond$ *Rules*

$$\text{(a)}\ \frac{\vdash u \supset v}{\vdash \Diamond u \supset \Diamond v} \qquad \text{(b)}\ \frac{\vdash u \equiv v}{\vdash \Diamond u \equiv \Diamond v}$$

Proof of (a):

1. $\vdash u \supset v$ given
2. $\vdash \sim v \supset \sim u$ by PR
3. $\vdash \Box \sim v \supset \Box \sim u$ by $\Box\Box$
4. $\vdash \sim \Diamond v \supset \sim \Diamond u$ by A1 and PR
5. $\vdash \Diamond u \supset \Diamond v$ by PR

Rule (b) then follows by propositional reasoning. ∎

$\bigcirc\bigcirc$ *Rules*

(a) $$\frac{\vdash u \supset v}{\vdash \bigcirc u \supset \bigcirc v}$$ (b) $$\frac{\vdash u \equiv v}{\vdash \bigcirc u \equiv \bigcirc v}$$

Proof of (a):

1. $\vdash u \supset v$ given
2. $\vdash \bigcirc(u \supset v)$ by $\bigcirc$I
3. $\vdash \bigcirc u \supset \bigcirc v$ by A5 and MP

Rule (b) follows by propositional reasoning. $\lrcorner$

Computational Induction Rule — CI

$$\frac{\vdash u \supset \bigcirc u}{\vdash u \supset \Box u}$$

Proof:

1. $\vdash u \supset \bigcirc u$ given
2. $\vdash \Box(u \supset \bigcirc u)$ by $\Box$I
3. $\vdash \Box(u \supset \bigcirc u) \supset (u \supset \Box u)$ by A8
4. $\vdash u \supset \Box u$ by 2, 3 and MP $\lrcorner$

Derived Computational Induction Rule — DCI

$$\frac{\vdash u \supset (v \wedge \bigcirc u)}{\vdash u \supset \Box v}$$

Proof:

1. $\vdash u \supset (v \wedge \bigcirc u)$ given
2. $\vdash u \supset \bigcirc u$ by PR
3. $\vdash u \supset \Box u$ by CI
4. $\vdash u \supset v$ by 1 and PR
5. $\vdash \Box u \supset \Box v$ by $\Box\Box$

6. $\vdash u \supset \Box v$ by 3, 5 and PR ∎

The following two theorems show that the $\Box$ and $\Diamond$ operators are both idempotent:

T3. $\vdash \Box w \equiv \Box\Box w$

Proof:

1. $\vdash \Box\Box w \supset \Box w$ by A3
2. $\vdash \Box w \supset \bigcirc\Box w$ by A7
3. $\vdash \Box w \supset \Box\Box w$ by CI
4. $\vdash \Box w \equiv \Box\Box w$ by 1, 3 and PR ∎

T4. $\vdash \Diamond w \equiv \Diamond\Diamond w$

Proof:

1. $\vdash \sim\Diamond w \equiv \Box{\sim}w$ by A1
2. $\vdash \Box{\sim}w \equiv \Box\Box{\sim}w$ by T3
3. $\vdash \Box{\sim}\Diamond w \equiv \Box\Box{\sim}w$ by 1 and $\Box\Box$
4. $\vdash \Box{\sim}\Diamond w \equiv {\sim}\Diamond\Diamond w$ by A1
5. $\vdash \sim\Diamond w \equiv {\sim}\Diamond\Diamond w$ by 1, 2, 3, 4 and PR
6. $\vdash \Diamond w \equiv \Diamond\Diamond w$ by PR ∎

Because of these last two theorems we can collapse any string of consecutive identical modalities such as $\Box \ldots \Box$ or $\Diamond \ldots \Diamond$ into a single modality of the same type.

The following theorem establishes that $\Box$ is the dual of $\Diamond$. Note that A1 states that $\Diamond$ is the dual of $\Box$, i.e., $\Diamond w \equiv {\sim}\Box{\sim}w$.

T5. $\vdash (\Diamond{\sim}w) \equiv ({\sim}\Box w)$

Proof:

1. $\vdash ({\sim}{\sim}w) \equiv w$ by PT

2. $\vdash (\Box \sim \sim w) \equiv \Box w$ by $\Box\Box$
3. $\vdash (\sim \Diamond \sim w) \equiv \Box w$ by A1 and PR
4. $\vdash (\Diamond \sim w) \equiv (\sim \Box w)$ by PR $\lrcorner$

T6. $\Box(w_1 \supset w_2) \supset (\Diamond w_1 \supset \Diamond w_2)$

Proof:

1. $\vdash (w_1 \supset w_2) \equiv (\sim w_2 \supset \sim w_1)$ by PT
2. $\vdash \Box(w_1 \supset w_2) \equiv \Box(\sim w_2 \supset \sim w_1)$ by $\Box\Box$
3. $\vdash \Box(\sim w_2 \supset \sim w_1) \supset (\Box \sim w_2 \supset \Box \sim w_1)$ by A2
4. $\vdash (\Box \sim w_2 \supset \Box \sim w_1) \equiv (\sim \Diamond w_2 \supset \sim \Diamond w_1)$ by A1 and PR
5. $\vdash (\sim \Diamond w_2 \supset \sim \Diamond w_1) \equiv (\Diamond w_1 \supset \Diamond w_2)$ by PT
6. $\vdash \Box(w_1 \supset w_2) \supset (\Diamond w_1 \supset \Diamond w_2)$ by 2, 3, 4, 5 and PR $\lrcorner$

The following theorems show the interaction between the temporal and the boolean operators.

T7. $\vdash \Box(w_1 \wedge w_2) \equiv (\Box w_1 \wedge \Box w_2)$

Proof:

1. $\vdash (w_1 \wedge w_2) \supset w_1$ by PT
2. $\vdash \Box(w_1 \wedge w_2) \supset \Box w_1$ by $\Box\Box$
3. $\vdash (w_1 \wedge w_2) \supset w_2$ by PT
4. $\vdash \Box(w_1 \wedge w_2) \supset \Box w_2$ by $\Box\Box$
5. $\vdash \Box(w_1 \wedge w_2) \supset (\Box w_1 \wedge \Box w_2)$ by 2, 4 and PR

6. $\vdash w_1 \supset (w_2 \supset w_1 \wedge w_2)$ by PT
7. $\vdash \Box w_1 \supset \Box(w_2 \supset (w_1 \wedge w_2))$ by $\Box\Box$
8. $\vdash \Box(w_2 \supset (w_1 \wedge w_2)) \supset (\Box w_2 \supset \Box(w_1 \wedge w_2))$ by A2
9. $\vdash \Box w_1 \supset (\Box w_2 \supset \Box(w_1 \wedge w_2))$ by 7, 8 and PR
10. $\vdash (\Box w_1 \wedge \Box w_2) \supset \Box(w_1 \wedge w_2)$ by PR

11. $\vdash \Box(w_1 \wedge w_2) \equiv (\Box w_1 \wedge \Box w_2)$ — by 5, 10 and PR ∎

T8. $\vdash \Diamond(w_1 \vee w_2) \equiv (\Diamond w_1 \vee \Diamond w_2)$

Proof:

1. $\vdash \Box\sim(w_1 \vee w_2) \equiv \Box(\sim w_1 \wedge \sim w_2)$ — by PT and □ □
2. $\vdash \Box(\sim w_1 \wedge \sim w_2) \equiv (\Box\sim w_1 \wedge \Box\sim w_2)$ — by **T7**
3. $\vdash (\Box\sim w_1 \wedge \Box\sim w_2) \equiv \sim(\sim\Box\sim w_1 \vee \sim\Box\sim w_2)$ — by **PR**
4. $\vdash \Box\sim(w_1 \vee w_2) \equiv \sim(\sim\Box\sim w_1 \vee \sim\Box\sim w_2)$ — by 1, 2, 3 and **PR**
5. $\vdash \sim\Diamond(w_1 \vee w_2) \equiv \sim(\Diamond w_1 \vee \Diamond w_2)$ — by A1 and **PR**
6. $\vdash \Diamond(w_1 \vee w_2) \equiv (\Diamond w_1 \vee \Diamond w_2)$ — by PR ∎

Note that because of the universal character of $\Box$ it can be distributed over $\wedge$ (Theorem **T7**), while $\Diamond$, which is of existential character can be distributed over $\vee$ (Theorem T8). Next, we show that interchanging a temporal operator with a boolean operator of the opposite character yields implication in one direction only; the implication is not necessarily true in the other direction.

T9. $\vdash (\Box w_1 \vee \Box w_2) \supset \Box(w_1 \vee w_2)$

Proof:

1. $\vdash \Box w_1 \supset \Box(w_1 \vee w_2)$ — by PT and □ □
2. $\vdash \Box w_2 \supset \Box(w_1 \vee w_2)$ — by PT and □ □
3. $\vdash (\Box w_1 \vee \Box w_2) \supset \Box(w_1 \vee w_2)$ — by 1, 2 and PR ∎

T10. $\vdash \Diamond(w_1 \wedge w_2) \supset (\Diamond w_1 \wedge \Diamond w_2)$

Proof:

1. $\vdash \Diamond(w_1 \wedge w_2) \supset \Diamond w_1$ — by PT and ◇ ◇
2. $\vdash \Diamond(w_1 \wedge w_2) \supset \Diamond w_2$ — by PT and ◇ ◇
3. $\vdash \Diamond(w_1 \wedge w_2) \supset (\Diamond w_1 \wedge \Diamond w_2)$ — by 1, 2 and PR ∎

T11. $\vdash (\Box w_1 \wedge \Diamond w_2) \supset \Diamond(w_1 \wedge w_2)$

Proof:

1. $\vdash w_1 \supset (w_2 \supset (w_1 \wedge w_2))$ by PT
2. $\vdash \Box w_1 \supset \Box(w_2 \supset (w_1 \wedge w_2))$ by $\Box\Box$
3. $\vdash \Box(w_2 \supset (w_1 \wedge w_2)) \supset (\Diamond w_2 \supset \Diamond(w_1 \wedge w_2))$ by T6
4. $\vdash \Box w_1 \supset (\Diamond w_2 \supset \Diamond(w_1 \wedge w_2))$ by 2, 3 and PR
5. $\vdash (\Box w_1 \wedge \Diamond w_2) \supset \Diamond(w_1 \wedge w_2)$ by PR ∎

Next we consider the commutativity properties of the *next* operator $\bigcirc$. In view of A4, $\bigcirc$ is self-dual and can be considered to be of both existential and universal character. Indeed it commutes with every other boolean or temporal operator as well as with quantifiers.

T12. $\vdash \bigcirc(w_1 \wedge w_2) \equiv (\bigcirc w_1 \wedge \bigcirc w_2)$

Proof:

1. $\vdash w_1 \supset (w_2 \supset (w_1 \wedge w_2))$ by PT
2. $\vdash \bigcirc w_1 \supset \bigcirc(w_2 \supset (w_1 \wedge w_2))$ by $\bigcirc\bigcirc$
3. $\vdash \bigcirc(w_2 \supset (w_1 \wedge w_2)) \supset (\bigcirc w_2 \supset \bigcirc(w_1 \wedge w_2))$ by A5
4. $\vdash \bigcirc w_1 \wedge (\bigcirc w_2 \supset \bigcirc(w_1 \wedge w_2))$ by 2, 3 and PR
5. $\vdash (\bigcirc w_1 \wedge \bigcirc w_2) \supset \bigcirc(w_1 \wedge w_2)$ by PR

6. $\vdash (w_1 \wedge w_2) \supset w_1$ by PT
7. $\vdash \bigcirc(w_1 \wedge w_2) \supset \bigcirc w_1$ by $\bigcirc\bigcirc$
8. $\vdash (w_1 \wedge w_2) \supset w_2$ by PT
9. $\vdash \bigcirc(w_1 \wedge w_2) \supset \bigcirc w_2$ by $\bigcirc\bigcirc$
10. $\vdash \bigcirc(w_1 \wedge w_2) \supset (\bigcirc w_1 \wedge \bigcirc w_2)$ by 7, 9 and PR

11. $\vdash \bigcirc(w_1 \wedge w_2) \equiv (\bigcirc w_1 \wedge \bigcirc w_2)$ by 5, 10 and PR ∎

T13. $\vdash \bigcirc(w_1 \vee w_2) \equiv (\bigcirc w_1 \vee \bigcirc w_2)$

Proof:

1. $\vdash \bigcirc(\sim w_1 \wedge \sim w_2) \equiv [(\bigcirc \sim w_1) \wedge (\bigcirc \sim w_2)]$ by T12
2. $\vdash \bigcirc(\sim w_1 \wedge \sim w_2) \equiv [(\sim \bigcirc w_1) \wedge (\sim \bigcirc w_2)]$ by A4 and PR
3. $\vdash \bigcirc \sim(w_1 \vee w_2) \equiv [(\sim \bigcirc w_1) \wedge (\sim \bigcirc w_2)]$ by $\bigcirc\bigcirc$ and PR
4. $\vdash \sim \bigcirc(w_1 \vee w_2) \equiv \sim(\bigcirc w_1 \vee \bigcirc w_2)$ by A4 and PR
5. $\vdash \bigcirc(w_1 \vee w_2) \equiv (\bigcirc w_1 \vee \bigcirc w_2)$ by PR ▟

T14. $\vdash \bigcirc(w_1 \supset w_2) \equiv (\bigcirc w_1 \supset \bigcirc w_2)$

Proof:

1. $\vdash \bigcirc(\sim w_1 \vee w_2) \equiv (\bigcirc \sim w_1) \vee (\bigcirc w_2)$ by T13
2. $\vdash \bigcirc(\sim w_1 \vee w_2) \equiv (\sim \bigcirc w_1) \vee (\bigcirc w_2)$ by A4 and PR
3. $\vdash \bigcirc(w_1 \supset w_2) \equiv (\bigcirc w_1 \supset \bigcirc w_2)$ by $\bigcirc\bigcirc$ and PR ▟

T15. $\vdash \bigcirc(w_1 \equiv w_2) \equiv (\bigcirc w_1 \equiv \bigcirc w_2)$

Proof:

1. $\vdash [\bigcirc(w_1 \supset w_2) \wedge \bigcirc(w_2 \supset w_1)] \equiv [(\bigcirc w_1 \supset \bigcirc w_2) \wedge (\bigcirc w_2 \supset \bigcirc w_1)]$ by T14 and PR
2. $\vdash \bigcirc[(w_1 \supset w_2) \wedge (w_2 \supset w_1)] \equiv [(\bigcirc w_1 \supset \bigcirc w_2) \wedge (\bigcirc w_2 \supset \bigcirc w_1)]$ by T12 and PR
3. $\vdash \bigcirc(w_1 \equiv w_2) \equiv (\bigcirc w_1 \equiv \bigcirc w_2)$ by $\bigcirc\bigcirc$ and PR ▟

The previous theorems show that the next operator, $\bigcirc$, commutes with each of the boolean operators. The following two theorems establish commutation of $\bigcirc$ with the temporal operators $\Box$ and $\Diamond$.

T16. $\vdash \bigcirc \Box w \equiv \Box \bigcirc w$

Proof:

1. $\vdash \bigcirc w \supset (w \supset \bigcirc w)$ by PT

2. $\vdash \Box\bigcirc w \supset \Box(w \supset \bigcirc w)$ by $\Box\Box$

3. $\vdash \Box(w \supset \bigcirc w) \supset \bigcirc\Box(w \supset \bigcirc w)$ by A7

4. $\vdash \bigcirc\Box(w \supset \bigcirc w) \supset \bigcirc(w \supset \Box w)$ by A8 and $\bigcirc\bigcirc$

5. $\vdash \bigcirc(w \supset \Box w) \supset (\bigcirc w \supset \bigcirc\Box w)$ by A5

6. $\vdash \Box\bigcirc w \supset (\bigcirc w \supset \bigcirc\Box w)$ by 2, 3, 4, 5 and PR

7. $\vdash \Box\bigcirc w \supset \bigcirc w$ by A3

8. $\vdash \Box\bigcirc w \supset \bigcirc\Box w$ by 6, 7 and PR

9. $\vdash \bigcirc\Box w \supset \bigcirc\bigcirc\Box w$ by A7 and $\bigcirc\bigcirc$

10. $\vdash \bigcirc\Box w \supset \Box\bigcirc\Box w$ by CI

11. $\vdash \bigcirc\Box w \supset \bigcirc w$ by A3 and $\bigcirc\bigcirc$

12. $\vdash \Box\bigcirc\Box w \supset \Box\bigcirc w$ by $\Box\Box$

13. $\vdash \bigcirc\Box w \supset \Box\bigcirc w$ by 10, 12 and PR

14. $\vdash \bigcirc\Box w \equiv \Box\bigcirc w$ by 8, 13 and PR ┛

T17. $\vdash \bigcirc\Diamond w \equiv \Diamond\bigcirc w$

Proof:

1. $\vdash \bigcirc\Box{\sim}w \equiv \Box\bigcirc{\sim}w$ by T16

2. $\vdash {\sim}\bigcirc\Diamond w \equiv {\sim}\Diamond\bigcirc w$ by A1, A4, $\Box\Box$, $\bigcirc\bigcirc$ and PR

3. $\vdash \bigcirc\Diamond w \equiv \Diamond\bigcirc w$ by PR ┛

T18. $\vdash \Box\Diamond\Box w \equiv \Diamond\Box w$

Proof:

1. $\vdash \Box\Diamond\Box w \supset \Diamond\Box w$ by A3

2. $\vdash \Box w \supset \bigcirc\Box w$ by A7

3. $\vdash \Diamond\Box w \supset \Diamond\bigcirc\Box w$ by $\Diamond\Diamond$

4. $\vdash \Diamond\bigcirc\Box w \supset \bigcirc\Diamond\Box w$ by T17 and PR

5. $\vdash \Diamond\Box w \supset \bigcirc\Diamond\Box w$ by 3, 4 and PR

6. $\vdash \Diamond\Box w \supset \Box\Diamond\Box w$ by CI

7. $\vdash \Box\Diamond\Box w \equiv \Diamond\Box w$ by 1, 6 and PR ∎

T19. $\vdash \Diamond\Box\Diamond w \equiv \Box\Diamond w$

Proof: By duality from T18.

These last two theorems together with T3 and T4 ($\Box\Box w \equiv \Box w$ and $\Diamond\Diamond w \equiv \Diamond w$, respectively) give us a normal prefix form for a string of the form

$$m_1 m_2 \ldots m_k(w),$$

where each m_i is either $\Box$ or $\Diamond$. We use first T2 and T3 to collapse any substring of the form $\Box^n$ and $\Diamond^n$ to a single $\Box$ or $\Diamond$. What remains must be a string of alternating $\Box$ and $\Diamond$. If it contains more than one operator then it is equivalent by T18 and T19 to a string with just two operators — the last two. Consequently any string such as the above must be equivalent to one of the following four possibilities:

$$\Box w, \quad \Diamond w, \quad \Box\Diamond w \quad \text{or} \quad \Diamond\Box w.$$

In the more general case that the string also contains some occurrences of the next-time operator $\bigcirc$, we may use the commutation of $\bigcirc$ with both $\Box$ and $\Diamond$ to obtain the four normal forms:

$$\bigcirc^k\Box w, \quad \bigcirc^k\Diamond w, \quad \bigcirc^k\Box\Diamond w \quad \text{and} \quad \bigcirc^k\Diamond\Box w$$

for some $k \geq 0$.

T20. $\vdash \Box w \equiv (w \wedge \bigcirc\Box w)$

Proof:

1. $\vdash \Box w \supset w$ by A3

2. $\vdash \Box w \supset \bigcirc\Box w$ by A7

3. $\vdash \Box w \supset (w \wedge \bigcirc\Box w)$ by 1, 2 and PR

4. $\vdash \bigcirc\Box w \supset \bigcirc(w \wedge \bigcirc\Box w)$ by $\bigcirc\bigcirc$

5. $\vdash (w \wedge \bigcirc\Box w) \supset \bigcirc(w \wedge \bigcirc\Box w)$ by PR

6. $\vdash (w \land \bigcirc\Box w) \supset \Box(w \land \bigcirc\Box w)$ by CI
7. $\vdash \Box(w \land \bigcirc\Box w) \supset \Box w$ by PT and $\Box\Box$
8. $\vdash (w \land \bigcirc\Box w) \supset \Box w$ by 6, 7 and PR
9. $\vdash \Box w \equiv (w \land \bigcirc\Box w)$ by 3, 8 and PR ∎

T21. $\vdash \Diamond w \equiv (w \lor \bigcirc\Diamond w)$

Proof:

1. $\vdash \Box{\sim}w \equiv ({\sim}w \land \bigcirc\Box{\sim}w)$ by T20
2. $\vdash {\sim}\Diamond w \equiv {\sim}(w \lor {\sim}\bigcirc\Box{\sim}w)$ by A1 and PR
3. $\vdash {\sim}\bigcirc\Box{\sim}w \equiv \bigcirc\Diamond w$ by A4, A1, $\bigcirc\bigcirc$ and PR
4. $\vdash \Diamond w \equiv (w \lor \bigcirc\Diamond w)$ by 2, 3 and PR ∎

Theorems T20 and T21 give a fixpoint characterization of the $\Box$ and $\Diamond$ operators respectively. They each give an equation using only boolean operators, the formula w and the operator $\bigcirc$. The solutions to these equations are $\Box w$ and $\Diamond w$ respectively. This shows that in some sense $\bigcirc$ is the most basic operator since the other operators may be defined by means of fixpoint equations using $\bigcirc$. Axiom A9 similarly characterizes the $\mathfrak{U}$ operator by a fixpoint equation.

T22. $\vdash (w \land \Diamond{\sim}w) \supset \Diamond(w \land \bigcirc{\sim}w)$.

This is the dual of the "computational induction" axiom A8. It states that if w is true now and is false sometime in the future, then there exists some instant such that w is true at that instant and false at the next.

Proof:

1. $\vdash \Box(w \supset \bigcirc w) \supset (w \supset \Box w)$ by A8
2. $\vdash {\sim}(w \supset \Box w) \supset {\sim}\Box(w \supset \bigcirc w)$ by PR
3. $\vdash (w \land {\sim}\Box w) \supset \Diamond{\sim}(w \supset \bigcirc w)$ by T5 and PR
4. $\vdash \Diamond{\sim}(w \supset \bigcirc w) \equiv \Diamond(w \land {\sim}\bigcirc w)$ by PT and $\Diamond\Diamond$
5. $\vdash (w \land {\sim}\Box w) \supset \Diamond(w \land {\sim}\bigcirc w)$ by 3, 4 and PR
6. $\vdash (w \land \Diamond{\sim}w) \supset \Diamond(w \land \bigcirc{\sim}w)$ by T5, A4 and PR ∎

The following derived rules correspond to proof rules existing in most axiomatic verification systems:

Consequence Rules

$$\begin{array}{lll}
\Box\text{Q } \textit{rule} & \Diamond\text{Q } \textit{rule} & \bigcirc\text{Q } \textit{rule} \\
 & & \\
\vdash u_1 \supset u_2 & \vdash u_1 \supset u_2 & \vdash u_1 \supset u_2 \\
\vdash u_2 \supset \Box v_1 & \vdash u_2 \supset \Diamond v_1 & \vdash u_2 \supset \bigcirc v_1 \\
\vdash v_1 \supset v_2 & \vdash v_1 \supset v_2 & \vdash v_1 \supset v_2 \\
\hline
\vdash u_1 \supset \Box v_2 & \vdash u_1 \supset \Diamond v_2 & \vdash u_1 \supset \bigcirc v_2
\end{array}$$

Proof of ◇Q:

1. $\vdash u_1 \supset u_2$ given
2. $\vdash u_2 \supset \Diamond v_1$ given
3 $\vdash v_1 \supset v_2$ given
4. $\vdash \Diamond v_1 \supset \Diamond v_2$ by 3 and ◇ ◇
5. $\vdash u_1 \supset \Diamond v_2$ by 1, 2, 4 and PR ∎

The □Q and ○Q rules are proved similarly by the □ □-rule and ○ ○-rule, respectively.

Concatenation Rules

$$\begin{array}{ll}
\Box\text{C } \textit{rule} & \Diamond\text{C } \textit{rule} \\
 & \\
\vdash u \supset \Box v & \vdash u \supset \Diamond v \\
\vdash v \supset \Box w & \vdash v \supset \Diamond w \\
\hline
\vdash u \supset \Box w & \vdash u \supset \Diamond w
\end{array}$$

Proof of □C:

1. $\vdash u \supset \Box v$ given
2. $\vdash v \supset \Box w$ given
3. $\vdash \Box v \supset \Box\Box w$ by 2 and □ □
4. $\vdash \Box v \supset \Box w$ by T3 and PR
5. $\vdash u \supset \Box w$ by 1, 4 and PR ∎

The ◇C rule is proved similarly by the ◇ ◇-rule. Note that the corresponding ○C rule does not hold.

UNTIL DERIVED RULES AND THEOREMS:

Right Until Introduction — R$\mathfrak{U}$I

$$\begin{array}{l} \vdash w \supset \Diamond v \\ \vdash w \supset [v \vee (u \wedge \bigcirc w)] \\ \hline \vdash w \supset (u \mathfrak{U} v) \end{array}$$

Proof:

1. $\vdash w \supset \Diamond v$ given
2. $\vdash w \supset [v \vee (u \wedge \bigcirc w)]$ given
3. $\vdash [v \vee (u \wedge \bigcirc(u \mathfrak{U} v))] \supset (u \mathfrak{U} v)$ by A9 and PR
4. $\vdash \sim(u \mathfrak{U} v) \supset [\sim v \wedge (\sim u \vee \bigcirc \sim(u \mathfrak{U} v))]$ by A4 and PR
5. $\vdash [w \wedge \sim(u \mathfrak{U} v)] \supset [\sim v \wedge \bigcirc w \wedge \bigcirc \sim(u \mathfrak{U} v)]$ by 2, 4 and PR
6. $\vdash [w \wedge \sim(u \mathfrak{U} v)] \supset [\sim v \wedge \bigcirc(w \wedge \sim(u \mathfrak{U} v))]$ by T12 and PR
7. $\vdash [w \wedge \sim(u \mathfrak{U} v)] \supset \Box \sim v$ by DCI, taking u to be $w \wedge \sim(u \mathfrak{U} v)$ and v to be $\sim v$
8. $\vdash [w \wedge \sim(u \mathfrak{U} v)] \supset \sim \Box \sim v$ by 1, T5 and PR
9. $\vdash w \supset (u \mathfrak{U} v)$ by 7, 8 and PR ∎

The R$\mathfrak{U}$I rule, together with axioms A9 and A10, can be viewed as a characterization of the $u \mathfrak{U} v$ construct as a *maximal* solution of the two implications:

$$(*) \quad \begin{cases} x \supset [v \vee (u \wedge \bigcirc x)] \\ x \supset \Diamond v \end{cases}$$

The ordering by which maximality is defined is the ordering induced by defining ***false*** $\sqsubset$ ***true***.

Axioms A9 and A10 imply that

$$(u \mathfrak{U} v) \supset [v \vee (u \wedge \bigcirc u \mathfrak{U} v)]$$

$$(u \mathfrak{U} v) \supset \Diamond v$$

Thus they show $x = u \mathfrak{U} v$ to be a solution of the implications $(*)$. The rule R$\mathfrak{U}$I states that any other solution $x = w$ must satisfy $w \supset (u \mathfrak{U} v)$ which implies that whenever w is true so is $u \mathfrak{U} v$. Interpreted in our ordering this is representable as $w \sqsubseteq (u \mathfrak{U} v)$. Thus $x = u \mathfrak{U} v$ is the maximal solution to $(*)$.

An intuitive explanation as to why $u \mathfrak{U} v$ is indeed the maximal solution of $(*)$ can be given as follows:

Let w be any proposition satisfying $(*)$ everywhere in a sequence $\sigma = s_0, s_1, \ldots$. We note that $(*)$ may have many solutions. In particular $x = \mathit{false}$ is a trivial solution. However an obvious property of every solution w is that if w is true in some state s_i, this state must satisfy u and the next state s_{i+1} must also satisfy w unless s_i satisfies v. Thus once w is true it can stop being true only in a v-state. In view of the second implication such a v-state is guaranteed. Consequently whenever w is true in a state, $u\mathcal{U}v$ must also be true in that state.

Left Until Introduction — L$\mathcal{U}$I

$$\frac{\vdash [v \lor (u \land \bigcirc w)] \supset w}{\vdash (u\mathcal{U}v) \supset w}$$

Proof:

1. $\vdash [v \lor (u \land \bigcirc w)] \supset w$ given
2. $\vdash u\mathcal{U}v \supset [v \lor (u \land \bigcirc(u\mathcal{U}v))]$ by A9 and PR
3. $\vdash \sim w \supset [\sim v \land (\sim u \lor \bigcirc \sim w)]$ by 1, A4 and PR
4. $\vdash [u\mathcal{U}v \land \sim w] \supset [\sim v \land u \land \bigcirc(u\mathcal{U}v) \land \bigcirc \sim w]$ by 2, 3 and PR
5. $\vdash [u\mathcal{U}v \land \sim w] \supset [\bigcirc(u\mathcal{U}v) \land \bigcirc \sim w]$ by PR
6. $\vdash [u\mathcal{U}v \land \sim w] \supset \bigcirc(u\mathcal{U}v \land \sim w)$ by T12 and PR
7. $\vdash [u\mathcal{U}v \land \sim w] \supset \Box(u\mathcal{U}v \land \sim w)$ by CI
8. $\vdash [u\mathcal{U}v \land \sim w] \supset \sim v$ by 3 and PR
9. $\vdash \Box(u\mathcal{U}v \land \sim w) \supset \Box \sim v$ by $\Box\Box$
10. $\vdash [u\mathcal{U}v \land \sim w] \supset \sim \Diamond v$ by 7, 9, A1 and PR
11. $\vdash [u\mathcal{U}v \land \sim w] \supset \Diamond v$ by A10 and PR
12. $\vdash u\mathcal{U}v \supset w$ by 10, 11 and PR ■

The L$\mathcal{U}$I rule, together with axiom A9, can be viewed as a characterization of the $u\mathcal{U}v$ construct as the *minimal* solution of the implication:

$$(**) \quad [v \lor (u \land \bigcirc x)] \supset x$$

Axiom A9 implies that $x = u\mathcal{U}v$ is a solution of $(**)$. The L$\mathcal{U}$I rule states that any other solution of $(**)$, $x = w$, is implied by $u\mathcal{U}v$. This means that whenever $u\mathcal{U}v$ is true so is w, which is interpretable in our ordering as $u\mathcal{U}v \sqsubseteq w$. Thus $u\mathcal{U}v$ is the minimal of all possible solutions.

Note that $(**)$ possesses many solutions. In particular $x = \mathit{true}$ is a trivial solution. However, the minimal solution is unique and is given by $u\mathcal{U}v$.

$\mathcal{U}\mathcal{U}$ *Rules*

$$
\text{(a)}\ \frac{\vdash u_1 \supset u_2 \qquad \vdash v_1 \supset v_2}{\vdash u_1 \mathcal{U} v_1 \supset u_2 \mathcal{U} v_2}
\qquad\qquad
\text{(b)}\ \frac{\vdash u_1 \equiv u_2 \qquad \vdash v_1 \equiv v_2}{\vdash u_1 \mathcal{U} v_1 \equiv u_2 \mathcal{U} v_2}
$$

Proof of (a):

1. $\vdash u_1 \supset u_2$ — given
2. $\vdash v_1 \supset v_2$ — given
3. $\vdash [v_2 \vee (u_2 \wedge \bigcirc(u_2 \mathcal{U} v_2))] \supset u_2 \mathcal{U} v_2$ — by A9
4. $\vdash [v_1 \vee (u_1 \wedge \bigcirc(u_2 \mathcal{U} v_2))] \supset u_2 \mathcal{U} v_2$ — by 1, 2, 3 and PR
5. $\vdash u_1 \mathcal{U} v_1 \supset u_2 \mathcal{U} v_2$ — by L$\mathcal{U}$I

The proof of part (b) follows from (a) by propositional reasoning and the symmetric application of (a). ▟

This rule together with the $\Box\Box$, $\Diamond\Diamond$ and $\bigcirc\bigcirc$ rules show that all the temporal operators are monotonic in all their arguments.

T23. $\vdash (\sim w) \mathcal{U} w \equiv \Diamond w$

Proof:

1. $\vdash (\sim w) \mathcal{U} w \supset \Diamond w$ — by A10
2. $\vdash \Diamond w \supset [w \vee \bigcirc \Diamond w]$ — by T21 and PR
3. $\vdash \Diamond w \supset [w \vee (\sim w \wedge \bigcirc \Diamond w)]$ — by PR
4. $\vdash \Diamond w \supset \Diamond w$ — by PT
5. $\vdash \Diamond w \supset (\sim w) \mathcal{U} w$ — by 3, 4 and R$\mathcal{U}$I
6. $\vdash (\sim w) \mathcal{U} w \equiv \Diamond w$ — by 1, 5 and PR ▟

T24. $\vdash (\Box w_1 \wedge \Diamond w_2) \supset (w_1 \mathcal{U} w_2)$

Proof:

1. $\vdash [\Box w_1 \wedge \Diamond w_2] \supset \Diamond w_2$ — by PR

2. $\vdash [\Box w_1 \wedge \Diamond w_2] \supset [(w_1 \wedge \bigcirc\Box w_1) \wedge (w_2 \vee \bigcirc\Diamond w_2)]$ by PR, T20 and T21
3. $\vdash (\Box w_1 \wedge \Diamond w_2) \supset [w_2 \vee (w_1 \wedge \bigcirc\Box w_1 \wedge \bigcirc\Diamond w_2)]$ by PR
4. $\vdash (\Box w_1 \wedge \Diamond w_2) \supset [w_2 \vee (w_1 \wedge \bigcirc(\Box w_1 \wedge \Diamond w_2))]$ by T12 and PR
5. $\vdash [\Box w_1 \wedge \Diamond w_2] \supset w_1 \mathcal{U} w_2$ by 1, 4 and R$\mathcal{U}$I,

taking w to be $\Box w_1 \wedge \Diamond w_2$, u to be w_1, and v to be w_2 ⌟

T25. $\vdash (w_1 \mathcal{U} w_2)\mathcal{U} w_2 \equiv w_1 \mathcal{U} w_2$

Proof:

1. $\vdash (w_1 \mathcal{U} w_2)\mathcal{U} w_2 \supset [w_2 \vee w_1 \mathcal{U} w_2]$ by A9 and PR
2. $\vdash w_2 \supset w_1 \mathcal{U} w_2$ by A9 and PR
3. $\vdash (w_1 \mathcal{U} w_2)\mathcal{U} w_2 \supset w_1 \mathcal{U} w_2$ by 1, 2 and PR
4. $\vdash w_1 \mathcal{U} w_2 \supset \Diamond w_2$ by A10
5. $\vdash w_1 \mathcal{U} w_2 \supset [w_2 \vee (w_1 \wedge \bigcirc(w_1 \mathcal{U} w_2))]$ by A9 and PR
6. $\vdash w_1 \mathcal{U} w_2 \supset [w_2 \vee (w_1 \mathcal{U} w_2 \wedge \bigcirc(w_1 \mathcal{U} w_2))]$ by PR
7. $\vdash w_1 \mathcal{U} w_2 \supset (w_1 \mathcal{U} w_2)\mathcal{U} w_2$ by 4, 6 and R$\mathcal{U}$I
8. $\vdash (w_1 \mathcal{U} w_2)\mathcal{U} w_2 \equiv w_1 \mathcal{U} w_2$ by 3, 7 and PR ⌟

T26. $\vdash w_1 \mathcal{U} w_2 \equiv w_1 \mathcal{U}(w_1 \mathcal{U} w_2)$

Proof:

1. $\vdash w_2 \supset w_1 \mathcal{U} w_2$ by A9 and PR
2. $\vdash w_1 \mathcal{U} w_2 \supset w_1 \mathcal{U}(w_1 \mathcal{U} w_2)$ by $\mathcal{U}\mathcal{U}$
3. $\vdash w_1 \mathcal{U}(w_1 \mathcal{U} w_2) \supset [w_1 \mathcal{U} w_2 \vee [w_1 \wedge \bigcirc(w_1 \mathcal{U}(w_1 \mathcal{U} w_2))]]$ by A9 and PR
4. $\vdash w_1 \mathcal{U}(w_1 \mathcal{U} w_2) \supset \{w_2 \vee [w_1 \wedge \bigcirc(w_1 \mathcal{U} w_2)] \vee [w_1 \wedge \bigcirc(w_1 \mathcal{U}(w_1 \mathcal{U} w_2))]\}$ by A9 and PR
5. $\vdash w_1 \mathcal{U}(w_1 \mathcal{U} w_2) \supset \{w_2 \vee [w_1 \wedge \bigcirc(w_1 \mathcal{U} w_2 \vee w_1 \mathcal{U}(w_1 \mathcal{U} w_2))]\}$ by T13 and PR
6. $\vdash [w_1 \mathcal{U} w_2 \vee w_1 \mathcal{U}(w_1 \mathcal{U} w_2)] \supset w_1 \mathcal{U}(w_1 \mathcal{U} w_2)$ by 2 and PR

7. $\vdash w_1 \mathcal{U} (w_1 \mathcal{U} w_2) \supset \{w_2 \lor [w_1 \land \bigcirc(w_1 \mathcal{U} (w_1 \mathcal{U} w_2))]\}$ by 6 with $\bigcirc\bigcirc$, 5, and PR

8. $\vdash w_1 \mathcal{U} (w_1 \mathcal{U} w_2) \supset \Diamond(w_1 \mathcal{U} w_2)$ by A10

9. $\vdash w_1 \mathcal{U} w_2 \supset \Diamond w_2$ by A10

10. $\vdash \Diamond(w_1 \mathcal{U} w_2) \supset \Diamond\Diamond w_2$ by $\Diamond\Diamond$

11. $\vdash w_1 \mathcal{U} (w_1 \mathcal{U} w_2) \supset \Diamond w_2$ by 8, 10, T4 and PR

12. $\vdash w_1 \mathcal{U} (w_1 \mathcal{U} w_2) \supset w_1 \mathcal{U} w_2$ by 11, 7 and R$\mathcal{U}$I, taking w to be $w_1 \mathcal{U} (w_1 \mathcal{U} w_2)$, u to be w_1, and v to be w_2

15. $\vdash w_1 \mathcal{U} w_2 \equiv w_1 \mathcal{U} (w_1 \mathcal{U} w_2)$ by 2, 12 and PR ∎

$\mathcal{U}$ *Insertion* — $\mathcal{U}$I

(a) $$\frac{\vdash v}{\vdash u \mathcal{U} v}$$ for an arbitrary u

(b) $$\frac{\vdash u, \ \vdash \Diamond v}{\vdash u \mathcal{U} v}$$

Proof:

(a)

1. $\vdash v$ given
2. $\vdash v \supset u \mathcal{U} v$ by A9 and PR
3. $\vdash u \mathcal{U} v$ by 1, 2 and PR

(b)

1. $\vdash u$ given

2 $\vdash \Diamond v$ given

3. $\vdash \Box u$ by 1 and $\Box$I

4. $\vdash (\Box u \land \Diamond v) \supset u \mathcal{U} v$ by T24

5. $\vdash u \mathcal{U} v$ by 2, 3, 4 and PR ∎

$\mathcal{U}$ *Concatenation* — $\mathcal{U}$C

$$\frac{\begin{array}{c}\vdash v_1 \supset u \mathcal{U} v_2 \\ \vdash v_2 \supset u \mathcal{U} v_3\end{array}}{\vdash v_1 \supset u \mathcal{U} v_3}$$

Proof:

1. $\vdash v_1 \supset u\mathcal{U}v_2$ given
2. $\vdash v_2 \supset u\mathcal{U}v_3$ given
3. $\vdash u\mathcal{U}v_2 \supset u\mathcal{U}(u\mathcal{U}v_3)$ by $\mathcal{U}\mathcal{U}$
4. $\vdash v_1 \supset u\mathcal{U}(u\mathcal{U}v_3)$ by 1, 3 and PR
5. $\vdash v_1 \supset u\mathcal{U}v_3$ by T26 and PR ∎

T27. $\vdash [\Box w_1 \wedge w_2\mathcal{U}w_3] \supset (w_1 \wedge w_2)\mathcal{U}(w_1 \wedge w_3)$

Proof:

1. $\vdash w_2\mathcal{U}w_3 \supset \Diamond w_3$ by A10
2. $\vdash [\Box w_1 \wedge w_2\mathcal{U}w_3] \supset (\Box w_1 \wedge \Diamond w_3)$ by PR
3. $\vdash [\Box w_1 \wedge w_2\mathcal{U}w_3] \supset \Diamond(w_1 \wedge w_3)$ by T11 and PR
4. $\vdash w_2\mathcal{U}w_3 \supset [w_3 \vee (w_2 \wedge \bigcirc(w_2\mathcal{U}w_3))]$ by A9 and PR
5. $\vdash [\Box w_1 \wedge w_2\mathcal{U}w_3] \supset [(\Box w_1 \wedge w_3) \vee (\Box w_1 \wedge w_2 \wedge \bigcirc(w_2\mathcal{U}w_3))]$ by PR
6. $\vdash (\Box w_1 \wedge w_3) \supset (w_1 \wedge w_3)$ by A3 and PR
7. $\vdash [\Box w_1 \wedge w_2 \wedge \bigcirc(w_2\mathcal{U}w_3)] \supset [w_1 \wedge w_2 \wedge \bigcirc\Box w_1 \wedge \bigcirc(w_2\mathcal{U}w_3)]$ by T20 and PR
8. $\vdash [\Box w_1 \wedge w_2 \wedge \bigcirc(w_2\mathcal{U}w_3)] \supset [(w_1 \wedge w_2) \wedge \bigcirc(\Box w_1 \wedge w_2\mathcal{U}w_3)]$ by T12 and PR
9. $\vdash [\Box w_1 \wedge w_2\mathcal{U}w_3] \supset \{(w_1 \wedge w_3) \vee [(w_1 \wedge w_2) \wedge \bigcirc(\Box w_1 \wedge w_2\mathcal{U}w_3)]\}$ by 5, 6, 8 and PR
10. $\vdash [\Box w_1 \wedge w_2\mathcal{U}w_3] \supset (w_1 \wedge w_2)\mathcal{U}(w_1 \wedge w_3)$ by 3, 9 and R$\mathcal{U}$I ∎

The next theorem displays the commutation relation between the $\bigcirc$ and the $\mathcal{U}$ operators.

T28. $\vdash (\bigcirc w_1)\mathcal{U}(\bigcirc w_2) \equiv \bigcirc(w_1\mathcal{U}w_2)$

Proof:

1. $\vdash w_1\mathcal{U}w_2 \equiv [w_2 \vee (w_1 \wedge \bigcirc(w_1\mathcal{U}w_2))]$ by A9

2. $\vdash \bigcirc(w_1 \mathcal{U} w_2) \equiv [\bigcirc w_2 \lor (\bigcirc w_1 \land \bigcirc\bigcirc(w_1 \mathcal{U} w_2))]$
   by T12, T13, $\bigcirc\bigcirc$ and PR
3. $\vdash [\bigcirc w_2 \lor (\bigcirc w_1 \land \bigcirc\bigcirc(w_1 \mathcal{U} w_2))] \supset \bigcirc(w_1 \mathcal{U} w_2)$ by PR
4. $\vdash (\bigcirc w_1)\mathcal{U}(\bigcirc w_2) \supset \bigcirc(w_1 \mathcal{U} w_2)$ by L$\mathcal{U}$I, taking w to be $w_1 \mathcal{U} w_2$

5. $\vdash w_1 \mathcal{U} w_2 \supset \Diamond w_2$ by A10
6. $\vdash \bigcirc(w_1 \mathcal{U} w_2) \supset \bigcirc\Diamond w_2$ by $\bigcirc\bigcirc$
7. $\vdash \bigcirc(w_1 \mathcal{U} w_2) \supset \Diamond\bigcirc w_2$ by T17 and PR
8. $\vdash \bigcirc(w_1 \mathcal{U} w_2) \supset \{\bigcirc w_2 \lor [\bigcirc w_1 \land \bigcirc\bigcirc(w_1 \mathcal{U} w_2)]\}$ by 2 and PR
9. $\vdash \bigcirc(w_1 \mathcal{U} w_2) \supset (\bigcirc w_1)\mathcal{U}(\bigcirc w_2)$ by 7, 8 and R$\mathcal{U}$I,
   taking w to be $\bigcirc(w_1 \mathcal{U} w_2)$, u to be $\bigcirc w_1$, and v to be $\bigcirc w_2$

10. $\vdash (\bigcirc w_1)\mathcal{U}(\bigcirc w_2) \equiv \bigcirc(w_1 \mathcal{U} w_2)$ by 4, 9 and PR $\lrcorner$

Having classified $\Box$ as a universal operator, $\Diamond$ as an existential operator and $\bigcirc$ as being both universal and existential, we observe that $\mathcal{U}$ is universal with respect to its first argument and existential with respect to its second argument. This yields the commutation properties listed in T29 and T30.

T29. $\vdash (w_1 \land w_2)\mathcal{U} w_3 \equiv [w_1 \mathcal{U} w_3 \land w_2 \mathcal{U} w_3]$

Proof:

1. $\vdash (w_1 \land w_2) \supset w_1$ by PT
2. $\vdash (w_1 \land w_2)\mathcal{U} w_3 \supset w_1 \mathcal{U} w_3$ by $\mathcal{U}\mathcal{U}$
3. $\vdash (w_1 \land w_2)\mathcal{U} w_3 \supset w_2 \mathcal{U} w_3$ similarly
4. $\vdash (w_1 \land w_2)\mathcal{U} w_3 \supset [w_1 \mathcal{U} w_3 \land w_2 \mathcal{U} w_3]$ by 2, 3 and PR

5. $\vdash w_1 \mathcal{U} w_3 \supset \Diamond w_3$ by A10
6. $\vdash [w_1 \mathcal{U} w_3 \land w_2 \mathcal{U} w_3] \supset \Diamond w_3$ by PR
7. $\vdash w_1 \mathcal{U} w_3 \supset \{w_3 \lor [w_1 \land \bigcirc(w_1 \mathcal{U} w_3)]\}$ by A9 and PR
8. $\vdash w_2 \mathcal{U} w_3 \supset \{w_3 \lor [w_2 \land \bigcirc(w_2 \mathcal{U} w_3)]\}$ by A9 and PR
9. $\vdash [w_1 \mathcal{U} w_3 \land w_2 \mathcal{U} w_3] \supset \{w_3 \lor [(w_1 \land w_2) \land \bigcirc(w_1 \mathcal{U} w_3 \land w_2 \mathcal{U} w_3)]\}$
   by 7, 8, T12 and PR
10. $\vdash [w_1 \mathcal{U} w_3 \land w_2 \mathcal{U} w_3] \supset (w_1 \land w_2)\mathcal{U} w_3$ by 6, 9 and R$\mathcal{U}$I,
    taking w to be $(w_1 \mathcal{U} w_3) \land (w_2 \mathcal{U} w_3)$, u to be $w_1 \land w_2$, and v to be w_3

11. $\vdash (w_1 \wedge w_2)\mathcal{U}w_3 \equiv [w_1\mathcal{U}w_3 \;\wedge\; w_2\mathcal{U}w_3]$ by 4, 10 and PR ▟

T30. $\vdash w_1\mathcal{U}(w_2 \vee w_3) \equiv [w_1\mathcal{U}w_2 \;\vee\; w_1\mathcal{U}w_3]$

Proof:

1. $\vdash w_2 \supset (w_2 \vee w_3)$ by PT
2. $\vdash w_1\mathcal{U}w_2 \supset w_1\mathcal{U}(w_2 \vee w_3)$ by $\mathcal{U}\mathcal{U}$
3. $\vdash w_1\mathcal{U}w_3 \supset w_1\mathcal{U}(w_2 \vee w_3)$ similarly
4. $\vdash [w_1\mathcal{U}w_2 \vee w_1\mathcal{U}w_3] \supset w_1\mathcal{U}(w_2 \vee w_3)$ by 2, 3 and PR
5. $\vdash w_1\mathcal{U}(w_2 \vee w_3) \supset \{(w_2 \vee w_3) \vee [w_1 \wedge \bigcirc(w_1\mathcal{U}(w_2 \vee w_3))]\}$ by A9 and PR
6. $\vdash [w_2 \vee (w_1 \wedge \bigcirc(w_1\mathcal{U}w_2))] \supset w_1\mathcal{U}w_2$ by A9 and PR
7. $\vdash \sim(w_1\mathcal{U}w_2) \supset \{\sim w_2 \wedge [\sim w_1 \vee \bigcirc\sim(w_1\mathcal{U}w_2)]\}$ by A4 and PR
8. $\vdash \sim(w_1\mathcal{U}w_3) \supset \{\sim w_3 \wedge [\sim w_1 \vee \bigcirc\sim(w_1\mathcal{U}w_3)]\}$ similarly
9. $\vdash [w_1\mathcal{U}(w_2 \vee w_3) \wedge \sim(w_1\mathcal{U}w_2) \wedge \sim(w_1\mathcal{U}w_3)] \supset$
   $[\sim w_2 \wedge \sim w_3 \wedge w_1 \wedge \bigcirc(w_1\mathcal{U}(w_2 \vee w_3)) \wedge \bigcirc\sim(w_1\mathcal{U}w_2) \wedge \bigcirc\sim(w_1\mathcal{U}w_3)]$
   by 5, 7, 8 and PR
10. $\vdash [w_1\mathcal{U}(w_2 \vee w_3) \wedge \sim(w_1\mathcal{U}w_2) \wedge \sim(w_1\mathcal{U}w_3)] \supset$
    $\{\sim(w_2 \vee w_3) \wedge \bigcirc[w_1\mathcal{U}(w_2 \vee w_3) \wedge \sim(w_1\mathcal{U}w_2) \wedge \sim(w_1\mathcal{U}w_3)]\}$
    by T12 and PR
11. $\vdash [w_1\mathcal{U}(w_2 \vee w_3) \wedge \sim(w_1\mathcal{U}w_2) \wedge \sim(w_1\mathcal{U}w_3)] \supset \Box\sim(w_2 \vee w_3)$ by DCI
12. $\vdash w_1\mathcal{U}(w_2 \vee w_3) \supset \Diamond(w_2 \vee w_3)$ by A10
13. $\vdash w_1\mathcal{U}(w_2 \vee w_3) \supset \sim[\sim(w_1\mathcal{U}w_2) \wedge \sim(w_1\mathcal{U}w_3)]$ by 11, 12, A1 and PR
14. $\vdash w_1\mathcal{U}(w_2 \vee w_3) \supset [w_1\mathcal{U}w_2 \vee w_1\mathcal{U}w_3]$ by PR
15. $\vdash w_1\mathcal{U}(w_2 \vee w_3) \equiv [w_1\mathcal{U}w_2 \vee w_1\mathcal{U}w_3]$ by 4, 14 and PR ▟

T31. $\vdash [\Diamond w_1 \vee \Diamond w_2] \supset [(\sim w_1)\mathcal{U}w_2 \vee (\sim w_2)\mathcal{U}w_1]$

Proof:

1. $\vdash [\Diamond w_1 \vee \Diamond w_2] \supset \Diamond(w_1 \vee w_2)$ by T8 and PR

2. $\vdash \Diamond(w_1 \vee w_2) \supset (\sim(w_1 \vee w_2))\mathfrak{U}(w_1 \vee w_2)$ by T23 and PR
3. $\vdash \Diamond(w_1 \vee w_2) \supset (\sim w_1 \wedge \sim w_2)\mathfrak{U}(w_1 \vee w_2)$ by $\mathfrak{U}\mathfrak{U}$ and PR
4. $\vdash \Diamond(w_1 \vee w_2) \supset [(\sim w_1 \wedge \sim w_2)\mathfrak{U}w_1 \vee (\sim w_1 \wedge \sim w_2)\mathfrak{U}w_2]$ by T30 and PR
5. $\vdash (\sim w_1 \wedge \sim w_2)\mathfrak{U}w_1 \supset (\sim w_2)\mathfrak{U}w_1$ by $\mathfrak{U}\mathfrak{U}$ and PR
6. $\vdash (\sim w_1 \wedge \sim w_2)\mathfrak{U}w_2 \supset (\sim w_1)\mathfrak{U}w_2$ by $\mathfrak{U}\mathfrak{U}$ and PR
7. $\vdash \Diamond(w_1 \vee w_2) \supset [(\sim w_1)\mathfrak{U}w_2 \vee (\sim w_2)\mathfrak{U}w_1]$ by 4, 5, 6 and PR
8. $\vdash (\Diamond w_1 \vee \Diamond w_2) \supset [(\sim w_1)\mathfrak{U}w_2 \vee (\sim w_2)\mathfrak{U}w_1]$ by 1, 7 and PR ∎

The following two theorems display the one way implication resulting from the interchange of the $\mathfrak{U}$ with a boolean operator of the opposite character.

T32. $\vdash w_1\mathfrak{U}(w_2 \wedge w_3) \supset [w_1\mathfrak{U}w_2 \wedge w_1\mathfrak{U}w_3]$

Proof:

1. $\vdash (w_2 \wedge w_3) \supset w_2$ by PT
2. $\vdash w_1\mathfrak{U}(w_2 \wedge w_3) \supset w_1\mathfrak{U}w_2$ by $\mathfrak{U}\mathfrak{U}$ and PR
3. $\vdash w_1\mathfrak{U}(w_2 \wedge w_3) \supset w_1\mathfrak{U}w_3$ similarly
4. $\vdash w_1\mathfrak{U}(w_2 \wedge w_3) \supset [w_1\mathfrak{U}w_2 \wedge w_1\mathfrak{U}w_3]$ by 2, 3 and PR ∎

T33. $\vdash [w_1\mathfrak{U}w_3 \vee w_2\mathfrak{U}w_3] \supset (w_1 \vee w_2)\mathfrak{U}w_3$

Proof:

1. $\vdash w_1 \supset (w_1 \vee w_2)$ by PT
2. $\vdash w_1\mathfrak{U}w_3 \supset (w_1 \vee w_2)\mathfrak{U}_3$ by $\mathfrak{U}\mathfrak{U}$
3. $\vdash w_2 \supset (w_1 \vee w_2)$ by PT
4. $\vdash w_2\mathfrak{U}w_3 \supset (w_1 \vee w_2)\mathfrak{U}_3$ by $\mathfrak{U}\mathfrak{U}$
5. $\vdash [w_1\mathfrak{U}w_3 \vee w_2\mathfrak{U}w_3] \supset (w_1 \vee w_2)\mathfrak{U}w_3$ by 2, 4 and PR ∎

T34. $\vdash (w_1 \supset w_2)\mathfrak{U}w_3 \supset [w_1\mathfrak{U}w_3 \supset w_2\mathfrak{U}w_3]$

Proof:

1. $\vdash (w_1 \supset w_2)\mathfrak{U}w_3 \;\supset\; \Diamond w_3$ by A10

2. $\vdash [(w_1 \supset w_2)\mathfrak{U}w_3 \;\wedge\; w_1\mathfrak{U}w_3] \;\supset$
$\{w_3 \vee [(w_1 \supset w_2) \wedge \bigcirc((w_1 \supset w_2)\mathfrak{U}w_3) \wedge w_1 \wedge \bigcirc(w_1\mathfrak{U}w_3)]\}$ by A9 and PR

3. $\vdash [(w_1 \supset w_2)\mathfrak{U}w_3 \;\wedge\; w_1\mathfrak{U}w_3] \;\supset$
$\{w_3 \vee [w_2 \wedge \bigcirc((w_1 \supset w_2)\mathfrak{U}w_3) \wedge \bigcirc(w_1\mathfrak{U}w_3)]\}$ by PR

4. $\vdash [(w_1 \supset w_2)\mathfrak{U}w_3 \;\wedge\; w_1\mathfrak{U}w_3] \;\supset$
$\{w_3 \vee [w_2 \wedge \bigcirc((w_1 \supset w_2)\mathfrak{U}w_3 \wedge w_1\mathfrak{U}w_3)]\}$ by T12 and PR

5. $\vdash [(w_1 \supset w_2)\mathfrak{U}w_3 \;\wedge\; w_1\mathfrak{U}w_3] \;\supset\; w_2\mathfrak{U}w_3$ by 1, 4 and R$\mathfrak{U}$I,
taking w to be $((w_1 \supset w_2)\mathfrak{U}w_3) \wedge (w_1\mathfrak{U}w_3)$, u to be w_2, and v to be w_3

6. $\vdash (w_1 \supset w_2)\mathfrak{U}w_3 \;\supset\; [w_1\mathfrak{U}w_3 \;\supset\; w_2\mathfrak{U}w_3]$ by PR ■

T35. $\vdash [w_1\mathfrak{U}w_2 \;\wedge\; (\sim w_2)\mathfrak{U}w_3] \;\supset\; w_1\mathfrak{U}w_3$

Proof:

1. $\vdash (\sim w_2)\mathfrak{U}w_3 \;\supset\; \Diamond w_3$ by A10

2. $\vdash [w_1\mathfrak{U}w_2 \;\wedge\; (\sim w_2)\mathfrak{U}w_3] \;\supset\; \Diamond w_3$ by PR

3. $\vdash w_1\mathfrak{U}w_2 \;\supset\; \{w_2 \vee [w_1 \wedge \bigcirc(w_1\mathfrak{U}w_2)]\}$ by A9 and PR

4. $\vdash (\sim w_2)\mathfrak{U}w_3 \;\supset\; \{w_3 \vee [\sim w_2 \wedge \bigcirc((\sim w_2)\mathfrak{U}w_3)]\}$ by A9 and PR

5. $\vdash [w_1\mathfrak{U}w_2 \;\wedge\; (\sim w_2)\mathfrak{U}w_3] \;\supset$
$\{w_3 \vee [w_1 \wedge \sim w_2 \wedge \bigcirc(w_1\mathfrak{U}w_2) \wedge \bigcirc((\sim w_2)\mathfrak{U}w_3)]\}$ by 3, 4 and PR

6. $\vdash [w_1\mathfrak{U}w_2 \;\wedge\; (\sim w_2)\mathfrak{U}w_3] \;\supset$
$\{w_3 \vee [w_1 \wedge \bigcirc(w_1\mathfrak{U}w_2 \wedge (\sim w_2)\mathfrak{U}w_3)]\}$ by T12 and PR

7. $\vdash [w_1\mathfrak{U}w_2 \;\wedge\; (\sim w_2)\mathfrak{U}w_3] \;\supset\; w_1\mathfrak{U}w_3$ by 2, 6 and R$\mathfrak{U}$I ■

T36. $\vdash w_1\mathfrak{U}(w_2 \wedge w_3) \;\supset\; (w_1\mathfrak{U}w_2)\mathfrak{U}w_3$

Proof:

1. $\vdash w_1\mathfrak{U}(w_2 \wedge w_3) \;\supset\; \Diamond(w_2 \wedge w_3)$ by A10

2. $\vdash (w_2 \wedge w_3) \supset w_3$ by PT

3. $\vdash \Diamond(w_2 \wedge w_3) \supset \Diamond w_3$ by $\Diamond\Diamond$

4. $\vdash w_1 \mathcal{U} (w_2 \wedge w_3) \supset \Diamond w_3$ by 1, 3 and PR

5. $\vdash w_1 \mathcal{U} (w_2 \wedge w_3) \supset \{(w_2 \wedge w_3) \vee [w_1 \wedge \bigcirc(w_1 \mathcal{U} (w_2 \wedge w_3))]\}$ by A9 and PR

6. $\vdash (w_2 \wedge w_3) \supset w_2$ by PT

7. $\vdash w_1 \mathcal{U} (w_2 \wedge w_3) \supset w_1 \mathcal{U} w_2$ by $\mathcal{U}\mathcal{U}$

8. $\vdash w_1 \mathcal{U} (w_2 \wedge w_3) \supset \{w_3 \vee [w_1 \mathcal{U} w_2 \wedge \bigcirc(w_1 \mathcal{U} (w_2 \wedge w_3))]\}$ by 5, 7 and PR

9. $\vdash w_1 \mathcal{U} (w_2 \wedge w_3) \supset (w_1 \mathcal{U} w_2) \mathcal{U} w_3$ by 4, 8 and R$\mathcal{U}$I $\lrcorner$

The following two theorems are referred to as "collapsing" theorems, since they may be used to derive a consequence of smaller nesting depth from a nested until expression.

T37. $\vdash (w_1 \mathcal{U} w_2) \mathcal{U} w_3 \supset (w_1 \vee w_2) \mathcal{U} w_3$

Proof:

1. $\vdash w_1 \mathcal{U} w_2 \supset [w_2 \vee (w_1 \wedge \bigcirc(w_1 \mathcal{U} w_2))]$ by A9 and PR

2. $\vdash w_1 \mathcal{U} w_2 \supset (w_1 \vee w_2)$ by PR

3. $\vdash (w_1 \mathcal{U} w_2) \mathcal{U} w_3 \supset (w_1 \vee w_2) \mathcal{U} w_3$ by $\mathcal{U}\mathcal{U}$ $\lrcorner$

T38. $\vdash w_1 \mathcal{U} (w_2 \mathcal{U} w_3) \supset (w_1 \vee w_2) \mathcal{U} w_3$

Proof:

1. $\vdash w_1 \mathcal{U} (w_2 \mathcal{U} w_3) \supset \Diamond(w_2 \mathcal{U} w_3)$ by A10

2. $\vdash w_2 \mathcal{U} w_3 \supset \Diamond w_3$ by A10

3. $\vdash w_1 \mathcal{U} (w_2 \mathcal{U} w_3) \supset \Diamond w_3$ by 1, 2 and $\Diamond$C

4. $\vdash w_1 \mathcal{U} (w_2 \mathcal{U} w_3) \supset \{w_2 \mathcal{U} w_3 \vee [w_1 \wedge \bigcirc(w_1 \mathcal{U} (w_2 \mathcal{U} w_3))]\}$ by A9 and PR

5. $\vdash w_1 \mathcal{U} (w_2 \mathcal{U} w_3) \supset \{w_3 \vee [w_2 \wedge \bigcirc(w_2 \mathcal{U} w_3)] \vee [w_1 \wedge \bigcirc(w_1 \mathcal{U} (w_2 \mathcal{U} w_3))]\}$
by A9 and PR

6. $\vdash w_2 \mathcal{U} w_3 \supset w_1 \mathcal{U} (w_2 \mathcal{U} w_3)$ by A9 and PR

7. $\vdash [w_2 \wedge \bigcirc(w_2 \mathcal{U} w_3)] \supset [(w_1 \vee w_2) \wedge \bigcirc(w_1 \mathcal{U} (w_2 \mathcal{U} w_3))]$ by $\bigcirc\bigcirc$ and PR

8. $\vdash [w_1 \land \bigcirc(w_1 \mathcal{U}(w_2 \mathcal{U} w_3))] \supset [(w_1 \lor w_2) \land \bigcirc(w_1 \mathcal{U}(w_2 \mathcal{U} w_3))]$ by PR

9. $\vdash w_1 \mathcal{U}(w_2 \mathcal{U} w_3) \supset \{w_3 \lor [(w_1 \lor w_2) \land \bigcirc(w_1 \mathcal{U}(w_2 \mathcal{U} w_3))]\}$ by 5, 7, 8 and PR

10. $\vdash w_1 \mathcal{U}(w_2 \mathcal{U} w_3) \supset (w_1 \lor w_2) \mathcal{U} w_3$ by 3, 9, and R$\mathcal{U}$I ∎

A very useful derived operator is the *unless* operator $u \mathfrak{U} v$ being defined by

$$u \mathfrak{U} v \equiv [\Box u \lor (u \mathcal{U} v)].$$

The unless operator does not insist on the fact that v actually happens but it requires that u holds until such an occurrence. If v never happens u must hold forever. This operator is related to the binary "as long as" operator $p \Box q$, reading "q as long as p," introduced by Lamport in [L2]. The meaning of this construct is that q holds continuously as long as p is continuously maintained. We may express $p \Box q$ by:

$$p \Box q \equiv q \mathfrak{U}(\sim p).$$

Following is a rule for establishing the unless operator.

Unless Introduction — $\mathfrak{U}$I

$$\frac{\vdash u \supset \bigcirc(u \lor v)}{\vdash u \supset (u \mathfrak{U} v)}$$

Proof:

1. $\vdash u \supset \bigcirc(u \lor v)$ given
2. $\vdash u \supset [\bigcirc u \lor \bigcirc v]$ by T13
3. $\vdash \sim(u \mathcal{U} v) \supset \{\sim v \land [\sim u \lor \bigcirc \sim(u \mathcal{U} v)]\}$ by A9, T4 and PR
4. $\vdash \bigcirc \sim(u \mathcal{U} v) \supset \bigcirc \sim v$ by ○○ and PR
5. $\vdash [u \land \sim(u \mathcal{U} v)] \supset [u \land \bigcirc \sim(u \mathcal{U} v)]$ by 3 and PR
6. $\vdash [u \land \sim(u \mathcal{U} v)] \supset [u \land \bigcirc \sim(u \mathcal{U} v) \land \sim \bigcirc v]$ by 4, 5, A4 and PR
7. $\vdash [u \land \sim(u \mathcal{U} v)] \supset [u \land \bigcirc u \land \bigcirc \sim(u \mathcal{U} v)]$ by 2, 6 and PR
8. $\vdash [u \land \sim(u \mathcal{U} v)] \supset [u \land \bigcirc(u \land \sim(u \mathcal{U} v))]$ by T7 and PR

9. $\vdash [u \land \sim(u \mathcal{U} v)] \supset \Box u$ by DCI
10. $\vdash u \supset (\Box u \lor (u \mathcal{U} v))$ by PR

11. $\vdash u \supset (u \,\mathcal{U}\, v)$ by definition of $\mathcal{U}$ ∎

This concludes the description of the propositional section of general temporal logic. The axiomatic system presented for this section of the logic is known to be complete, and the validity problem decidable ([PS]). Consequently, there exists a procedure that tests each formula in PTL (Propositional Temporal Logic) for validity, and constructs a proof in the presented system if the statement is valid. The procedure given in [PS] takes exponential time in the size of the tested formula.

4. QUANTIFIERS

Since we intend to use terms and predicates in our reasoning we have to extend our system to admit individual variables, terms and quantification. Let us consider additional axioms involving quantifiers and their interaction with the temporal operators.

AXIOMS:

A11. $\vdash \sim\exists x.w \equiv \forall x. \sim w$

A12. $\vdash (\forall x.w(x)) \supset w(t)$
where t is any term globally free for x in w

A13. $\vdash (\forall x. \bigcirc w) \supset (\bigcirc \forall x.w)$

In these axioms, x is any global individual variable. Axioms A11 and A12 are the usual predicate calculus axioms: A11 defines $\exists$ as the dual of $\forall$ and A12 is the *instantiation axiom*. Axiom A13 is the Barcan formula for the $\bigcirc$ operator; it states that since both operators $\forall$ and $\bigcirc$ have universal characteristics they commute. We use the substitution notation $w(x)$ replaced by $w(t)$ to denote the substitution of the term t for all free occurrences of x in w.

A term t is said to be ***globally free for x in w*** if substitution of t for all free occurrences of x in w: (a) does not create new bound occurrences of (global) variables, and (b) does not create new occurrences of local variables in the scope of a temporal operator. A trivial case: if t is x itself, then t is free for x. Condition (a) is the one stipulated in classical predicate logic. Condition (b) is special to modal and temporal logics with quantification. Condition (b) is essential for A12, because without it we could derive the formula

$$(\forall x. \Diamond(x < y)) \supset \Diamond(y < y),$$

which is not valid for a local variable y.

An additional rule of inference is:

INFERENCE RULE:

R4. $\forall$ Insertion — $\forall I$

$$\frac{\vdash u \supset v}{\vdash u \supset \forall x.v}$$

where x is not free in u.

DERIVED RULES AND THEOREMS:

From R4 we can obtain the derived rule

Instantiation Rule — INST

$$\frac{\vdash w(x)}{\vdash w(t)}$$

where t is any term globally free for x in w.

Proof:

1. $\vdash w(x)$ given
2. $\vdash \forall x.w(x)$ by $\forall$I (taking u to be *true*)
3. $\vdash (\forall x.w(x)) \supset w(t)$ by A12
4. $\vdash w(t)$ by 2, 3 and MP ∎

The following are the duals of A12 and R4 for the existential quantifier $\exists$:

T39. $\vdash w(t) \supset \exists x.w(x)$
where t is any term globally free for x in w.

Proof:

1. $\vdash (\forall x. \sim w(x)) \supset \sim w(t)$ by A12
2. $\vdash (\sim \exists x.w(x)) \supset \sim w(t)$ by A11 and PR
3. $\vdash w(t) \supset \exists x.w(x)$ by PR ∎

Note again that we need here the additional condition (b) ensuring that the substitution of t for x in w does not create new occurrences of local variables in the scope of a modal operator.

$\exists$ *Insertion* — $\exists$I

$$\frac{\vdash u \supset v}{\vdash \exists x.u \supset v}$$

where x is not free in v

Proof:

1. $\vdash u \supset v$ given
2. $\vdash \sim v \supset \sim u$ by PR
3. $\vdash \sim v \supset \forall x. \sim u$ by $\forall$I
4. $\vdash \sim v \supset \sim\exists x.u$ by A11 and PR
5. $\vdash \exists x.u \supset v$ by PR ∎

$\forall\forall$ *Rules*

(a) $$\frac{\vdash u \supset v}{\vdash \forall x.u \supset \forall x.v}$$ (b) $$\frac{\vdash u \equiv v}{\vdash \forall x.u \equiv \forall x.v}$$

Proof of (a):

1. $\vdash \forall x.u \supset u$ by A12
2. $\vdash u \supset v$ given
3. $\vdash \forall x.u \supset v$ by PR
4. $\vdash \forall x.u \supset \forall x.v$ by $\forall$I, since $\forall x.u$ contains no free occurrences of x.

Rule (b) then follows by propositional reasoning. ∎

$\exists\exists$ *Rules*

(a) $$\frac{\vdash u \supset v}{\vdash \exists x.u \supset \exists x.v}$$ (b) $$\frac{\vdash u \equiv v}{\vdash \exists x.u \equiv \exists x.v}$$

Proof of (a):

1. $\vdash u \supset v$ given
2. $\vdash (\sim v) \supset (\sim u)$ by PR
3. $\vdash (\forall x. \sim v) \supset (\forall x. \sim u)$ by $\forall\forall$
4. $\vdash (\sim\exists x.v) \supset (\sim\exists x.u)$ by A11 and PR

5. $\vdash \exists x.u \supset \exists x.v$ by PR

Rule (b) then follows by propositional reasoning. ■

From the axiom A1,

$$\vdash \sim\Diamond w \equiv \Box\sim w,$$

we can clearly deduce the formula

$$\vdash \sim(w \vee \Box\sim w) \equiv \sim(w \vee \sim\Diamond w)$$

by propositional reasoning (PR). However, we cannot deduce by PR the formula

$$\Box\Box\sim w \equiv \Box\sim\Diamond w$$

or

$$\forall x.\Box\sim w \equiv \forall x.\sim\Diamond w.$$

Here, the replacement of $\Box\sim w$ by $\sim\Diamond w$ is under the scope of the operator $\Box$ and the quantifier $\forall x$, respectively, and thus cannot be justified by propositional reasoning alone. For this reason we need the following equivalence rule.

> *Equivalence Rule* — ER
>
> Let w' be the result of replacing an occurrence of a subformula v_1 in w by v_2. Then
>
> $$\frac{\vdash v_1 \equiv v_2}{\vdash w \equiv w'}$$

Proof:

By induction on the structure of w.

Case: w is v_1. Then w' is v_2 and $\vdash v_1 \equiv v_2$ implies $\vdash w \equiv w'$.

Case: w is of the form $\sim u$. We assume that $\vdash v_1 \equiv v_2$ implies $\vdash u \equiv u'$. Then by propositional reasoning $\vdash \sim u \equiv \sim u'$, i.e., $\vdash w \equiv w'$.

Case: w is of the form $u_1 \vee u_2$. We assume that if $\vdash v_1 \equiv v_2$, then $\vdash u_1 \equiv u_1'$ and $\vdash u_2 \equiv u_2'$. Then by propositional reasoning $\vdash (u_1 \vee u_2) \equiv (u_1' \vee u_2')$, i.e., $\vdash w \equiv w'$.

The cases where w is of forms $u_1 \wedge u_2$, $u_1 \supset u_2$, etc. are similar.

Case: w is of the form $\Box u$. We assume that if $\vdash v_1 \equiv v_2$, then $\vdash u \equiv u'$. By the $\Box\Box$-rule, $\vdash \Box u \equiv \Box u'$, i.e., $\vdash w \equiv w'$.

The cases in which w is of forms $\Diamond u$, $\bigcirc u$, and $u_1 \mathcal{U} u_2$ are treated similarly, using the $\Diamond\Diamond$-rule, the $\bigcirc\bigcirc$-rule, and the $\mathcal{U}\mathcal{U}$-rule, respectively.

Case: w is of the form $\forall x.u$. We assume that if $\vdash v_1 \equiv v_2$, then $\vdash u \equiv u'$. Then by the $\forall\forall$-rule, $\vdash \forall x.u \equiv \forall x.u'$, i.e., $\vdash w \equiv w'$.

The case where w is of form $\exists x.u$ is proved similarly by the $\exists\exists$-rule. ∎

Deduction Rule — DED

$$\frac{w_1 \vdash w_2}{\vdash (\Box w_1) \supset w_2}$$

where the $\forall$I rule (Rule R4) is never applied to a free variable of w_1 in the derivation of $w_1 \vdash w_2$.

That is, if under the assumption w_1 we can derive $\vdash w_2$, where rule R4 is never applied to a free variable of w_1, then there exists a proof establishing $\vdash (\Box w_1) \supset w_2$. We clearly must also be careful in using any theorem or derived rule such as the $\forall\forall$ or ER rule that was established using the $\forall$I rule.

The additional $\Box$ operator in the conclusion is obviously necessary since in general $w_1 \vdash w_2$ does not imply $\vdash w_1 \supset w_2$. For example, obviously $w \vdash \Box w$ is true (an immediate application of rule R3: $\vdash w$ by assumption and therefore $\vdash \Box w$ by $\Box$I); but $w \supset \Box w$ is not a theorem.

Proof:

The proof of the temporal Deduction Rule follows the same arguments used in the proof of the classical deduction theorem of Predicate Calculus. By the given $w_1 \vdash w_2$, there exists a proof of the form:

$\vdash u_1$
$\vdash u_2$
.
.
.
$\vdash u_m$

such that $u_1 = w_1$ is the hypothesis on which the proof relies, and $u_m = w_2$ is the consequence of the proof. We replace each line $\vdash u_i$ in the proof of $w_1 \vdash w_2$ by the line $\vdash \Box w_1 \supset u_i$, and show that this transformation preserves soundness. That is

given	show
$\vdash u_1$	$\vdash (\Box w_1) \supset u_1$
$\vdash u_2$	$\vdash (\Box w_1) \supset u_2$
.	.
.	.
.	.

$\vdash u_i$	$\vdash (\Box w_1) \supset u_i$
.	.
.	.
.	.
$\vdash u_m$	$\vdash (\Box w_1) \supset u_m$
i.e., $\vdash w_2$	i.e.$\vdash (\Box w_1) \supset w_2$

where each u_i is either the assumption w_1, an axiom, or derived from previous u_j's by some rule of inference.

The proof is by a complete induction on i. We assume that for all $k < i$, $\vdash (\Box w_1) \supset u_k$, and prove that $\vdash (\Box w_1) \supset u_i$.

Case: u_i is an axiom.

1.	$\vdash u_i$	axiom
2.	$\vdash (\Box w_1) \supset u_i$	by PR

Note that $\vdash w'$ implies $\vdash w \supset w'$ for any w, by propositional reasoning.

Case: u_i is w_1.

1.	$\vdash (\Box w_1) \supset w_1$	by A3

Case: u_i is obtained by rule R1, i.e., u_i is an instance of a tautology.

1.	$\vdash u_i$	by PT
2.	$\vdash (\Box w_1) \supset u_i$	by PR

Case: u_i is obtained by rule R2 (using previous $\vdash u_k$ and $\vdash u_k \supset u_i$).

1.	$\vdash (\Box w_1) \supset u_k$	induction hypothesis
2.	$\vdash (\Box w_1) \supset (u_k \supset u_i)$	induction hypothesis
3.	$\vdash (\Box w_1) \supset u_i$	by 1, 2 and PR

Case: u_i is obtained by rule R3 (using previous $\vdash u_k$), i.e., u_i is $\Box u_k$.

1.	$\vdash (\Box w_1) \supset u_k$	induction hypothesis
2.	$\vdash (\Box\Box w_1) \supset \Box u_k$	by $\Box\Box$
3.	$\vdash (\Box w_1) \supset \Box\Box w_1$	by T3 and PR
4.	$\vdash (\Box w_1) \supset \Box u_k$	by 2, 3 and PR

Case: u_i is obtained by rule R4 (using previous $\vdash u \supset v$, i.e. u_k, to get $\vdash u \supset \forall x.v$, i.e. u_i, where x is not free in u).

By our deduction rule assumption, we know that x is also not free in w_1.

1.	$\vdash (\Box w_1) \supset (u \supset v)$	induction hypothesis
2.	$\vdash ((\Box w_1) \wedge u) \supset v$	by PR
3.	$\vdash ((\Box w_1) \wedge u) \supset \forall x.v$	by R4 (since x is not free in u or w_1)
4.	$\vdash (\Box w_1) \supset (u \supset \forall x.v)$	by PR ∎

A different approach to coping with the application of the □ insertion rule (rule R3) is to forbid it altogether. We then get the following restricted deduction rule:

Restricted Deduction Rule — RDED

$$\frac{w_1 \vdash w_2}{\vdash w_1 \supset w_2}$$

where □I (rule R3) is never applied and ∀I (rule R4) is never applied to a free variable of w_1 in the derivation of $w_1 \vdash w_2$.

Here, we are not allowed to use rule □I or any theorem or derived rule in whose proof □I was used.

The proof of RDED follows exactly that of DED except that the case in which rule R3 is applied does not arise.

QUANTIFIER THEOREMS:

T40. $\vdash (\sim\forall x.w) \equiv (\exists x. \sim w)$

Proof:

1.	$\vdash (\sim \sim w) \equiv w$	by PT
2.	$\vdash (\forall x. \sim \sim w) \equiv \forall x.w$	by ∀∀
3.	$\vdash (\sim\exists x. \sim w) \equiv \forall x.w$	by A11 and PR
4.	$\vdash \sim\forall x.w \equiv \exists x. \sim w$	by PR ∎

T41. $\vdash \forall x.(w_1 \wedge w_2) \equiv (\forall x.w_1 \wedge \forall x.w_2)$

Proof:

1. $\vdash \forall x.w_1 \supset w_1$ by A12
2. $\vdash \forall x.w_2 \supset w_2$ by A12
3. $\vdash (\forall x.w_1 \wedge \forall x.w_2) \supset (w_1 \wedge w_2)$ by 1, 2 and PR
4. $\vdash (\forall x.w_1 \wedge \forall x.w_2) \supset \forall x.(w_1 \wedge w_2)$ by $\forall$I

5. $\vdash (w_1 \wedge w_2) \supset w_1$ by PT
6. $\vdash \forall x.(w_1 \wedge w_2) \supset \forall x.w_1$ by $\forall\forall$
7. $\vdash (w_1 \wedge w_2) \supset w_2$ by PT
8. $\vdash \forall x.(w_1 \wedge w_2) \supset \forall x.w_2$ by $\forall\forall$
9. $\vdash \forall x.(w_1 \wedge w_2) \supset (\forall x.w_1 \wedge \forall x.w_2)$ by 6, 8 and PR

10. $\vdash \forall x.(w_1 \wedge w_2) \equiv (\forall x.w_1 \wedge \forall x.w_2)$ by 4, 9 and PR ∎

T42. $\vdash \exists x.(w_1 \vee w_2) \equiv (\exists x.w_1 \vee \exists x.w_2)$

Proof:

1. $\vdash \forall x.(\sim w_1 \wedge \sim w_2) \equiv (\forall x. \sim w_1 \wedge \forall x. \sim w_2)$ by T41
2. $\vdash \forall x. \sim (w_1 \vee w_2) \equiv (\forall x. \sim w_1 \wedge \forall x. \sim w_2)$ by ER
3. $\vdash \sim\exists x.(w_1 \vee w_2) \equiv (\sim\exists x.w_1 \wedge \sim\exists x.w_2)$ by A11 and PR
4. $\vdash \exists x.(w_1 \vee w_2) \equiv (\exists x.w_1 \vee \exists x.w_2)$ by PR ∎

T43. $\vdash \forall x.(w_1 \vee w_2) \equiv [w_1 \vee \forall x.w_2]$ where x is not free in w_1.

Proof:

1. $\vdash \forall x.(w_1 \vee w_2) \supset [w_1 \vee w_2]$ by A12
2. $\vdash [\forall x.(w_1 \vee w_2) \wedge \sim w_1] \supset w_2$ by PR

3. $\vdash [\forall x.(w_1 \lor w_2) \land \sim w_1] \supset \forall x.w_2$ by $\forall$I, since x is not free in $\forall x.(w_1 \lor w_2) \land \sim w_1$
4. $\vdash \forall x.(w_1 \lor w_2) \supset [w_1 \lor \forall x.w_2]$ by PR
5. $\vdash w_1 \supset [w_1 \lor w_2]$ by PT
6. $\vdash \forall x.w_2 \supset w_2$ by A12
7. $\vdash \forall x.w_2 \supset [w_1 \lor w_2]$ by PR
8. $\vdash [w_1 \lor \forall x.w_2] \supset [w_1 \lor w_2]$ by 5, 7 and PR
9. $\vdash [w_1 \lor \forall x.w_2] \supset \forall x.(w_1 \lor w_2)$ by $\forall$I, since x is not free in $w_1 \lor \forall x.w_2$
10. $\vdash \forall x.(w_1 \lor w_2) \equiv [w_1 \lor \forall x.w_2]$ by 4, 9 and PR ∎

T44. $\vdash \exists x.(w_1 \land w_2) \equiv [w_1 \land \exists x.w_2]$ where x is not free in w_1

Proof: By duality on the previous theorem.

The following two theorems show that the $\bigcirc$ operator also commutes with the quantifiers.

T45. $\vdash (\forall x. \bigcirc w) \equiv (\bigcirc \forall x.w)$

Proof:

1. $\vdash (\forall x. \bigcirc w) \supset (\bigcirc \forall x.w)$ by A13
2. $\vdash \forall x.w \supset w$ by A12
3. $\vdash (\bigcirc \forall x.w) \supset \bigcirc w$ by $\bigcirc\bigcirc$
4. $\vdash (\bigcirc \forall x.w) \supset (\forall x. \bigcirc w)$ by $\forall$I
5. $\vdash (\forall x. \bigcirc w) \equiv (\bigcirc \forall x.w)$ by 1, 4 and PR ∎

T46. $\vdash (\exists x. \bigcirc w) \equiv (\bigcirc \exists x.w)$

Proof:

1. $\vdash (\forall x. \bigcirc \sim w) \equiv (\bigcirc \forall x. \sim w)$ by T45

2. $\vdash (\forall x. \sim \bigcirc w) \equiv (\bigcirc \sim\exists x.w)$ by A4, A11 and ER
3. $\vdash (\sim\exists x. \bigcirc w) \equiv (\sim \bigcirc \exists x.w)$ by A4, A11 and PR
4. $\vdash (\exists x. \bigcirc w) \equiv (\bigcirc \exists x.w)$ by PR ∎

The following two theorems show that each temporal operator commutes with the quantifier that has similar character (universal, or existential).

T47. $\vdash (\forall x. \Box w) \equiv (\Box \forall x.w)$

Proof:

1. $\vdash \Box w \supset [w \wedge \bigcirc \Box w]$ by T20 and PR
2. $\vdash (\forall x. \Box w) \supset \forall x.(w \wedge \bigcirc \Box w)$ by $\forall\forall$
3. $\vdash (\forall x. \Box w) \supset [(\forall x.w) \wedge (\forall x. \bigcirc \Box w)]$ by T41 and PR
4. $\vdash (\forall x. \Box w) \supset [(\forall x.w) \wedge (\bigcirc \forall x. \Box w)]$ by T45 and PR
5. $\vdash (\forall x. \Box w) \supset (\Box \forall x.w)$ by DCI, taking u to be $\forall x. \Box w$ and v to be $\forall x.w$
6. $\vdash (\forall x.w) \supset w$ by A12
7. $\vdash (\Box \forall x.w) \supset \Box w$ by $\Box\Box$
8. $\vdash (\Box \forall x.w) \supset (\forall x. \Box w)$ by $\forall$I
9. $\vdash (\forall x. \Box w) \equiv (\Box \forall x.w)$ by 5, 8 and PR ∎

T48. $\vdash (\exists x. \Diamond w) \equiv (\Diamond \exists x.w)$

Proof:

1. $\vdash (\forall x. \Box \sim w) \equiv (\Box \forall x. \sim w)$ by T47
2. $\vdash (\forall x. \sim \Diamond w) \equiv (\Box \sim\exists x.w)$ by A1, A11 and ER (twice)
3. $\vdash (\sim\exists x. \Diamond w) \equiv (\sim \Diamond \exists x.w)$ by A1, A11 and PR
4. $\vdash (\exists x. \Diamond w) \equiv (\Diamond \exists x.w)$ by PR ∎

Theorem T47 implies the commutativity of $\forall$ with $\Box$: Both have a universal character, with one quantifying over individuals and the other quantifying over states. Similarly, theorem T48 implies the commutativity of $\exists$ with $\Diamond$. The first two theorems (T45 and T46) imply the commutativity of $\forall$ and $\exists$ with $\bigcirc$.

The next two theorems are consistent with the interpretation that the $\mathcal{U}$ operator is universal with respect to its first argument and existential with respect to the second.

T49. $\vdash \forall x.(w_1 \mathcal{U} w_2) \equiv (\forall x.w_1)\mathcal{U} w_2$ where x is not free in w_2

Proof:

1. $\vdash w_1 \mathcal{U} w_2 \supset [w_2 \vee (w_1 \wedge \bigcirc(w_1 \mathcal{U} w_2))]$ by A9 and PR
2. $\vdash \forall x.(w_1 \mathcal{U} w_2) \supset \forall x.[w_2 \vee (w_1 \wedge \bigcirc(w_1 \mathcal{U} w_2))]$ by $\forall\forall$
3. $\vdash \forall x.(w_1 \mathcal{U} w_2) \supset [w_2 \vee \forall x.(w_1 \wedge \bigcirc(w_1 \mathcal{U} w_2))]$ by $\forall$I and PR, since x is not free in w_2
4. $\vdash \forall x.(w_1 \mathcal{U} w_2) \supset [w_2 \vee (\forall x.w_1 \wedge \forall x.\bigcirc(w_1 \mathcal{U} w_2))]$ by T41 and PR
5. $\vdash \forall x.(w_1 \mathcal{U} w_2) \supset [w_2 \vee (\forall x.w_1 \wedge \bigcirc\forall x.(w_1 \mathcal{U} w_2))]$ by T45 and PR
6. $\vdash \forall x.(w_1 \mathcal{U} w_2) \supset \Diamond w_2$ by A12, A10 and PR
7. $\vdash \forall x.(w_1 \mathcal{U} w_2) \supset (\forall x.w_1)\mathcal{U} w_2$ by 5, 6 and R$\mathcal{U}$I, taking w to be $\forall x.(w_1 \mathcal{U} w_2)$, u to be $\forall x.w_1$, and v to be w_2
8. $\vdash (\forall x.w_1) \supset w_1$ by A12
9. $\vdash (\forall x.w_1)\mathcal{U} w_2 \supset w_1 \mathcal{U} w_2$ by $\mathcal{U}\mathcal{U}$
10. $\vdash (\forall x.w_1)\mathcal{U} w_2 \supset \forall x.(w_1 \mathcal{U} w_2)$ by $\forall$I, since x is not free in w_2
11. $\vdash \forall x.(w_1 \mathcal{U} w_2) \equiv (\forall x.w_1)\mathcal{U} w_2$ by 7, 10 and PR ∎

T50. $\vdash \exists x.(w_1 \mathcal{U} w_2) \equiv w_1 \mathcal{U}(\exists x.w_2)$ where x is not free in w_1

Proof:

1. $\vdash w_1 \mathcal{U} w_2 \supset \Diamond w_2$ by A10
2. $\vdash \exists x.(w_1 \mathcal{U} w_2) \supset (\exists x.\Diamond w_2)$ by $\exists\exists$
3. $\vdash \exists x.(w_1 \mathcal{U} w_2) \supset (\Diamond \exists x.w_2)$ by T48 and PR
4. $\vdash w_1 \mathcal{U} w_2 \supset [w_2 \vee (w_1 \wedge \bigcirc(w_1 \mathcal{U} w_2))]$ by A9 and PR
5. $\vdash \exists x.(w_1 \mathcal{U} w_2) \supset [(\exists x.w_2) \vee \exists x.(w_1 \wedge \bigcirc(w_1 \mathcal{U} w_2))]$ by T42, $\exists\exists$ and PR

6. $\vdash \exists x.(w_1 \mathcal{U} w_2) \supset [(\exists x.w_2) \lor (w_1 \land \exists x.\bigcirc(w_1 \mathcal{U} w_2))]$ by T44 and PR, since x is not free in w_1

7. $\vdash \exists x.(w_1 \mathcal{U} w_2) \supset \{(\exists x.w_2) \lor [w_1 \land \bigcirc \exists x.(w_1 \mathcal{U} w_2)]\}$ by T46 and PR

8. $\vdash \exists x.(w_1 \mathcal{U} w_2) \supset w_1 \mathcal{U}(\exists x.w_2)$ by 3, 7, R$\mathcal{U}$I and PR

9. $\vdash [w_2 \lor (w_1 \land \bigcirc(w_1 \mathcal{U} w_2))] \supset w_1 \mathcal{U} w_2$ by A9 and PR

10. $\vdash \exists x.[w_2 \lor (w_1 \land \bigcirc(w_1 \mathcal{U} w_2))] \supset \exists x.(w_1 \mathcal{U} w_2)$ by $\exists\exists$

11. $\vdash [(\exists x.w_2) \lor \exists x.(w_1 \land \bigcirc(w_1 \mathcal{U} w_2))] \supset \exists x.(w_1 \mathcal{U} w_2)$ by T42 and PR

12. $\vdash [(\exists x.w_2) \lor (w_1 \land \exists x.\bigcirc(w_1 \mathcal{U} w_2))] \supset \exists x.(w_1 \mathcal{U} w_2)$ by T44 and PR, since x is not free in w_1

13. $\vdash [(\exists x.w_2) \lor (w_1 \land \bigcirc \exists x.(w_1 \mathcal{U} w_2))] \supset \exists x.(w_1 \mathcal{U} w_2)$ by T46 and PR

14. $\vdash w_1 \mathcal{U}(\exists x.w_2) \supset \exists x.(w_1 \mathcal{U} w_2)$ by L$\mathcal{U}$I, taking u to be w_1, v to be $\exists x.w_2$ and w to be $\exists x.(w_1 \mathcal{U} w_2)$

15. $\vdash \exists x.(w_1 \mathcal{U} w_2) \equiv w_1 \mathcal{U}(\exists x.w_2)$ by 8, 14 and PR ■

While operators of similar character, i.e., both universal or both existential, commute to yield equivalent formulas, operators of opposite character usually admit implication in one direction only. Thus we have:

T51. $\vdash \exists x.\Box w \supset \Box \exists x.w$

T52. $\vdash \Diamond \forall x.w \supset \forall x.\Diamond w$

T53(a). $\vdash \exists x.(w_1 \mathcal{U} w_2) \supset (\exists x.w_1) \mathcal{U} w_2$ where x is not free in w_2

(b). $\vdash w_1 \mathcal{U}(\forall x.w_2) \supset \forall x.(w_1 \mathcal{U} w_2)$ where x is not free in w_1

Theorems of similar character are:

T54(a). $\vdash \exists x.(u \mathcal{U} v) \supset (\exists x.u) \mathcal{U} (\exists x.v)$

(b). $\vdash (\forall x.u) \mathcal{U} (\forall x.v) \supset \forall x.(u \mathcal{U} v)$

THE NEXT OPERATOR APPLIED TO TERMS:

The use of the next operator $\bigcirc$ applied to terms is governed by the axioms:

A14. $\vdash \bigcirc f(t_1, \ldots, t_n) = f(\bigcirc t_1, \ldots, \bigcirc t_n)$
for any function f and terms $t_1, \ldots, t_n$

A15. $\vdash \bigcirc p(t_1, \ldots, t_n) \equiv p(\bigcirc t_1, \ldots, \bigcirc t_n)$
for any predicate p and terms $t_1, \ldots, t_n$

These axioms are consistent with the evaluation rules that we gave which stated that in order to evaluate an expression $\bigcirc \mathcal{E}(t_1, \ldots, t_n)$, we can evaluate $\mathcal{E}(\bigcirc t_1, \ldots, \bigcirc t_n)$ whether $\mathcal{E}$ is a function or a predicate.

5. EQUALITY

Equality is handled by the following axioms:

AXIOMS:

A16. *Reflexivity of Equality*

$\vdash t = t$ for any term t

A17. *Substitutivity of Equality*

$\vdash (t_1 = t_2) \supset [w(t_1, t_1) \equiv w(t_1, t_2)]$
where t_2 is any term globally free for t_1 in w
and where w does not contain temporal operators

A18. $\vdash \bigcirc(t_1 = t_2) \equiv (\bigcirc t_1 = \bigcirc t_2)$

We use $w(t_1, t_2)$ to indicate that t_2 replaces some of the occurrences of t_1 in w.

The axiom A18 is a special case of A15 when the predicate p is the equality predicate.

Recall that a term t_2 is said to be ***globally free for*** t_1 ***in*** w if substitution of t_2 for all free occurrences of t_1 in w: (a) does not create new bound occurrences of (global) variables, (i.e., t_2 is ***free for*** t_1 ***in*** w), and (b) does not create new occurrences of local variables in the scope of a modal operator.

Note that the classical axiom for substitutivity of equality A17

$$\vdash (t_1 = t_2) \supset [w(t_1, t_1) \equiv w(t_1, t_2)]$$

(where t_2 is free for t_1 in w) is not correct if w contains temporal operators. We could take $w(t_1, t_2)$ to be $\Box(t_1 = t_2)$ and deduce from A17

$$\vdash (t_1 = t_2) \supset [\Box(t_1 = t_1) \equiv \Box(t_1 = t_2)],$$

i.e.,

$$\vdash (t_1 = t_2) \supset \Box(t_1 = t_2),$$

which is not a valid statement (since $t_1 = t_2$ may contain local variables).

T55. *Commutativity of Equality*

$$\vdash (t_1 = t_2) \supset (t_2 = t_1)$$

Proof:

1. $\vdash (t_1 = t_2) \supset [(t_1 = t_1) \equiv (t_2 = t_1)]$ by A17
2. $\vdash t_1 = t_1$ by A16
3. $\vdash (t_1 = t_2) \supset (t_2 = t_1)$ by 1, 2 and PR ⌟

T56. *Transitivity of Equality*

$$\vdash [(t_1 = t_2) \wedge (t_2 = t_3)] \supset (t_1 = t_3)$$

Proof:

1. $\vdash (t_1 = t_2) \supset [(t_1 = t_3) \equiv (t_2 = t_3)]$ by A17
2. $\vdash [(t_1 = t_2) \wedge (t_2 = t_3)] \supset (t_1 = t_3)$ by PR ⌟

T57. *Term Equality*

(a) $\vdash \Box(t_1 = t_2) \supset [\tau(t_1, t_1) = \tau(t_1, t_2)]$ for any term τ

(b) $\vdash (t_1 = t_2) \supset [\tau(t_1, t_1) = \tau(t_1, t_2)]$
provided τ does not contain the next operator.

Proof of (a):

By induction on the structure of τ.

Case: $\tau(t_1, t_1) = t_1$ *and* $\tau(t_1, t_2) = t_1$. Then

1. $\vdash t_1 = t_1$ by A16
2. $\vdash \Box(t_1 = t_2) \supset [\tau(t_1, t_1) = \tau(t_1, t_2)]$
by PR and definition of $\tau(t_1, t_1)$ and $\tau(t_1, t_2)$

Case: $\tau(t_1,t_1) = t_1$ *and* $\tau(t_1,t_2) = t_2$. Then

1. $\vdash \Box(t_1 = t_2) \supset (t_1 = t_2)$ by A3
2. $\vdash \Box(t_1 = t_2) \supset [\tau(t_1,t_1) = \tau(t_1,t_2)]$ by the definition of $\tau(t_1,t_1)$ and $\tau(t_1,t_2)$

Case: $\tau(t_1,t_1) = f(\tau_1(t_1,t_1), \ldots, \tau_k(t_1,t_1))$ *and* $\tau(t_1,t_2) = f(\tau_1(t_1,t_2), \ldots, \tau_k(t_1,t_2))$. Then

1. $\vdash \Box(t_1 = t_2) \supset [\tau_i(t_1,t_1) = \tau_i(t_1,t_2)]$, for $i = 1, \ldots, k$ by the induction assumption.
2. $\vdash \bigwedge_{i=1}^{k} [\tau_i(t_1,t_1) = \tau_i(t_1,t_2)] \supset$ $[f(\tau_1(t_1,t_1), \ldots, \tau_k(t_1,t_1)) = f(\tau_1(t_1,t_2), \ldots, \tau_k(t_1,t_2))]$ by repeated application of A17 and using T56 for transitivity of equality.

A typical step in this repeated application is:

$$\vdash [\tau_i(t_1,t_1) = \tau_i(t_1,t_2)] \supset$$
$$[f(\tau_1(t_1,t_2), \ \ldots, \ \tau_{i-1}(t_1,t_2), \ \tau_i(t_1,t_1), \ \ldots, \ \tau_k(t_1,t_1)) =$$
$$f(\tau_1(t_1,t_2), \ \ldots, \ \tau_{i-1}(t_1,t_2), \ \tau_i(t_1,t_2), \ \tau_{i+1}(t_1,t_1), \ \ldots, \ \tau_k(t_1,t_1))]$$

justified by A17 and the fact that $\tau_i(t_1,t_2)$ is free for $\tau_i(t_1,t_1)$ in $f(\ldots)$ since f does not contain any temporal operators.

3. $\vdash \Box(t_1 = t_2) \supset [\tau(t_1,t_1) = \tau(t_1,t_2)]$ by 1, 2, PR and the definition of $\tau(t_1,t_1)$ and $\tau(t_1,t_2)$.

Case: $\tau(t_1,t_1) = \bigcirc\,\tau'(t_1,t_1)$ *and* $\tau(t_1,t_2) = \bigcirc\,\tau'(t_1,t_2)$. Then

1. $\vdash \Box(t_1 = t_2) \supset [\tau'(t_1,t_1) = \tau'(t_1,t_2)]$ by the induction hypothesis
2. $\vdash \bigcirc\Box(t_1 = t_2) \supset \bigcirc[\tau'(t_1,t_1) = \tau'(t_1,t_2)]$ by $\bigcirc\bigcirc$
3. $\vdash \bigcirc[\tau'(t_1,t_1) = \tau'(t_1,t_2)] \supset [\bigcirc\,\tau'(t_1,t_1) = \bigcirc\,\tau'(t_1,t_2)]$ by A18 and PR
4. $\vdash \Box(t_1 = t_2) \supset \bigcirc\Box(t_1 = t_2)$ by A7
5. $\vdash \Box(t_1 = t_2) \supset (\bigcirc\,\tau'(t_1,t_1) = \bigcirc\,\tau'(t_1,t_2))$ by 4, 2, 3 and PR
6. $\vdash \Box(t_1 = t_2) \supset [\tau(t_1,t_1) = \tau(t_1,t_2)]$ by the definition of $\tau(t_1,t_1)$, $\tau(t_1,t_2)$.

Proof of (b):

1. $\vdash (t_1 = t_2) \supset [(\tau(t_1) = \tau(t_2)) \equiv (\tau(t_2) = \tau(t_2))]$ by A17 (no $\bigcirc$ in τ)
2. $\vdash \tau(t_2) = \tau(t_2)$ by A16

3. $\vdash (t_1 = t_2) \supset (\tau(t_1) = \tau(t_2))$ by 1, 2 and PR ∎

The following theorem generalizes A17 to arbitrary formulas.

T58. *Substitutivity of Equality*

$$\vdash \Box(t_1 = t_2) \supset [w(t_1, t_1) \equiv w(t_1, t_2)] \qquad \text{where } t_2 \text{ is free for } t_1 \text{ in } w.$$

Proof:

By induction on the structure of w.

Case: w contains no temporal operators. Then

1. $\vdash (t_1 = t_2) \supset [w(t_1, t_1) \equiv w(t_1, t_2)]$ by A17
2. $\vdash \Box(t_1 = t_2) \supset (t_1 = t_2)$ by A3
3. $\vdash \Box(t_1 = t_2) \supset [w(t_1, t_1) \equiv w(t_1, t_2)]$ by MP

Case: $w(t_1, t_2)$ is of the form $\tau_1(t_1, t_2) = \tau_2(t_1, t_2)$. Then

1. $\vdash \Box(t_1 = t_2) \supset [\tau_1(t_1, t_1) = \tau_1(t_1, t_2)]$ by T57
2. $\vdash \Box(t_1 = t_2) \supset [\tau_2(t_1, t_1) = \tau_2(t_1, t_2)]$ by T57
3. $\vdash [\tau_1(t_1, t_1) = \tau_1(t_1, t_2)] \supset [(\tau_1(t_1, t_1) = \tau_2(t_1, t_1)) \equiv (\tau_1(t_1, t_2) = \tau_2(t_1, t_1))]$
   by A17 of the form $(\theta_1 = \theta_2) \supset [(\theta_1 = \tau_2(t_1, t_1)) \equiv (\theta_2 = \tau_2(t_1, t_1))]$
   with $\theta_1 = \tau_1(t_1, t_1)$ and $\theta_2 = \tau_1(t_1, t_2)$
4. $\vdash \Box(t_1 = t_2) \supset [(\tau_1(t_1, t_1) = \tau_2(t_1, t_1)) \equiv (\tau_1(t_1, t_2) = \tau_2(t_1, t_1))]$
   by 1, 3 and PR
5. $\vdash \Box(t_1 = t_2) \supset [(\tau_1(t_1, t_2) = \tau_2(t_1, t_1)) \equiv (\tau_1(t_1, t_2) = \tau_2(t_1, t_2))]$
   similarly by A17, using 2
6. $\vdash \Box(t_1 = t_2) \supset [(\tau_1(t_1, t_1) = \tau_2(t_1, t_1)) \equiv (\tau_1(t_1, t_2) = \tau_2(t_1, t_2))]$
   by 4, 5 and PR
7. $\vdash \Box(t_1 = t_2) \supset [w(t_1, t_1) \equiv w(t_1, t_2)]$ by the definition of $w(t_1, t_2)$

Case: w is of the form $\Box u$. Then

1. $\vdash \Box(t_1 = t_2) \supset [u(t_1, t_1) \equiv u(t_1, t_2)]$ induction hypothesis
2. $\vdash \Box(t_1 = t_2)$ assumption

3. $\vdash u(t_1,t_1) \equiv u(t_1,t_2)$ by MP

4. $\vdash \Box u(t_1,t_1) \equiv \Box u(t_1,t_2)$ by $\Box\Box$

Thus, $\Box(t_1 = t_2) \vdash [\Box u(t_1,t_1) \equiv \Box u(t_1,t_2)]$

5. $\vdash \Box\Box(t_1 = t_2) \supset [\Box u(t_1,t_1) \equiv \Box u(t_1,t_2)]$ by DED

6. $\vdash \Box(t_1 = t_2) \supset [\Box u(t_1,t_1) \equiv \Box u(t_1,t_2)]$ by T3 and PR

The cases in which w is of the form $\Diamond u$, $\bigcirc u$, $\forall x.u$ and $\exists x.u$ are treated similarly, using the $\Diamond\Diamond$-rule, the $\bigcirc\bigcirc$-rule, the $\forall\forall$-rule and the $\exists\exists$-rule, respectively.

Case: w is of the form $u \mathcal{U} v$.

1. $\vdash \Box(t_1 = t_2) \supset [u(t_1,t_1) \equiv u(t_1,t_2)]$ induction hypothesis

2. $\vdash \Box(t_1 = t_2) \supset [v(t_1,t_1) \equiv v(t_1,t_2)]$ induction hypothesis

3. $\vdash \Box(t_1 = t_2)$ assumption

4. $\vdash u(t_1,t_1) \equiv u(t_1,t_2)$ by 1, 3 and MP

5. $\vdash v(t_1,t_1) \equiv v(t_1,t_2)$ by 2, 3 and MP

6. $\vdash (u(t_1,t_1) \mathcal{U} v(t_1,t_1)) \equiv (u(t_1,t_2) \mathcal{U} v(t_1,t_2))$ by 4, 5 and ER

Thus, $\Box(t_1 = t_2) \vdash [(u(t_1,t_1) \mathcal{U} v(t_1,t_1)) \equiv (u(t_1,t_2) \mathcal{U} v(t_1,t_2))]$

7. $\vdash \Box\Box(t_1 = t_2) \supset [(u(t_1,t_1) \mathcal{U} v(t_1,t_1)) \equiv (u(t_1,t_2) \mathcal{U} v(t_1,t_2))]$ by DED

8. $\vdash \Box(t_1 = t_2) \supset [(u(t_1,t_1) \mathcal{U} v(t_1,t_1)) \equiv (u(t_1,t_2) \mathcal{U} v(t_1,t_2))]$
by T3 and PR ∎

6. FRAME AXIOMS AND RULES

In this section we consider the consequences of the partition of the set of all variables into local and global variables. By the semantic definition, global variables are given their value by the global assignment α, and these values do not vary from state to state. Consequently, for a global variable u it must be universally true that $u = \bigcirc u$, i.e., the value of u at any state is identical to its value in the next state (see A19 below). The following axioms are called ***frame axioms*** in reference to the "frame axiom" in Hoare's deductive system for program verification ([ILL]).

Recall that we split the set of our symbols into two subsets: global and local symbols. The logical consequence of this convention is the following frame axiom:

A19. *Frame Axiom*

$\vdash x = \bigcirc x$ for every global variable x

We can therefore prove by induction on the structure of the term t and the formula w the following *frame theorems*:

T59. For a term t and formula w

(a) $\vdash t = \bigcirc t$
where t is global, i.e., does not contain local symbols

(b) $\vdash w \equiv \Box w$
where w is global, i.e., does not contain local symbols.

(c) $\vdash w(\bigcirc y_1, \ldots, \bigcirc y_n) \equiv \bigcirc w(y_1, \ldots, y_n)$
where $y_1, \ldots, y_n$ are all the local variables in w.

We present several frame theorems that facilitate moving global formulas in and out of the scope of temporal operators.

T60. $\vdash \Box(w_1 \vee w_2) \equiv (w_1 \vee \Box w_2)$
where w_1 is global, i.e., contains no local symbols.

Proof:

1. $\vdash \sim w_1 \supset \Box \sim w_1$ by T59b
2. $\vdash [\Box(w_1 \vee w_2) \wedge \Box \sim w_1] \supset \Box((w_1 \vee w_2) \wedge \sim w_1)$ by T7 and PR
3. $\vdash [(w_1 \vee w_2) \wedge \sim w_1] \supset w_2$ by PT
4. $\vdash [\Box(w_1 \vee w_2) \wedge \Box \sim w_1] \supset \Box w_2$ by 2, 3, $\Box\Box$ and PR
5. $\vdash [\Box(w_1 \vee w_2) \wedge \sim w_1] \supset \Box w_2$ by 1, 4 and PR
6. $\vdash \Box(w_1 \vee w_2) \supset (w_1 \vee \Box w_2)$ by PR

7. $\vdash w_1 \supset \Box w_1$ by T59b
8. $\vdash (w_1 \vee \Box w_2) \supset (\Box w_1 \vee \Box w_2)$ by PR
9. $\vdash (\Box w_1 \vee \Box w_2) \supset \Box(w_1 \vee w_2)$ by T9
10. $\vdash (w_1 \vee \Box w_2) \supset \Box(w_1 \vee w_2)$ by 8, 9 and PR

11. $\vdash \Box(w_1 \vee w_2) \equiv (w_1 \vee \Box w_2)$ by 6, 10 and PR ∎

T61. $\vdash \Diamond(w_1 \wedge w_2) \equiv (w_1 \wedge \Diamond w_2)$ where w_1 is global.

Proof: The proof follows from T60 by duality.

A derived frame rule that we will be using is

Frame Rule — FR

$$\frac{\vdash u \supset \Diamond v}{\vdash (w \wedge u) \supset \Diamond(w \wedge v)}$$

where w is global

Proof:

1.	$\vdash u \supset \Diamond v$	given
2.	$\vdash (w \wedge u) \supset (w \wedge \Diamond v)$	by PR
3.	$\vdash (w \wedge \Diamond v) \supset \Diamond(w \wedge v)$	by T61 and PR
4.	$\vdash (w \wedge u) \supset \Diamond(w \wedge v)$	by 2, 3 and PR ▟

C. DOMAIN PART

The next part of the system contains domain axioms that specify the necessary properties of the domain of interest. Thus, to reason about programs manipulating natural numbers, we need the set of Peano Axioms, and to reason about trees we need a set of axioms giving the basic properties of trees and the basic operations defined on them.

7. INDUCTION AXIOMS AND RULES

An essential axiom schema for many domains is the *induction axiom schema.* This (and all other schemas) should be formulated to admit temporal instances as subformulas. Thus the induction principle for natural numbers can be stated as follows:

A20. *Induction Axiom*

$$\vdash \{R(0) \ \land \ \forall n[R(n) \ \supset \ R(n+1)]\} \ \supset \ R(k)$$

for any statement R.

One instance of this axiom, which will be used later, is obtained by taking $R(n)$ to be $\Box(Q(n) \supset \Diamond\psi)$:

T62. *Induction Theorem*:

$$\vdash \{\Box(Q(0) \supset \Diamond\psi) \ \land \ \forall n[\Box(Q(n) \supset \Diamond\psi) \ \supset \ \Box(Q(n+1) \supset \Diamond\psi)]\}$$
$$\supset \ \Box(Q(k) \supset \Diamond\psi).$$

Using this induction theorem we can derive the following useful induction rule:

$\Diamond$ *Induction Rule* — $\Diamond$IND

$$\frac{\begin{array}{l}\vdash Q(0) \ \supset \ \Diamond\psi \\ \vdash Q(n+1) \ \supset \ [\Diamond\psi \ \lor \ \Diamond Q(n)]\end{array}}{\vdash Q(k) \ \supset \ \Diamond\psi}$$

$\Diamond$IND is useful for proving convergence of a loop: show that $Q(0)$ guarantees $\Diamond\psi$ and that for each n, either $Q(n+1)$ implies $Q(n)$ across the loop or it already establishes $\Diamond\psi$ and no further execution is necessary. Then for any k, $Q(k)$ ensures that $\Diamond\psi$ is established.

Proof:

1. $\vdash Q(0) \supset \Diamond\psi$ given
2. $\vdash \Box(Q(0) \supset \Diamond\psi)$ by $\Box$I

3.	$\vdash Q(n+1) \supset (\Diamond \psi \vee \Diamond Q(n))$	given
4.	$\vdash \Box(Q(n) \supset \Diamond \psi) \supset (\Diamond Q(n) \supset \Diamond \psi)$	by T6, T4 and PR
5.	$\vdash [Q(n+1) \wedge \Box(Q(n) \supset \Diamond \psi)] \supset \Diamond \psi$	by 3, 4 and PR
6.	$\vdash \Box(Q(n) \supset \Diamond \psi) \supset (Q(n+1) \supset \Diamond \psi)$	by PR
7.	$\vdash \Box\Box(Q(n) \supset \Diamond \psi) \supset \Box(Q(n+1) \supset \Diamond \psi)$	by $\Box\Box$
8.	$\vdash \Box(Q(n) \supset \Diamond \psi) \supset \Box(Q(n+1) \supset \Diamond \psi)$	by T3 and PR
9.	$\vdash \forall n[\Box(Q(n) \supset \Diamond \psi) \supset \Box(Q(n+1) \supset \Diamond \psi)]$	by $\forall$I
10.	$\vdash \Box(Q(k) \supset \Diamond \psi)$	by 2, 9 and T62
11.	$\vdash Q(k) \supset \Diamond \psi$	by A3 and MP ∎

While induction over the natural numbers is usually sufficient in order to prove properties of sequential programs, we need induction over more general orderings in order to reason about concurrent programs ([LPS]). Thus we have to formulate a more general induction principle over arbitrary well-founded orderings.

Let $(\mathcal{A}, \prec)$ be a partially ordered set. We call the ordering $\prec$ a *well-founded ordering* if there exists no infinitely decreasing sequence of elements in $\mathcal{A}$:

$$\alpha_1 \succ \alpha_2 \succ \alpha_3 \succ \ldots$$

For each well-founded ordering $(\mathcal{A}, \prec)$, the following is a valid induction rule:

R5. *Well-Founded Induction Rule* — WIND

$$\frac{\vdash \forall \beta[(\beta \prec \alpha) \supset w(\beta)] \supset w(\alpha)}{\vdash w(\alpha)}$$

This rule should hold for an arbitrary temporal formula $w(\alpha)$ dependent on a global variable $\alpha \in \mathcal{A}$, and we adopt it as a primitive inference rule.

To justify the rule semantically we may argue as follows:

Assume that the premise to the rule is true but the conclusion is not. Then there must exist a model $\mathcal{M}$ and an α_1 such that $w(\alpha_1)$ is false under $\mathcal{M}$. By the premise there must exist some α_2 such that $\alpha_2 \prec \alpha_1$ and $w(\alpha_2)$ is false under $\mathcal{M}$. Arguing in a similar way we obtain an infinitely decreasing sequence:

$$\alpha_1 \succ \alpha_2 \succ \alpha_3 \succ \ldots$$

such that for each i, $w(\alpha_i)$ is false under $\mathcal{M}$. This of course contradicts the well foundedness of $(\mathcal{A}, \prec)$.

Note that the induction axiom and rules can be derived from WIND by taking $(\mathcal{A}, \prec)$ to be $(N, <)$.

In order to use the WIND rule, one has to establish that the ordering $\prec$ is indeed a well-founded ordering. Several specific orderings are known to be well-founded (such as lexicographic ordering over tuples of integers, multisets, etc.), and may be freely used. However the general statement that an ordering '$\prec$' is well-founded is a second order statement which may require second order reasoning for its establishment.

By substitution of a special form of a temporal formula we can obtain the following induction principle for $\Diamond$ formulas:

Well-Founded $\Diamond$ *Induction Rule* — $\Diamond$WIND

$$\frac{\vdash w(\alpha) \supset \Diamond(\psi \lor \exists\beta[(\beta \prec \alpha) \land w(\beta)])}{\vdash w(\alpha) \supset \Diamond\psi}$$

We show that $\Diamond$WIND follows from WIND.

Proof:

1. $\vdash w(\alpha) \supset \Diamond(\psi \lor \exists\beta[(\beta \prec \alpha) \land w(\beta)])$ given
2. $\vdash w(\alpha) \supset (\Diamond\psi \lor \Diamond\exists\beta[(\beta \prec \alpha) \land w(\beta)])$ by T8 and PR
3. $\vdash \Box(\exists\beta[(\beta \prec \alpha) \land w(\beta)] \supset \Diamond\psi) \supset$
   $(\Diamond\exists\beta[(\beta \prec \alpha) \land w(\beta)] \supset \Diamond\psi)$ by T6, T4 and PR
4. $\vdash \{w(\alpha) \land \Box(\exists\beta[(\beta \prec \alpha) \land w(\beta)] \supset \Diamond\psi)\} \supset \Diamond\psi$ by 2, 3 and PR
5. $\vdash \Box(\exists\beta[(\beta \prec \alpha) \land w(\beta)] \supset \Diamond\psi) \supset (w(\alpha) \supset \Diamond\psi)$ by PR
6. $\vdash (\exists\beta[(\beta \prec \alpha) \land w(\beta)] \supset \Diamond\psi) \equiv (\sim\exists\beta[(\beta \prec \alpha) \land w(\beta)] \lor \Diamond\psi)$ by PT
7. $\vdash (\sim\exists\beta[(\beta \prec \alpha) \land w(\beta)] \lor \Diamond\psi) \equiv (\forall\beta[\sim(\beta \prec \alpha) \lor \sim w(\beta)] \lor \Diamond\psi)$
   by A11, ER and PR
8. $\vdash (\forall\beta[\sim(\beta \prec \alpha) \lor \sim w(\beta)] \lor \Diamond\psi) \equiv \forall\beta[(\beta \prec \alpha) \supset (w(\beta) \supset \Diamond\psi)]$
   by T43, PR and ER, since $\Diamond\psi$ does not depend on β
9. $\vdash (\exists\beta[(\beta \prec \alpha) \land w(\beta)] \supset \Diamond\psi) \equiv \forall\beta[(\beta \prec \alpha) \supset (w(\beta) \supset \Diamond\psi)]$
   by 6, 7, 8 and PR
10. $\vdash \Box\forall\beta[(\beta \prec \alpha) \supset (w(\beta) \supset \Diamond\psi)] \supset (w(\alpha) \supset \Diamond\psi)$ by 9, 5 and ER
11. $\vdash \Box\forall\beta[(\beta \prec \alpha) \supset (w(\beta) \supset \Diamond\psi)] \supset \Box(w(\alpha) \supset \Diamond\psi)$ by T3, $\Box\Box$ and PR
12. $\vdash \forall\beta\,\Box[(\beta \prec \alpha) \supset (w(\beta) \supset \Diamond\psi)] \supset \Box(w(\alpha) \supset \Diamond\psi)$ by T47 and PR
13. $\vdash \forall\beta[(\beta \prec \alpha) \supset \Box(w(\beta) \supset \Diamond\psi)] \supset \Box(w(\alpha) \supset \Diamond\psi)$
    by T60, ER and PR, since $(\beta \prec \alpha)$ is global
14. $\vdash \Box(w(\alpha) \supset \Diamond\psi)$ by WIND, taking $w(\alpha)$ to be $\Box(w(\alpha) \supset \Diamond\psi)$
15. $\vdash w(\alpha) \supset \Diamond\psi$ by A3 and PR ∎

D. PROGRAM PART

Our proof system must be augmented by additional axioms that reflect the structure of the program under consideration. The additional axioms constrain the state sequences to be exactly the set of execution sequences of the program under study. This relieves us from the need to include program text explicitly in the system; all the necessary information is captured by the additional axioms.

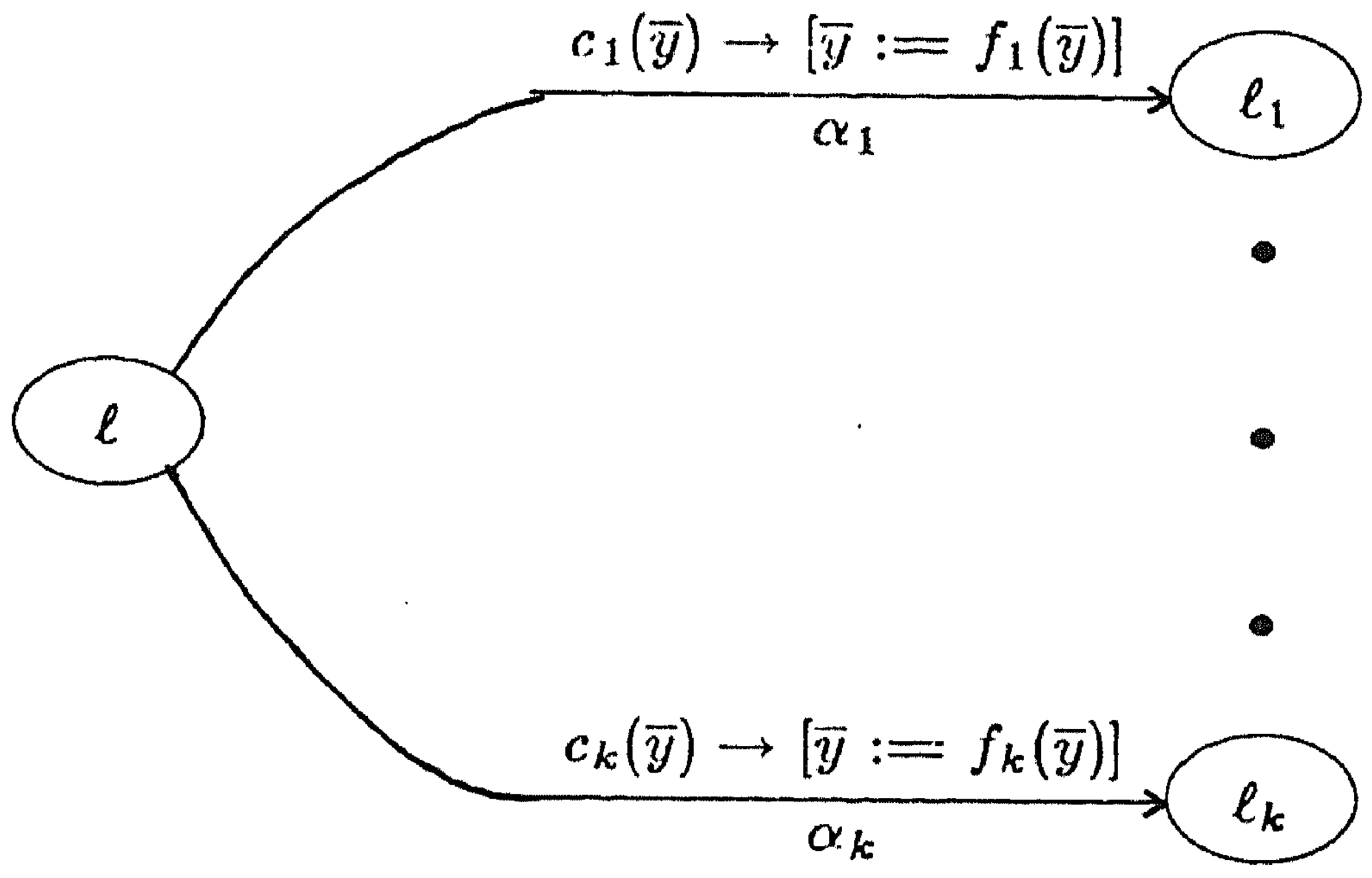
8. PROGRAMS AND COMPUTATIONS

In our model a concurrent program consists of m parallel processes:

$$P: \quad \overline{y} := g(\overline{x}); \ [P_1 \| \dots \| P_m].$$

Each process P_i is represented as a transition graph with locations (nodes) $L_i = \{\ell_0^i, \dots, \ell_t^i\}$. The edges in the graph are labelled by guarded commands of the form $c(\overline{y}) \rightarrow [\overline{y} := f(\overline{y})]$ whose meaning is that if $c(\overline{y})$ is true the edge may be traversed while replacing $\overline{y}$ by $f(\overline{y})$.

Let $\ell, \ell_1, \ell_2, \dots, \ell_k \in L_i$ be locations in process P_i:



The variables $\overline{y} = (y_1, \dots, y_n)$ are shared by all processes. We define $E_\ell(\overline{y}) = c_1(\overline{y}) \vee \dots \vee c_k(\overline{y})$ to be the ***exit condition*** at node ℓ. We do not require that the conditions c_i be either exclusive or exhaustive.

The advantage of the transition graph representation is that programs are represented in a uniform way and that we have only to deal with one type of instruction. We show first that programs represented in a linear text form can easily be translated into graph form.

Assume that a linear text program allows the following types of instructions:

Assignment: $\overline{y} := f(\overline{y})$

Conditional Branch: *if* $p(\overline{y})$ *then go to* ℓ_1 *else go to* ℓ_2

Halt: *halt*

Waiting loop: *loop until* $p(\overline{y})$

loop while $p(\overline{y})$

and the semaphore instructions

Request: *request*(y)

Release: *release*(y)

A linear text program for each of the processes has the following form:

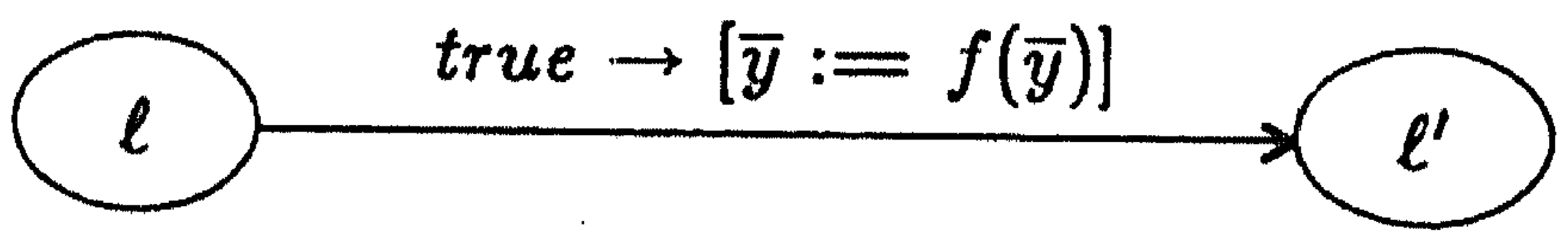
$\ell_0 :\ I_0$
$\ell_1 :\ I_1$
.
.
.
$\ell_t :$ *halt* or *go to* ℓ_j

where $\ell_0, \ell_1, \ldots, \ell_t$ are labels and $I_0, I_1, \ldots$ are instructions from the list above.

The graph representation of such a program for process P_i will be a labelled graph with $L_i = \{\ell_0, \ldots, \ell_t\}$ as the set of nodes. For each instruction I at label $\ell \in L_i$ we construct edges as follows:

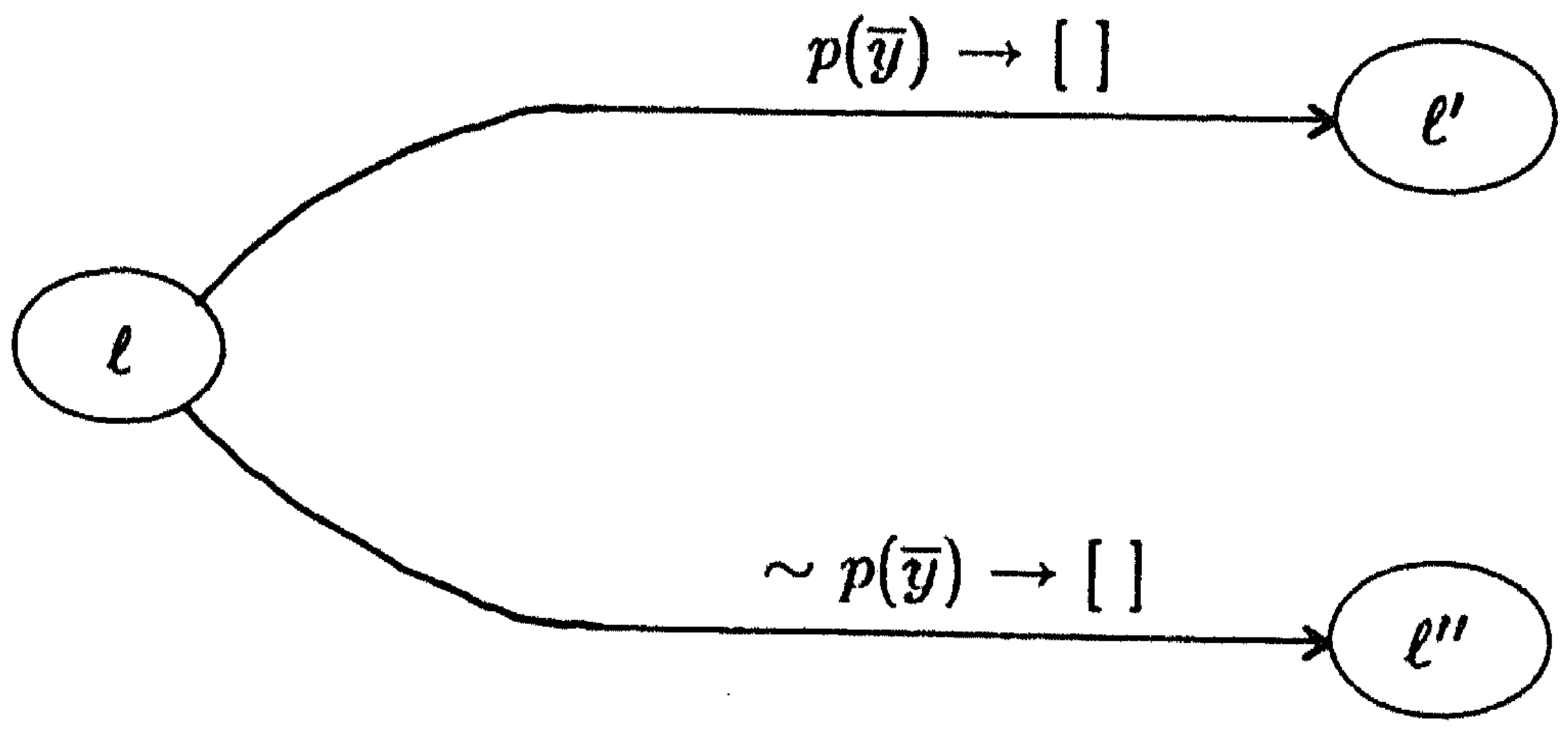
- for the instruction
  $\ell :\ \overline{y} := f(\overline{y})$
  $\ell' :$

construct



- for the instruction
  $\ell :$ *if* $p(\overline{y})$ *then go to* ℓ' *else go to* ℓ''
  $\ell' :$

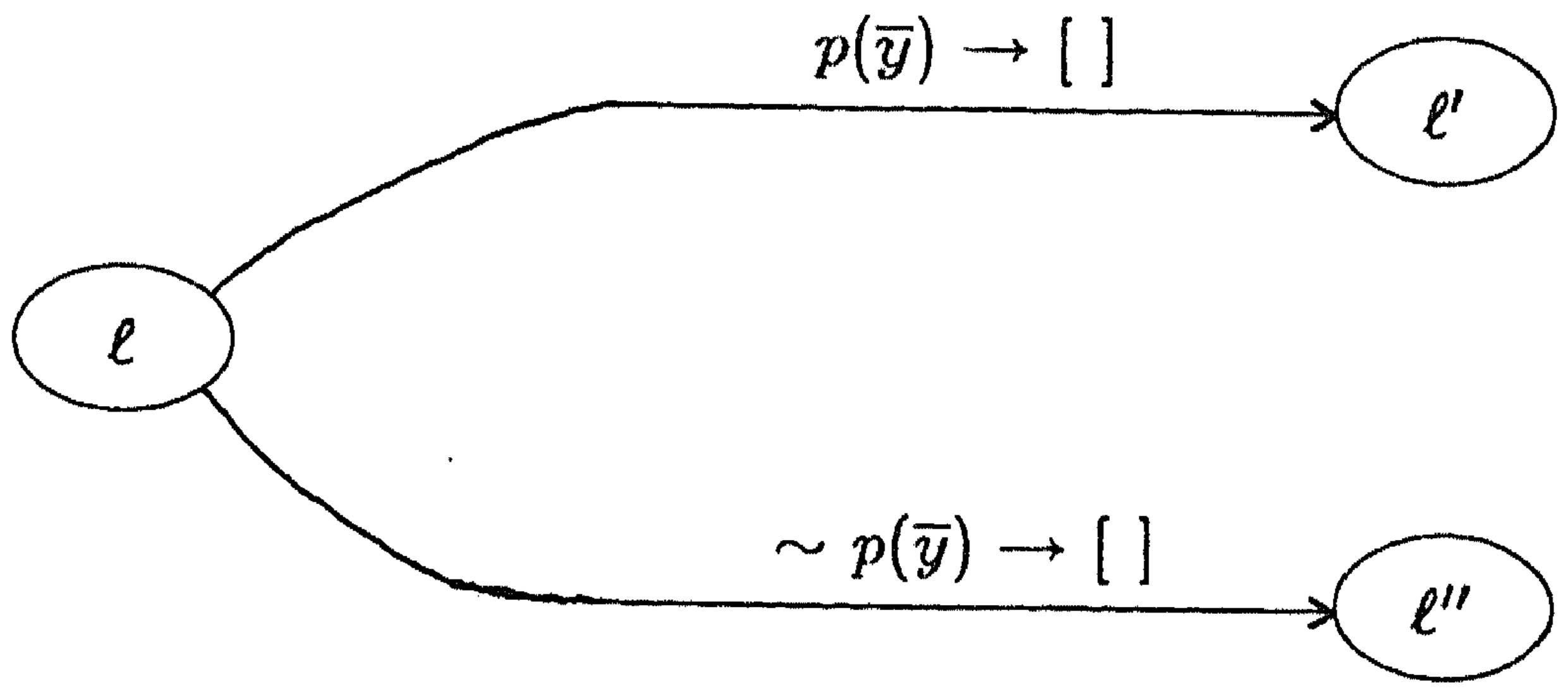
construct



- for the instruction

  ℓ : *if* $p(\bar{y})$ *then go to* ℓ'
  ℓ'' :

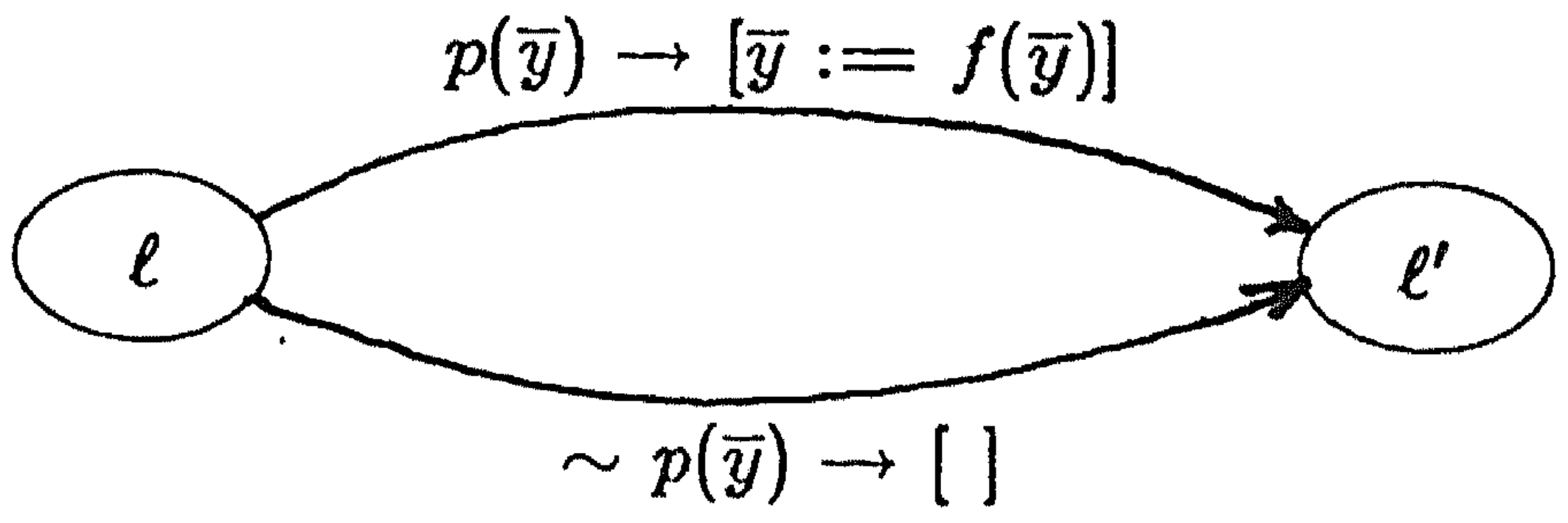
construct



- for the instruction

  ℓ : *if* $p(\bar{y})$ *then* $\bar{y} := f(\bar{y})$
  ℓ' :

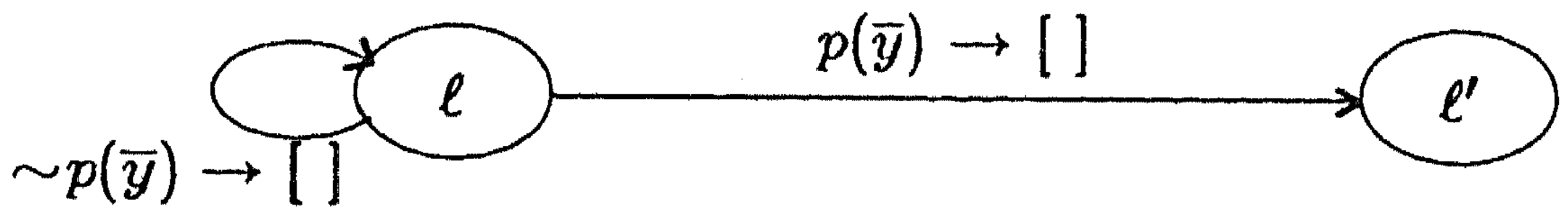
construct



- for the instruction

  ℓ : *loop until* $p(\bar{y})$
  ℓ' :

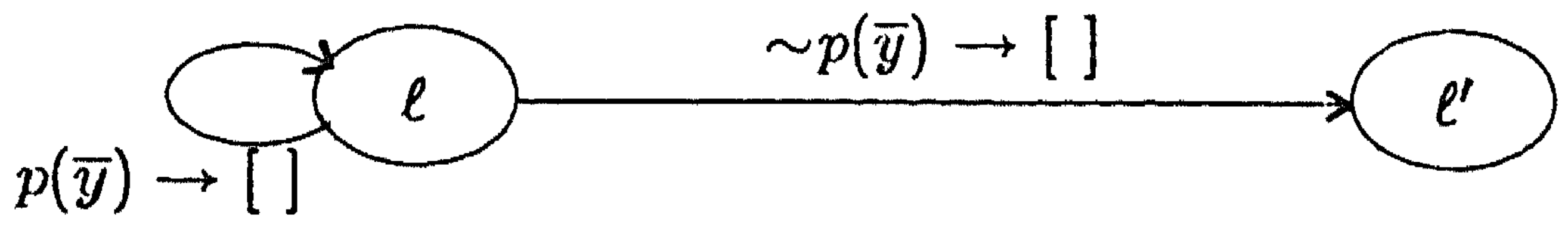
construct



- for the instruction

  ℓ : *loop while* $p(\bar{y})$
  ℓ' :

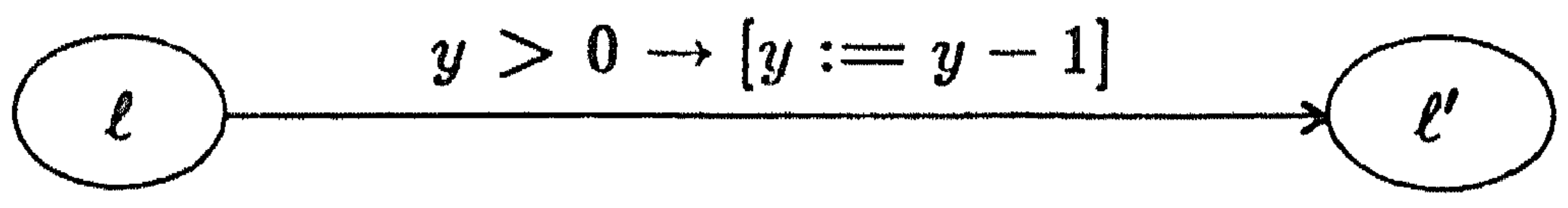
construct



- for the instruction

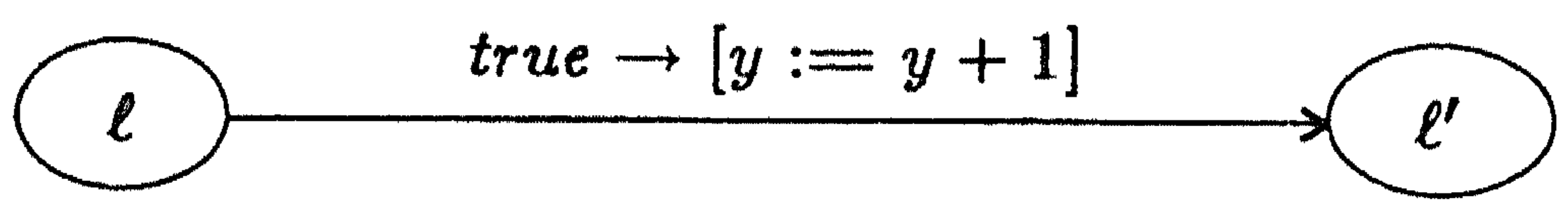
ℓ : *request*(y)
ℓ' :

construct



- ▸ for the instruction
  ℓ : *release*(y)
  ℓ' :

construct



For ***halt*** at label ℓ we construct no edges out of ℓ.

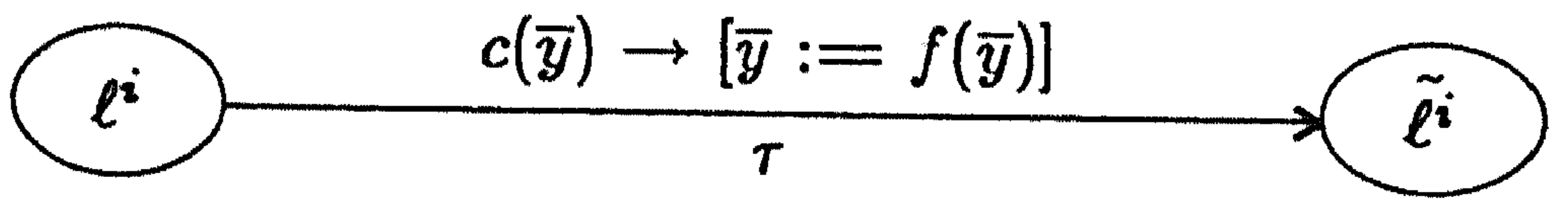
The actual translation into graph form need not be carried out explicitly. Rather, the general axiomatic description of transition diagrams can be easily translated to axioms for each of the types of instructions in the linear text form.

A state of the program P is a tuple of the form $s = \langle \bar{\ell}; \bar{\eta} \rangle$ with $\bar{\ell} \in \mathcal{L}_1 \times \ldots \times \mathcal{L}_m$ and $\bar{\eta} \in D^n$, where D is the domain over which the program variables $y_1, \ldots, y_n$ range. The vector $\bar{\ell} = (\ell^1, \ldots, \ell^m)$ is the set of current locations which are next to be executed in each of the processes. The vector $\bar{\eta}$ is the set of current values assumed by the program variables $\bar{y}$ at state s.

Let $s = \langle \ell^1, \ldots, \ell^i, \ldots, \ell^m; \bar{\eta} \rangle$ be a state. We say that process P_i is *enabled* on s if $E_{\ell^i}(\bar{\eta}) =$ *true*. This implies that if we let P_i run at this point, there is at least one condition c_j among the edges departing from ℓ^i that is true. Otherwise, we say that P_i is *disabled* on s. An example of a disabled process is the case where ℓ^i labels an instruction *request*(y) and $y = 0$. Another example is that of ℓ^i labeling a *halt* statement. A state is defined to be *terminal* if no P_i is enabled on it.

Given a program P we define the notion of a *computation step* of P.

Let $s = \langle \ell^1, \ldots, \ell^m; \bar{\eta} \rangle$ and $\tilde{s} = \langle \tilde{\ell}^1, \ldots, \tilde{\ell}^m; \tilde{\bar{\eta}} \rangle$ be two states of P. Let τ be a transition in P_i of the form:



such that $c(\bar{\eta}) =$ *true*, $\tilde{\bar{\eta}} = f(\bar{\eta})$, and for every $j \neq i$, $\tilde{\ell}^j = \ell^j$. Then we say that $\tilde{s}$ can be obtained from s by a P_i-*step* (a single computation step), and write

$$s \xrightarrow{P_i} \tilde{s}.$$

An *initialized admissible computation* of a program P for an input $\bar{x} = \bar{\xi}$ is a labelled maximal sequence of states of P:

$$\sigma : \; s_0 \xrightarrow{P_{i_1}} s_1 \xrightarrow{P_{i_2}} s_2 \xrightarrow{P_{i_3}} s_3 \longrightarrow \ldots$$

which satisfies the following three conditions. (The sequence σ is considered *maximal* if it cannot be extended, i.e., it is either infinite or ends with a state s_k which is terminal.)

A. Initialization:

The first state s_0 has the form:

$$s_0 = \langle \bar{\ell}_0; g(\bar{\xi}) \rangle$$

where $\bar{\ell}_0 = (\ell_0^1, \ldots, \ell_0^m)$ is the vector of initial locations. The values $g(\bar{\xi})$ are the initial values assigned to the $\bar{y}$ variables for the input $\bar{\xi}$.

B. State to State Sequencing:

Every step in the computation $s \xrightarrow{P_i} \tilde{s}$, is justified by a P_i-step.

C. Fairness:

Every P_i which is enabled on infinitely many states in σ must be activated infinitely many times in σ, i.e., there must be an infinite number of P_i-steps in σ.

We define an *admissible computation* of P for input $\bar{\xi}$ to be either an initialized admissible computation or a suffix of an initialized admissible computation.

Thus the class of admissible computations is closed under the operation of taking the suffix. This is needed in order to ensure soundness of the inference rule $\Box$I (R3). We denote the class of all $\bar{\xi}$-admissible computations of a program P by $\mathcal{A}(P, \bar{\xi})$.

An admissible computation is said to be *convergent* if it is finite:

$$\sigma : \; s_0 \xrightarrow{P_{i_1}} s_1 \longrightarrow \; \ldots \; \xrightarrow{P_{i_f}} s_f \, .$$

If the terminal state s_f in a convergent computation is of the form $s_f = \langle \ell_t^1, \ldots, \ell_t^m; \bar{\eta} \rangle$, where each ℓ_t^i labels a halt instruction, we say that the *computation has terminated.* Otherwise, we say that the *computation has blocked* or *is deadlocked.*

In order to describe properties of states we introduce a vector of *location variables* $\bar{\pi} = (\pi_1, \ldots, \pi_m)$. Each π_i ranges over L_i, and assumes the location value ℓ^i in a state

$$s = \langle \ell^1, \ldots, \ell^i, \ldots, \ell^m; \bar{\eta} \rangle .$$

Thus we may describe a state $s = \langle \bar{\ell}; \bar{\eta} \rangle$ by saying that in this state $\bar{\pi} = \bar{\ell}$ and $\bar{y} = \bar{\eta}$.

A *state formula* $Q = Q(\bar{\pi}; \bar{y})$ is any formula which contains no temporal operators. It is built up of terms and predicates over the location and program variables $(\bar{\pi}; \bar{y})$ and may also refer to global variables.

We frequently abbreviate the statement $\pi_i = \ell$ to *at* ℓ. Since the L_i's are disjoint, there is no difficulty in identifying the particular π_i which assumes the value ℓ.

Let us consider a program P over a domain D with fixed interpretation I for all the predicate, function and individual constant symbols. A model M is said to be ***admissible*** for P if it has the form:

$$M = (I, \alpha, \hat{\sigma})$$

where α and $\hat{\sigma}$ satisfy the following condition:

There exists an $\alpha[\bar{x}]$-admissible computation $\sigma \in \mathcal{A}(P, \alpha[\bar{x}])$ such that

either

$$\sigma \text{ is infinite:} \quad \sigma \;=\; s_0 \xrightarrow{P_{i_1}} s_1 \xrightarrow{P_{i_2}} s_2 \longrightarrow s_3 \ldots$$

and

$$\hat{\sigma} \;=\; s_0,\ s_1,\ s_2,\ \ldots$$

or

$$\sigma \text{ is finite:} \quad \sigma \;=\; s_0 \xrightarrow{P_{i_1}} s_1 \xrightarrow{P_{i_2}} s_2 \longrightarrow \ldots \xrightarrow{P_{i_f}} s_f$$

and then

$$\hat{\sigma} \;=\; s_0,\ s_1,\ s_2,\ \ldots,\ s_f,\ s_f,\ \ldots.$$

Thus we force $\hat{\sigma}$ to be always infinite by indefinitely repeating the last state of σ if it is finite. This corresponds to our intuition that while the computation may have terminated, time still marches on, but no further change in the program will ever occur.

Let us denote the class of all admissible models for a program P by $C(P)$. Note that this class, differently from $\mathcal{A}(P, \bar{\xi})$, contains computations corresponding to different inputs.

We define the state formula stating that a process P_i is enabled as follows:

$$Enabled(P_i; \bar{\pi}; \bar{y}) = \bigwedge_{\ell \in L_i} [(\pi_i = \ell) \supset E_\ell(\bar{y})].$$

For the complete program P we defined

$$Enabled(P; \bar{\pi}; \bar{y}) = \bigvee_{i=1}^{m} Enabled(P_i; \bar{\pi}; \bar{y}).$$

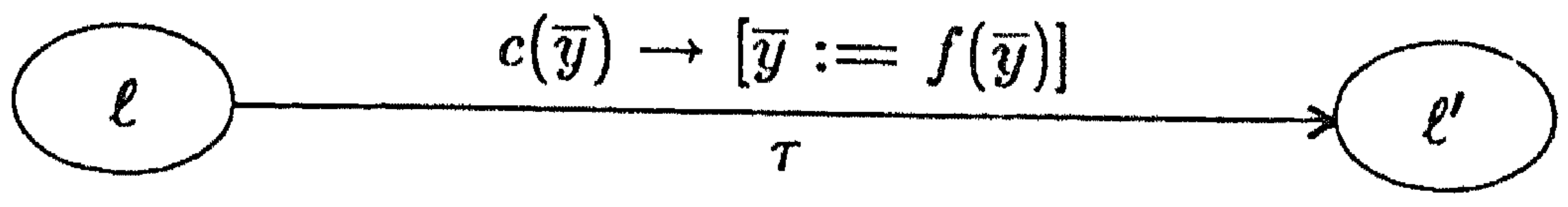
Thus a state $s = \langle \bar{\ell}; \bar{\eta} \rangle$ is terminal iff

$$Enabled(P; \bar{\ell}; \bar{\eta}) = \mathit{false}$$

and we may define

$$Terminal(\bar{\pi}; \bar{y}) \;\equiv\; {\sim}Enabled(P; \bar{\pi}; \bar{y}).$$

Let the following be a transition τ in process P_i:



We define the transformation associated with the transition τ by:

$$r_\tau(\overline{\pi}; \overline{y}) = \big(\overline{\pi}[\ell'/\pi_i]; f(\overline{y})\big).$$

The transformation is obtained by replacing the current value ℓ of π_i by ℓ' and the values of $\overline{y}$ by $f(\overline{y})$.

Let $\varphi(\overline{\pi}; \overline{y})$ and $\psi(\overline{\pi}; \overline{y})$ be two state formulas. We say:

- *The transition τ leads from φ to ψ* if the following implication is valid:
$$[\varphi(\overline{\pi}; \overline{y}) \;\wedge\; at\,\ell \;\wedge\; c(\overline{y})] \;\supset\; \psi\big(r_\tau(\overline{\pi}; \overline{y})\big).$$
- *The process P_i leads from φ to ψ* if every transition τ in P_i leads from φ to ψ.
- *The program P leads from φ to ψ* if every P_i leads from φ to ψ.

We are ready now to give a temporal axiomatization for the notion of computation under the program P.

9. AXIOMS AND RULES FOR CONCURRENT PROGRAMS

The first axiom states that the location variable π_i may only assume values in L_i.

A21. *Location Axiom* — LOC

$$\vdash \pi_i \in L_i \quad \text{for } i = 1, \ldots, m.$$

This is an abbreviation for:

$$\vdash (\pi_i = \ell_0^i) \;\vee\; (\pi_i = \ell_1^i) \;\vee\; \ldots \;\vee (\pi_i = \ell_t^i).$$

Since all the locations are disjoint, it also follows from the equality axioms that π_i may be equal to at most one ℓ_j^i at a time.

For each of the three requirements defining an admissible computation we have a corresponding inference rule scheme:

R6. *Initialization* — INIT

For an arbitrary temporal formula w:

$$\frac{\vdash [at\,\ell_0 \;\wedge\; \overline{y} = g(\overline{x})] \;\supset\; \Box w}{\vdash \Box w}$$

For let us assume that the premise to this rule holds. This implies that $\Box w$ is true for all initialized computations. By the semantic definition of $\Box$, this implies that w is true for every suffix of an initialized computation, i.e., for every admissible computation. Thus, w is $C(P)$-valid, and by generalization ($\Box$I) so is $\Box w$.

R7. *Transition* — TRNS

Let $\varphi(\overline{\pi};\overline{y})$ and $\psi(\overline{\pi};\overline{y})$ be two state formulas.

$$\frac{\begin{array}{l}\vdash P \text{ leads from } \varphi \text{ to } \psi \\ \vdash [\varphi(\overline{\pi};\overline{y}) \land \mathit{Terminal}(\overline{\pi};\overline{y})] \supset \psi(\overline{\pi};\overline{y})\end{array}}{\vdash \varphi \supset \bigcirc \psi}$$

Indeed let s be a state in the sequence $\hat{\sigma}$ corresponding to an admissible computation σ, and let s' be its successor in $\hat{\sigma}$. Assume that $\varphi(s)$ is true. There are two cases to be considered. In the first case, s' is derived from s by a P_i-step for some $i = 1, \ldots, m$. But then, by the first premise, P_i leads from φ to ψ and therefore ψ must be true for s'. In the other case, s is terminal and $s' = s$ the repetition of the terminal state of a finite computation. But then s is terminal and satisfies the antecedent of the second premise, leading to $\psi(s) = \psi(s') = \mathit{true}$. Hence, in both cases $\psi(s')$ must hold and the conclusion of the rule follows.

Note that the first premise to this rule requires establishing many conditions involving the individual transitions of each of the processes. However, by examining the definitions of "leading from φ to ψ" we see that they are all expressible as classical statements involving no temporal operators. Therefore this premise should be provable from the domain axioms plus the usual predicate calculus proof system. The second premise is also classical, and ensures the consequence after the sequence has reached a terminal state.

R8. *Fairness* — FAIR

Let $\varphi(\overline{\pi};\overline{y})$ and $\psi(\overline{\pi};\overline{y})$ be two state formulas and P_k be one of the processes.

$$\frac{\begin{array}{ll}\text{A.} & \vdash P \text{ leads from } \varphi \text{ to } \varphi \lor \psi \\ \text{B.} & \vdash P_k \text{ leads from } \varphi \text{ to } \psi\end{array}}{\vdash [\varphi \land \Box \Diamond \mathit{Enabled}(P_k)] \supset \varphi \mathcal{U} \psi}$$

To give a semantic justification of this rule, consider a computation such that φ is true initially. By A, φ will hold until ψ is realized, if ever. By B, once P_k will be activated in a state satisfying φ it will achieve ψ in one step. Consider now a sequence σ such that $\varphi \land \Box \Diamond \mathit{Enabled}(P_k)$ is true on σ. This means that φ is initially true and P_k is enabled infinitely many times in σ. By fairness, P_k will eventually be activated, which, if ψ has not been realized before, will achieve ψ in one step.

Since $(\varphi \mathcal{U} \psi) \supset \Diamond \psi$, we often use the FAIR rule in order to derive the consequence

$$[\varphi \land \Box \Diamond \mathit{Enabled}(P_k)] \supset \Diamond \psi.$$

There are several derived rules that can be obtained from the above axiomatization.

Invariance Rule — INV

$$\frac{\vdash P \text{ leads from } \varphi \text{ to } \varphi}{\vdash \varphi \supset \Box\varphi}$$

Proof:

1.	$\vdash P$ leads from φ to φ	given
2.	$\vdash [\varphi \wedge \mathit{Terminal}] \supset \varphi$	by PT
3.	$\vdash \varphi \supset \bigcirc\varphi$	by TRNS
4.	$\vdash \varphi \supset \Box\varphi$	by CI ∎

Initialized Invariance Rule — IINV

Let φ be a state formula

$$\frac{\begin{array}{c}\vdash [at\,\bar{\ell}_0 \wedge \bar{y} = g(\bar{x})] \supset \varphi \\ \vdash P \text{ leads from } \varphi \text{ to } \varphi\end{array}}{\vdash \Box\varphi}$$

Proof:

1.	$\vdash [at\,\bar{\ell}_0 \wedge \bar{y} = g(\bar{x})] \supset \varphi$	given
2.	$\vdash P$ leads from φ to φ	given
3.	$\vdash \varphi \supset \Box\varphi$	by 2 and INV
4.	$\vdash [at\,\ell_0 \wedge \bar{y} = g(\bar{x})] \supset \Box\varphi$	by 1, 3 and PR
5.	$\vdash \Box\varphi$	by INIT ∎

The IINV rule is the rule most often used in order to establish invariance properties of programs.

Unless Establishment Rule — UER

Let φ be a state formula

$$\frac{\vdash P \text{ leads from } \varphi \text{ to } \varphi \vee \psi}{\vdash \varphi \supset (\varphi\,\mathcal{U}\,\psi)}$$

Proof:

1.	$\vdash P$ leads from φ to $\varphi \vee \psi$	given

2. $\vdash \varphi \supset (\varphi \vee \psi)$ by PT
3. $\vdash [\varphi \wedge \mathit{Terminal}] \supset (\varphi \vee \psi)$ by PR
4. $\vdash \varphi \supset \bigcirc(\varphi \vee \psi)$ by 1, 3 and TRNS
5. $\vdash \varphi \supset (\varphi \mathfrak{U} \psi)$ by $\mathfrak{U}$I ∎

The following rule is a consequence of the FAIR rule.

Eventuality Rule — EVNT

Let $\varphi(\overline{\pi};\overline{y})$ and $\psi(\overline{\pi};\overline{y})$ be two state formulas and P_k one of the processes.

$$\begin{array}{ll} \text{A.} & \vdash P \text{ leads from } \varphi \text{ to } \varphi \vee \psi \\ \text{B.} & \vdash P_k \text{ leads from } \varphi \text{ to } \psi \\ \text{C.} & \vdash \varphi \supset \Diamond(\psi \vee \mathit{Enabled}(P_k)) \\ \hline & \vdash \varphi \supset \varphi \mathfrak{U} \psi \end{array}$$

Proof:

1. $\vdash P$ leads from φ to $\varphi \vee \psi$ given
2. $\vdash P_k$ leads from φ to ψ given
3. $\vdash \varphi \supset \Diamond(\psi \vee \mathit{Enabled}(P_k))$ given
4. $\vdash [\varphi \wedge \Box\Diamond \mathit{Enabled}(P_k)] \supset \varphi \mathfrak{U} \psi$ by 1, 2 and FAIR
5. $\vdash \varphi \supset (\Box\varphi \vee \varphi \mathfrak{U} \psi)$ by 1 and CINV
6. $\vdash [\varphi \wedge \Box{\sim}\psi] \supset \Diamond \mathit{Enabled}(P_k)$ by 3, T8, A1 and PR
7. $\vdash \Box(\varphi \wedge \Box{\sim}\psi) \supset \Box\Diamond \mathit{Enabled}(P_k)$ by $\Box\Box$
8. $\vdash [\Box\varphi \wedge \Box{\sim}\psi] \supset \Box\Diamond \mathit{Enabled}(P_k)$ by T3, T7 and PR
9. $\vdash [\Box\varphi \wedge {\sim}\Box\Diamond \mathit{Enabled}(P_k)] \supset \Diamond\psi$ by A1 and PR
10. $\vdash \Box\varphi \supset \Diamond\psi$ by 4, 9, A3, A10 and PR
11. $\vdash \Box\varphi \supset \varphi \mathfrak{U} \psi$ by 10, T24 and PR
12. $\vdash \varphi \supset \varphi \mathfrak{U} \psi$ by 5, 11 and PR ∎

In contrast with earlier rules, premise C of EVNT is not purely classical since it contains the temporal operator $\Diamond$. Since C has a form similar to the conclusion of the EVNT rule, it is to be expected that its derivation will require once more the application of the EVNT rule. This seems to imply circular reasoning. However, note that at each nested application of the EVNT rule, another P_k is taken out of consideration. This is because in trying to establish $\Diamond\, Enabled(P_k)$ we need not consider any P_k-steps at all, since when they are possible, P_k is already enabled.

A useful special case of C that frequently suffices for the application of the EVNT rule is:

$$C': \quad \vdash \varphi \supset [\psi \vee Enabled(P_k)].$$

Note that the EVNT rule can also be used to establish properties of the form

$$\varphi \supset \Diamond\psi,$$

since $\varphi \mathcal{U} \psi \supset \Diamond\psi$.

The EVNT rule is the one most often used in order to establish both eventuality (liveness) properties and precedence properties.

E. EXAMPLES

In this section we present several examples of proofs of properties of programs using the proof system described above.

10. EXAMPLE 1: DISTRIBUTED GCD

Let us consider the following example of a program computing the greatest common divisor of two positive integers in a distributed manner.

```
                         $(y_1, y_2) := (x_1, x_2)$

$\ell_0$ : if $y_1 > y_2$ then $y_1 := y_1 - y_2$          $m_0$ : if $y_1 < y_2$ then $y_2 := y_2 - y_1$
$\ell_1$ : if $y_1 \neq y_2$ then go to $\ell_0$                $m_1$ : if $y_1 \neq y_2$ then go to $m_0$
$\ell_2$ : halt                                   $m_2$ : halt

          – $P_1$ –                                      – $P_2$ –
```

We wish to prove total correctness for this program, i.e.,

Theorem:

$$\vdash [at(\ell_0, m_0) \wedge (y_1, y_2) = (x_1, x_2)] \supset \Diamond[at(\ell_2, m_2) \wedge y_1 = gcd(x_1, x_2)]$$

We will split the proof into two parts, proving separately invariance and termination.

Lemma A:

$$\vdash \Box[gcd(y_1, y_2) = gcd(x_1, x_2)]$$

Proof of Lemma A:

Let us denote $gcd(y_1, y_2) = gcd(x_1, x_2)$ by $\tilde{\varphi}(x_1, x_2, y_1, y_2)$.

It is easy to check that every transition in P leads from $\tilde{\varphi}$ to $\tilde{\varphi}$. Also

$$\vdash [(y_1, y_2) = (x_1, x_2)] \supset \tilde{\varphi}(x_1, x_2, y_1, y_2).$$

Thus we have the two premises to the HNV rule, which yields the desired result. ▟

Lemma B:

$$\vdash [at\,\ell_{0,1} \ \wedge\ at\,m_{0,1} \ \wedge\ (y_1, y_2) > 0 \ \wedge\ (y_1 + y_2) \leq n + 1) \ \wedge\ y_1 \neq y_2]$$
$$\supset \ \Diamond[at\,\ell_{0,1} \ \wedge\ at\,m_{0,1} \ \wedge\ (y_1, y_2) > 0 \ \wedge\ (y_1 + y_2 \leq n)]$$

Here we use $at\,\ell_{0,1}$ as an abbreviation for $at\,\ell_0 \vee at\,\ell_1$, $at\,m_{0,1}$ for $at\,m_0 \vee at\,m_1$ and $(y_1, y_2) > 0$ for $(y_1 > 0) \wedge (y_2 > 0)$.

Proof of Lemma B:

Let us define

$$\varphi(y_1, y_2, n): \quad at\,\ell_{0,1} \ \wedge\ at\,m_{0,1} \ \wedge\ (y_1, y_2) > 0 \ \wedge\ (y_1 + y_2 \leq n).$$

Thus we have to prove:

$$\vdash [\varphi(y_1, y_2, n + 1) \ \wedge\ (y_1 \neq y_2)] \ \supset\ \Diamond\,\varphi(y_1, y_2, n).$$

We will split the proof into two cases:

B1. $\vdash [\varphi(y_1, y_2, n + 1) \ \wedge\ (y_1 > y_2)] \ \supset\ \Diamond\,\varphi(y_1, y_2, n)$

B2. $\vdash [\varphi(y_1, y_2, n + 1) \ \wedge\ (y_1 < y_2)] \ \supset\ \Diamond\,\varphi(y_1, y_2, n)$

The lemma obviously follows from these two statements.

To prove B1 we first observe that by PR:

1. $\vdash \varphi(y_1, y_2, n + 1) \ \supset\ (at\,\ell_0 \ \vee\ at\,\ell_1)$

Consider therefore first the case that P_1 is at ℓ_0. We take

$$\varphi': \quad \varphi(y_1, y_2, n + 1) \ \wedge\ (y_1 > y_2) \ \wedge\ at\,\ell_0$$
$$\psi': \quad \varphi(y_1, y_2, n).$$

We claim that φ' and ψ' satisfy the premises of EVNT with $P_k = P_1$.

To see this, consider requirement A of EVNT that states that every transition in P leads from φ' to $\varphi' \vee \psi'$.

Consider transitions in P_2. The only relevant ones are $m_0 \rightarrow m_1$ and transitions leading out of m_1. The transition $m_0 \rightarrow m_1$ under $y_1 > y_2$ leaves φ' invariant. Again, under $y_1 > y_2$ the only transition out of m_1 goes to m_0 leaving φ' invariant.

The only transition enabled in P_1 is $\ell_0 \rightarrow \ell_1$ which replaces (y_1, y_2) by $(y_1 - y_2, y_2)$. If $y_1 + y_2 \leq n + 1$ and $y_1 > 0, y_2 > 0$ then certainly $(y_1 - y_2) + y_2 \leq n$ and $(y_1 - y_2) > 0, y_2 > 0$. Thus $\ell_0 \rightarrow \ell_1$ leads from φ' to ψ'. This also establishes requirement B with $P_k = P_1$.

Since $E_{\ell_0} = \mathit{true}$, condition C is trivially fulfilled. Consequently we conclude by the EVNT rule that $\vdash \varphi' \supset \Diamond \psi'$, i.e.,

2. $\vdash [\varphi(y_1, y_2, n+1) \wedge (y_1 > y_2) \wedge \mathit{at}\,\ell_0] \supset \Diamond \varphi(y_1, y_2, n).$

Consider next the case where P_1 is at ℓ_1. By taking

$$\varphi'' : \quad \varphi(y_1, y_2, n+1) \wedge (y_1 > y_2) \wedge \mathit{at}\,\ell_1$$
$$\psi'' = \varphi' : \quad \varphi(y_1, y_2, n+1) \wedge (y_1 > y_2) \wedge \mathit{at}\,\ell_0.$$

We can show that the premises of the EVNT rule are satisfied with respect to φ'', ψ''. Consequently we have $\vdash \varphi'' \supset \Diamond \psi''$, i.e.,

3. $\vdash [\varphi(y_1, y_2, n+1) \wedge (y_1 > y_2) \wedge \mathit{at}\,\ell_1] \supset$
   $\Diamond [\varphi(y_1, y_2, n+1) \wedge (y_1 > y_2) \wedge \mathit{at}\,\ell_0]$
4. $\vdash [\varphi(y_1, y_2, n+1) \wedge (y_1 > y_2) \wedge \mathit{at}\,\ell_1] \supset \Diamond \varphi(y_1, y_2, n)$ — by 2, 3 and $\Diamond$C
5. $\vdash [\varphi(y_1, y_2, n+1) \wedge (y_1 > y_2)] \supset \Diamond \varphi(y_1, y_2, n)$ — by 1, 2, 4 and PR

This establishes B1.

By a symmetric argument we can establish B2. By propositional reasoning B1 and B2 lead to Lemma B. ∎

Proof of theorem:

We will now proceed with the proof of the main theorem.

6. $\vdash [\varphi(y_1, y_2, n+1) \wedge (y_1 \neq y_2)] \supset \Diamond \varphi(y_1, y_2, n)$ — Lemma B
7. $\vdash \varphi(y_1, y_2, n+1) \supset [(y_1 = y_2) \vee \Diamond \varphi(y_1, y_2, n)]$ — by PR
8. $\vdash \varphi(y_1, y_2, n+1) \supset [\Diamond(y_1 = y_2) \vee \Diamond \varphi(y_1, y_2, n)]$ — by T1 and PR
9. $\vdash \sim\varphi(y_1, y_2, 0)$ — by PR, using the domain property that the conjunction $(y_1 > 0) \wedge (y_2 > 0) \wedge (y_1 + y_2 \leq 0)$ is impossible
10. $\vdash \varphi(y_1, y_2, 0) \supset \Diamond(y_1 = y_2)$ — by PR
11. $\vdash \varphi(y_1, y_2, n) \supset \Diamond(y_1 = y_2)$ — by 8, 10 and $\Diamond$IND
12. $\vdash \exists n.\varphi(y_1, y_2, n) \supset \Diamond(y_1 = y_2)$ — by $\exists$I
13. $\vdash [\mathit{at}(\ell_0, m_0) \wedge (y_1, y_2) = (x_1, x_2) > 0] \supset \exists n.\varphi(y_1, y_2, n)$

by taking $n = x_1 + x_2 > 0$.

By considering the different locations of P_1 and P_2 under the assumption that $y_1 = y_2$ it is easy (though long if carried out in full detail) to establish

14. $\vdash (y_1 = y_2) \supset \Diamond[at(\ell_2, m_2) \wedge (y_1 = y_2)]$.

By combining 12, 13 and 14 using ◇C we obtain:

15. $\vdash [at(\ell_0, m_0) \wedge (y_1, y_2) = (x_1, x_2) > 0] \supset \Diamond[at(\ell_2, m_2) \wedge (y_1 = y_2)]$.

Together with lemma A and T10 this gives

16. $\vdash [at(\ell_0, m_0) \wedge (y_1, y_2) = (x_1, x_2) > 0] \supset \Diamond[at(\ell_2, m_2) \wedge y_1 = gcd(x_1, x_2)]$
since $(y_1 = y_2) \supset y_1 = gcd(y_1, y_2)$

Note that theorem T10 enables us to infer from a previously established invariant $\vdash \Box\tilde{\varphi}$ and an implication $\vdash w_1 \supset \Diamond w_2$ the implication $\vdash w_1 \supset \Diamond(w_2 \wedge \tilde{\varphi})$. ∎

11. EXAMPLE 2: SEMAPHORES

For our next example we will present a very simple program with semaphores:

```
                        y := 1

  ℓ0 : request(y)                   m0 : request(y)
  ℓ1 : release(y)                   m1 : release(y)
  ℓ2 : go to ℓ0                     m2 : go to m0

       - P1 -                            - P2 -
```

This example models a solution to the mutual exclusion problem using semaphores.

There are two properties that we wish to prove for this program. The first is that of mutual exclusion, namely:

Lemma A:

$$\vdash \Box[(\sim at\,\ell_1) \vee (\sim at\,m_1)]$$

Proof:

Take

$$\varphi(\pi_1, \pi_2; y): \quad (at\,\ell_1 + at\,m_1 + y = 1) \wedge (y \geq 0).$$

In expressions such as the above we interpret propositions as having the numerical value 1 when true and 0 otherwise.

We can easily show that φ is preserved under every transition. For example, consider the transition $\ell_0 \rightarrow \ell_1$. When it is enabled, we have $y > 0$, and the transition assigns to the variable y the value $y - 1$ which is nonnegative. Considering the value of the sum

$$at\,\ell_1 + at\,m_1 + y,$$

$at\,\ell_1$ changes from 0 to 1 on this transition but y is decremented by 1. Consequently the value of the sum remains invariant.

Initially, $at\,\ell_1 + at\,m_1 + y = 0 + 0 + 1 = 1$ and $y = 1 \geq 0$.

Hence φ satisfies the two premises of the IINV rule, from which we conclude

$$I_1: \quad \vdash \Box[at\,\ell_1 + at\,m_1 + y = 1) \land (y \geq 0)].$$

This implies

$$\vdash \Box[at\,\ell_1 + at\,m_1 \leq 1]$$

which is equivalent to Lemma A. ∎

The second property is that of accessibility. It states that each process will eventually be admitted to its critical section. This is established by:

Lemma B:

$$\vdash at\,\ell_0 \supset \Diamond\, at\,\ell_1$$

and

$$\vdash at\,m_0 \supset \Diamond\, at\,m_1$$

Proof:

Let us define

$$\varphi_1: \quad at\,\ell_0 \land at\,m_1 \land y = 0$$

$$\psi_1: \quad y > 0$$

We show that φ_1 and ψ_1 satisfy the conditions of the EVNT rule with $k = 2$.

In fact the only enabled transition is $m_1 \rightarrow m_2$ which does lead from φ_1 to ψ_1. While at m_1, P_2 is always enabled. Thus we conclude:

1. $\vdash [at\,\ell_0 \land at\,m_1 \land y = 0] \supset \Diamond(y > 0)$ by EVNT with $k = 2$

2. $\vdash [at\,\ell_0 \;\wedge\; at\,m_1] \supset \Diamond(y > 0)$ by I_1 above, 1 and PR

3. $\vdash [at\,\ell_0 \;\wedge\; at\,m_{2,3}] \supset (y > 0)$ also by I_1 and PR

4. $\vdash at\,\ell_0 \supset \Diamond(y > 0)$ by T1, 2, 3, LOC and PR

Take now

$\varphi_2 : \; at\,\ell_0$

$\psi_2 : \; at\,\ell_1$

We check premises A to C in the EVNT rule with respect to the pair $\{\varphi_2, \psi_2\}$ taking $k = 1$. Clearly P always leads from φ_2 to $\varphi_2 \vee \psi_2$. The process P_1 always leads (when enabled) from φ_2 to ψ_2. Condition C is guaranteed by 4 above. We therefore conclude

5. $\vdash at\,\ell_0 \supset \Diamond\, at\,\ell_1$.

By a completely symmetric argument we can show that:

$\vdash at\,m_0 \supset \Diamond\, at\,m_1$. $\blacksquare$

12. EXAMPLE 3: MUTUAL EXCLUSION

As a third example we consider a program that solves the mutual exclusion problem without semaphores:

```
                         $(y_1, y_2, t) := (false, false, 1)$

$\ell_0$ : Noncritical Section                    $m_0$ : Noncritical Section
$\ell_1$ : $y_1$ := true                          $m_1$ : $y_2$ := true
$\ell_2$ : $t$ := 1                               $m_2$ : $t$ := 2
$\ell_3$ : if $y_2$ = false then go to $\ell_5$   $m_3$ : if $y_1$ = false then go to $m_5$
$\ell_4$ : if $t$ = 1 then go to $\ell_3$         $m_4$ : if $t$ = 2 then go to $m_3$
$\ell_5$ : Critical Section                       $m_5$ : Critical Section
$\ell_6$ : $y_1$ := false                         $m_6$ : $y_2$ := false
$\ell_7$ : go to $\ell_0$                         $m_7$ : go to $m_0$

           – $P_1$ –                                         – $P_2$ –
```

For convenience we will abbreviate formulas $at\,\ell_i$ to ℓ_i.

The principle of operation of this program is that each process P_i has a variable y_i, $i = 1, 2$, which expresses the process's wish to enter its critical section. The variable y_i is set to *true* at ℓ_1 and m_1 and reset to *false* at ℓ_6 and m_6, respectively. In addition, each process leaves a signature in the common variable t. The process P_1 sets it to 1 at ℓ_2 and P_2 sets it to 2 at m_2. A process P_i may enter its critical section only if either $y_j = \mathit{false}$ (meaning that the other process is not interested) or if $t = j$, for $j \neq i$. The latter case corresponds to both processes being interested in entering the critical section but P_j being the *last* to pass through the signing instructions at (ℓ_2, m_2).

To formally prove that this program is correct we first prove several invariance properties.

Lemma A:

$$\vdash y_1 \;\equiv\; \ell_{2..6}$$

Here $\ell_{2..6}$ stands for $at\,\ell_{2..6}$. Thus the lemma states that

$$y_1 = \mathit{true} \quad \text{if and only if} \quad \pi_1 \in \{\ell_2, \ell_3, \ell_4, \ell_5, \ell_6\}.$$

Proof:

To prove the Lemma we take

$$\varphi_1 : \;(y_1 \;\equiv\; \ell_{2..6})$$

and show that it is invariant under every transition, i.e., every transition leads from φ_1 to φ_1.

The only transitions that can affect the truth of φ_1 are $\ell_1 \to \ell_2$ and $\ell_6 \to \ell_7$.

In $\ell_1 \to \ell_2$ both y_1 and $at\,\ell_{2..6}$ become simultaneously true. Similarly in $\ell_6 \to \ell_7$ both y_1 and $at\,\ell_{2..6}$ become simultaneously false. Thus

1. $\vdash (y_1 \;\equiv\; \ell_{2..6}) \supset \bigcirc(y_1 \;\equiv\; \ell_{2..6})$ by TRNS
2. $\vdash \{at(\ell_0, m_0) \wedge [(y_1, y_2, t) = (\mathit{false}, \mathit{false}, 1)]\} \supset (y_1 \;\equiv\; \ell_{2..6})$
3. $\vdash \Box(y_1 \;\equiv\; \ell_{2..6})$ by 1, 2 and IINV ▄

Lemma B:

$$\vdash y_2 \;\equiv\; m_{2..6}$$

The lemma is proved by a symmetric argument.

Lemma C:

$$\vdash (t = 1) \vee (t = 2)$$

This lemma states that the only possible values of the variable t are 1 or 2.

Proof:

The Lemma is clearly provable by the IINV principle. Obviously, it is true initially since $t = 1$. The only transitions that modify the value of t set it either to 1 or to 2. Thus P always leads to a state satisfying $(t = 1) \vee (t = 2)$. ■

Lemma D:

$$\vdash \ell_{5,6} \supset [(\sim y_2) \vee (t = 2) \vee m_2]$$

Proof:

Let φ_2 stand for $\ell_{5,6} \supset [(\sim y_2) \vee (t = 2) \vee m_2]$.

It is clearly true initially since $\vdash \ell_0 \supset \sim\ell_{5,6}$. To show that every transition leads from φ_2 to φ_2, consider the only transitions that may falsify φ_2, i.e., that may possibly lead from φ_2 to $\sim\varphi_2$. Potentially they are:

- $\ell_3 \rightarrow \ell_5$. This transition is possible only under $\sim y_2$ which makes
  $$(\sim y_2) \vee (t = 2) \vee m_2$$
  true.
- $\ell_4 \rightarrow \ell_5$. This is possible only when $t \neq 1$ which by Lemma C makes
  $$(\sim y_2) \vee (t = 2) \vee m_2$$
  again true.

The other transitions we should consider are transitions of P_2 while P_1 is already at $\ell_{5,6}$. The only ones to be considered are those which affect any of the variables in $\sim y_2 \vee (t = 2) \vee m_2$.

- $m_1 \rightarrow m_2$. Causes m_2 to become true.
- $m_2 \rightarrow m_3$. Causes t to be set to 2.
- $m_6 \rightarrow m_7$. Sets y_2 to *false*, making $\sim y_2$ true.

The lemma follows by the IINV principle. ■

Lemma E:

$$\vdash m_{5,6} \supset [(\sim y_1) \vee (t = 1) \vee \ell_2]$$

The lemma is proved by a completely symmetric argument.

Theorem:

$$\vdash (\sim \ell_{5,6}) \lor (\sim m_{5,6})$$

This theorem proves the mutual exclusion of the processes.

Proof:

1. $\vdash (\ell_{5,6} \land m_{5,6}) \supset [((\sim y_2) \lor (t=2) \lor m_2) \land ((\sim y_1) \lor (t=1) \lor \ell_2)]$ by lemmas C, D and PR
2. $\vdash (\ell_{5,6} \land m_{5,6}) \supset [y_1 \land y_2 \land \sim \ell_2 \land \sim m_2]$ by lemmas A, B, LOC and PR
3. $\vdash (\ell_{5,6} \land m_{5,6}) \supset [(t=1) \land (t=2)]$ by 1, 2 and PR
4. $\vdash \sim(\ell_{5,6} \land m_{5,6})$ by the equality axioms and PR, using the domain fact that $1 \neq 2$
5. $\vdash (\sim \ell_{5,6}) \lor (\sim m_{5,6})$ by PR ∎

Next we will prove accessibility. We will only prove:

Theorem:

$$\vdash at\ell_1 \supset \Diamond\, at\ell_5$$

The result for P_2 is completely symmetric.

Proof:

The proof will proceed by a sequence of statements most of which are proved by the EVNT rule in the version whose conclusion is $\varphi \supset \Diamond \psi$. Simple passages justified by propositional temporal reasoning will not be fully presented and their omission is denoted by mentioning PTR in the justification clause.

1. $\vdash (\ell_4 \land m_{3,4} \land t=2) \supset \Diamond \ell_5$ by EVNT with $k=1$, using lemma A
2. $\vdash (\ell_3 \land m_{3,4} \land t=2) \supset \Diamond(\ell_4 \land m_{3,4} \land t=2)$ by EVNT with $k=2$, using lemmas A, B
3. $\vdash (\ell_3 \land m_{3,4} \land t=2) \supset \Diamond \ell_5$ by 2, 1 and $\Diamond$C
4. $\vdash (\ell_{3,4} \land m_{3,4} \land t=2) \supset \Diamond \ell_5$ by 1, 3 and PR
5. $\vdash (\ell_{3,4} \land m_2) \supset \Diamond[\ell_5 \lor (\ell_{3,4} \land m_{3,4} \land t=2)]$ by EVNT with $k=2$

6. $\vdash (\ell_{3,4} \wedge m_2) \supset \Diamond \ell_5$ — by 4, 5 and PTR
7. $\vdash (\ell_{3,4} \wedge m_1) \supset \Diamond[\ell_5 \vee (\ell_{3,4} \wedge m_2)]$ — by EVNT with $k = 2$
8. $\vdash (\ell_{3,4} \wedge m_1) \supset \Diamond \ell_5$ — by 7, 6 and PTR
9. $\vdash (\ell_3 \wedge m_0) \supset \Diamond[\ell_5 \vee (\ell_{3,4} \wedge m_1)]$ — by EVNT with $k = 1$
10. $\vdash (\ell_3 \wedge m_0) \supset \Diamond \ell_5$ — by 9, 8 and PTR
11. $\vdash (\ell_4 \wedge m_0) \supset \Diamond[\ell_5 \vee (\ell_{3,4} \wedge m_1) \vee (\ell_3 \wedge m_0)]$ — by EVNT with $k = 1$
12. $\vdash (\ell_4 \wedge m_0) \supset \Diamond \ell_5$ — by 11, 8, 10 and PTR
13. $\vdash (\ell_{3,4} \wedge m_0) \supset \Diamond \ell_5$ — by 10, 12 and PR
14. $\vdash (\ell_{3,4} \wedge m_7) \supset \Diamond[\ell_5 \vee (\ell_{3,4} \wedge m_0)]$ — by EVNT with $k = 2$
15. $\vdash (\ell_{3,4} \wedge m_7) \supset \Diamond \ell_5$ — by 14, 13 and PTR
16. $\vdash (\ell_{3,4} \wedge m_6) \supset \Diamond(\ell_{3,4} \wedge m_7)$ — by EVNT with $k = 2$ and lemma E
17. $\vdash (\ell_{3,4} \wedge m_6) \supset \Diamond \ell_5$ — by 16, 15 and PTR
18. $\vdash (\ell_{3,4} \wedge m_5) \supset \Diamond(\ell_{3,4} \wedge m_6)$ — by EVNT with $k = 2$ and lemma E
19. $\vdash (\ell_{3,4} \wedge m_5) \supset \Diamond \ell_5$ — by 18, 17 and PTR
20. $\vdash (\ell_{3,4} \wedge m_4 \wedge t = 1) \supset \Diamond(\ell_{3,4} \wedge m_5)$ — by EVNT with $k = 2$ and lemma A
21. $\vdash (\ell_{3,4} \wedge m_4 \wedge t = 1) \supset \Diamond \ell_5$ — by 20, 19 and PTR
22. $\vdash (\ell_{3,4} \wedge m_3 \wedge t = 1) \supset \Diamond(\ell_{3,4} \wedge m_4 \wedge t = 1)$ — by EVNT with $k = 2$ and lemma A
23. $\vdash (\ell_{3,4} \wedge m_3 \wedge t = 1) \supset \Diamond \ell_5$ — by 22, 21 and PTR
24. $\vdash (\ell_{3,4} \wedge m_{3,4} \wedge t = 1) \supset \Diamond \ell_5$ — by 21, 23 and PR
25. $\vdash (\ell_{3,4} \wedge m_{3,4}) \supset \Diamond \ell_5$ — by 4, 24, lemma C and PR

We may summarize now as follows:

26. $\vdash \ell_{3,4} \supset [\ell_{3,4} \wedge (m_0 \vee m_1 \vee m_2 \vee m_3 \vee m_4 \vee m_5 \vee m_6 \vee m_7)]$ — by LOC
27. $\vdash \ell_{3,4} \supset \Diamond \ell_5$ — by 26, 13, 8, 6, 25, 19, 17, 15 and PTR
28. $\vdash \ell_2 \supset \Diamond \ell_{3,4}$ — by EVNT with $k = 1$
29. $\vdash \ell_2 \supset \Diamond \ell_5$ — by 27, 28 and $\Diamond$C
30. $\vdash \ell_1 \supset \Diamond \ell_2$ — by EVNT with $k = 1$
31. $\vdash \ell_1 \supset \Diamond \ell_5$ — by 29, 30 and $\Diamond$C ∎

F. COMPACT PROOF PRINCIPLES

In the preceding sections we introduced a comprehensive proof system for proving arbitrary temporal properties of concurrent programs. However, as demonstrated in the last examples a fully formal proof tends to be rather lengthy and sometimes tedious to follow. Consequently we will next discuss shorter and more compact representations of proofs and corresponding compact proof principles. All these principles can be derived in the basic proof system presented above. Consequently, a proof according to these principles can always be mechanically expanded into a more detailed proof using just the basic axioms. We will discuss the three main classes of properties one may wish to prove about programs, namely: invariance, liveness and precedence properties.

13. THE INVARIANCE PRINCIPLE

The IINV principle does not significantly simplify formal proofs. Most of the needed work in applying the IINV principle is in establishing the premise that the program P leads from φ to φ. Several heuristics or meta-rules can be suggested in order to reduce the number of transitions that have to be checked, which in the worst case is proportional to the size of the program. For example:

a) Only transitions that modify variables on which φ depends should be checked.

b) Assume that φ has the form $\varphi = \varphi_1 \vee \varphi_2$ (similarly for implication), and that some variables $y_1, \ldots, y_m$ appear only in φ_1. Then, in checking transitions that only modify these variables, it is sufficient to check transitions that may falsify φ_1 and one may assume in checking them that $\varphi_2 = \mathit{false}$.

c) Assume that an invariance χ has already been established before. Let

$$[\varphi \wedge \chi] \supset (\sim \mathit{at}\,\ell)$$

for some location ℓ. Then no transitions of the form $\ell \to \ell'$ need ever be considered in showing that P leads from φ to ψ.

A simple generalization of the IINV rule is given by:

Generalized Invariance Rule — GINV

A. $\vdash \varphi \supset \psi$

B. $\vdash [\mathit{at}\,\bar{\ell}_0 \wedge \bar{y} = g(\bar{x})] \supset \varphi$

C. $\vdash P$ leads from φ to φ

$$\frac{}{\vdash \Box \psi}$$

Certainly premises B and C establish $\vdash \Box \varphi$ according to IINV, from which by premise A and the $\Box\Box$ rule, $\vdash \Box \psi$ follows.

The advantage of the GINV principle is that no additional temporal reasoning is required and the rule can be proved complete by itself. By this we mean that, given a program P, any state property ψ which is invariant for all executions of P can be proven invariant by a single application of the GINV rule and no additional temporal reasoning.

Theorem:

The GINV rule is complete for proving invariance properties.

Proof:

Let $\psi = \psi(\overline{x};\overline{\pi};\overline{y})$ be a state property, possibly dependent on the input variables $\overline{x}$. We define a state $s = \langle \overline{\ell};\overline{\eta}\rangle$ to be $\overline{\xi}$-accessible in P if there exists a segment of some computation initialized with $\overline{x} = \overline{\xi}$ that reaches s, i.e.,

$$\langle \overline{\ell}_0; g(\overline{\xi})\rangle \;\rightarrow\; \ldots \;\rightarrow\; \langle \overline{\ell};\overline{\eta}\rangle.$$

Define the predicate $\varphi = \varphi(\overline{x};\overline{\pi};\overline{y})$ by:

$$\varphi(\overline{\xi};\overline{\ell};\overline{\eta}) = \mathit{true} \;\Leftrightarrow\; \langle \overline{\ell};\overline{\eta}\rangle \text{ is } \overline{\xi}\text{-accessible.}$$

Thus, φ characterizes all the states that are $\overline{x}$-accessible. We will show that the predicate φ so defined satisfies, together with ψ, all the premises required by the rule GINV.

Consider premise A. Since ψ is invariantly true in all computations of P it must be true for every accessible state $\langle \overline{\ell};\overline{\eta}\rangle$. Consequently

$$\varphi(\overline{\xi};\overline{\ell};\overline{\eta}) \;\supset\; \psi(\overline{\xi};\overline{\ell};\overline{\eta});$$

when generalized to arbitrary $\overline{\xi}$, $\overline{\ell}$ and $\overline{\eta}$ this implies

$$\models \varphi \;\supset\; \psi.$$

Since we assume that the underlying domain theory is adequate for proving all classically sound formulas this implies

$$\vdash \varphi \;\supset\; \psi.$$

Consider now premise B. Since every initial state is by definition accessible we certainly have

$$\models \varphi(\overline{x};\overline{\ell}_0;g(\overline{x})).$$

Again by completeness of our domain part with respect to classical formulas, this leads to

$$\vdash [\mathit{at}\,\overline{\ell}_0 \;\wedge\; y = g(\overline{x})] \;\supset\; \varphi(\overline{x};\overline{\pi};\overline{y}).$$

Finally, consider premise C. Clearly every transition in P leads from an $\overline{x}$-accessible state to another $\overline{x}$-accessible state. Consequently

$$\models P \text{ leads from } \varphi \text{ to } \varphi.$$

From this premise C follows by completeness of the domain part. ∎

In the preceding theorem we have only shown the existence of an appropriate state predicate φ. We have not discussed the question of the exact formal language in which such a predicate can be expressed. However, assuming that our domain contains the integers or some isomorphic structure, and using a first-order language, it is not difficult to show that the statement:

> "There exists a finite computation of P leading from $\langle \bar{\ell}_0; g(\bar{\xi}) \rangle$ to $\langle \bar{\ell}; \bar{\eta} \rangle$"

can be Gödel-encoded into a first-order statement over the integers.

14. LIVENESS PRINCIPLES

As a typical example of a detailed proof of liveness properties we may reexamine the proof of accessibility for the mutual exclusion program (Example 3). The structure of such a proof proceeds through a chain of events characterized by state assertions. Let the eventuality to be proved be $\varphi \supset \Diamond \psi$ where both φ and ψ are state properties. We may regard $\psi = \varphi_0$ as being the last assertion in the chain. Then we identify an assertion φ_1 such that by a single application of the EVNT principle we can prove

$$\vdash \varphi_1 \supset \Diamond \psi.$$

In the example considered we have

$$\psi : \ell_5$$

$$\varphi_1 : \ell_4 \wedge m_{3,4} \wedge (t = 2).$$

Next, we identify an assertion φ_2 such that by a single application of the EVNT principle we can prove

$$\vdash \varphi_2 \supset \Diamond(\varphi_1 \vee \psi).$$

In the general step, we identify an assertion φ_i such that by a single application of the EVNT principle we can prove

$$\vdash \varphi_i \supset \Diamond(\bigvee_{j<i} \varphi_j).$$

Finally we have to prove $\varphi \supset (\bigvee_{i=0}^{r} \varphi_i)$ where $\varphi_0, \varphi_1, \ldots, \varphi_r$ is the chain of assertions constructed. We may summarize this proof pattern by the following proof principle:

The hain Reasoning Proof Principle — CHAIN

Let $\varphi_0, \varphi_1, \ldots, \varphi_r$ be a sequence of state properties satisfying the following requirements:

A. $\vdash P$ leads from φ_i to $\bigvee_{j \leq i} \varphi_j$ for $i = 1, \ldots, r$.

B. For every $i > 0$ there exists a $k = k_i$ such that:

$$\vdash P_k \text{ leads from } \varphi_i \text{ to } \bigvee_{j<i} \varphi_j$$

C. For $i > 0$ and $k = k_i$ as above:

$$\vdash \varphi_i \supset \Diamond[(\bigvee_{j<i} \varphi_j) \vee Enabled(P_k)]$$

$$\vdash (\bigvee_{i=0}^{r} \varphi_i) \supset (\bigvee_{i=1}^{r} \varphi_i) \mathcal{U} \varphi_0$$

Proof:

To justify this principle we will prove by induction on n, $n = 0, 1, \ldots, r$, that

$$\vdash (\bigvee_{i=0}^{n} \varphi_i) \supset (\bigvee_{i=1}^{n} \varphi_i) \mathcal{U} \varphi_0.$$

For $n = 0$ we have $\vdash \varphi_0 \supset \varphi_0$ from which trivially follows by axiom A9

$$\vdash \varphi_0 \supset (false\ \mathcal{U}\ \varphi_0).$$

Note that we interpret an empty disjunction as *false*.

We assume that the statement above has been proved for certain n and we attempt to prove it for $n + 1$.

Consider the EVNT rule with $\varphi = \varphi_{n+1}$, $\psi = (\bigvee_{i=0}^{n} \varphi_i)$. By premise A of CHAIN we obtain that P leads from $\varphi_{n+1} = \varphi$ to

$$(\bigvee_{j \leq n+1} \varphi_j) = (\varphi_{n+1} \vee (\bigvee_{j \leq n} \varphi_j)) = (\varphi \vee \psi).$$

This provides premise A of EVNT. Let $k = k_{n+1}$. Then by premise B of CHAIN, P_k leads from $\varphi_{n+1} = \varphi$ to $(\bigvee_{j<n+1} \varphi_j) = \psi$. Similarly, premise C of CHAIN yields that

1. $\vdash \varphi \supset \Diamond(\psi \vee Enabled(P_k))$.

By the EVNT rule it follows that

$$2.\quad \vdash \varphi \supset \varphi \mathcal{U} \psi$$

or

$$3.\quad \vdash \varphi_{n+1} \supset \varphi_{n+1} \mathcal{U} \Big(\bigvee_{i=0}^{n} \varphi_i \Big).$$

By the induction hypothesis and the $\mathcal{U}\mathcal{U}$ rule this yields

$$4.\quad \vdash \varphi_{n+1} \supset \varphi_{n+1} \mathcal{U} \Big(\Big(\bigvee_{i=1}^{n} \varphi_i \Big) \mathcal{U} \varphi_0 \Big).$$

Again by the induction hypothesis using part of A9, $w_2 \supset w_1 \mathcal{U} w_2$, we can obtain

$$5.\quad \vdash \Big(\bigvee_{i=0}^{n} \varphi_i \Big) \supset \varphi_{n+1} \mathcal{U} \Big(\Big(\bigvee_{i=1}^{n} \varphi_i \Big) \mathcal{U} \varphi_0 \Big).$$

Combining this with 4 above yields

$$6.\quad \vdash \Big(\bigvee_{i=0}^{n+1} \varphi_i \Big) \supset \varphi_{n+1} \mathcal{U} \Big(\Big(\bigvee_{i=1}^{n} \varphi_i \Big) \mathcal{U} \varphi_0 \Big).$$

By T38, $p \mathcal{U} (q \mathcal{U} r) \supset (p \vee q) \mathcal{U} r$, we can reduce the nesting depth of the $\mathcal{U}$ operator to get:

$$7.\quad \vdash \Big(\bigvee_{i=0}^{n+1} \varphi_i \Big) \supset \Big(\Big(\bigvee_{i=1}^{n+1} \varphi_i \Big) \mathcal{U} \varphi_0 \Big)$$

as needed.

Taking $n = r$ concludes the proof of the principle. ∎

In presenting a proof according to the chain-reasoning principle it is usually sufficient to identify $\varphi_0, \varphi_1, \ldots, \varphi_r$ and for each i to point out the "helpful" process $P_k = P_{k_i}$. It can be left to the reader to verify that premises A to C are satisfied for each $i = 1, 2, \ldots, r$.

We prefer to present such proofs in the form of a diagram. Consider a diagram consisting of nodes that correspond to the assertions $\varphi_0, \varphi_1, \ldots, \varphi_r$. For each transition affected by some process P_j, that leads from a state s satisfying φ_i to a state s' satisfying φ_ℓ, $\ell < i$, we draw an edge from the node φ_i to the node φ_ℓ and label it by P_j, the name of the responsible process. All edges corresponding to the helpful process $P_k = P_{k_i}$ are drawn as double arrows. We do not explicitly draw edges corresponding to transitions from φ_i back to itself. However it is assumed that such edges may exist for all but the helpful process for φ_i.

As an example we present a diagram form of the proof of accessibility for the Mutual Exclusion program. It is given in Fig. 1. In constructing such a proof we may freely use any invariants previously derived.

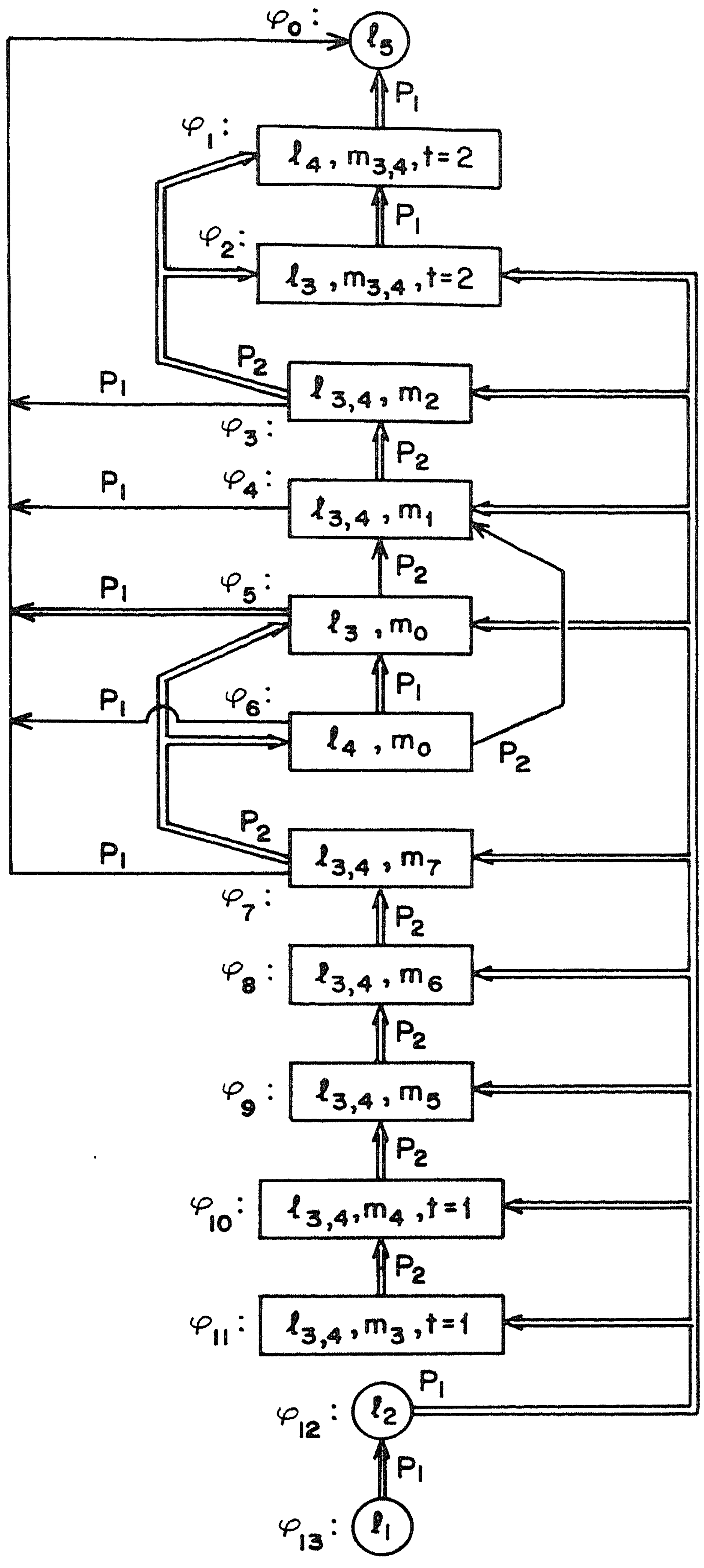


Fig. 1. Proof Diagram for the Mutual Exclusion Program

In this program, and typically in all non-terminating programs that have no semaphore instructions, we do not have to check premise C of the CHAIN or EVNT rule. This is because in non-terminating programs without semaphores every process is continuously enabled and therefore condition C is automatically satisfied.

In contrast let us consider the proof of accessibility for example 2 – a program with semaphores. Here we want to prove $\ell_0 \supset \diamond \ell_1$. The main diagram here is very simple:

$$\boxed{\ell_0} \xRightarrow{P_1} \boxed{\ell_1}$$

It denotes a single application of the EVNT rule with $\varphi : at\,\ell_0$ and $\psi : at\,\ell_1$ with $P_k = P_1$ being the helpful process.

However, in order to justify premise C, which is not trivial in this case, we have to prove

$$\vdash \ell_0 \supset \diamond(\ell_1 \vee y > 0).$$

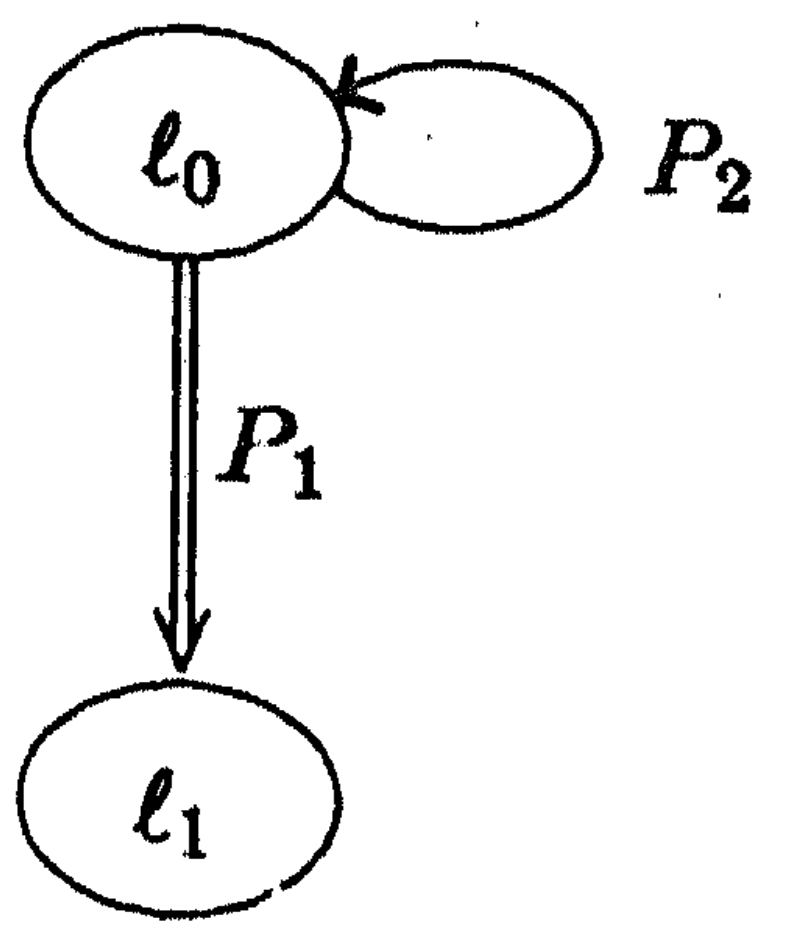
For this we have to consider P_2's position. If P_2 is at m_0 or m_2 then $y = 1$ by the invariant I_1 proved above. The only other case is when P_2 is at m_1 where by a single application of the EVNT rule it will eventually move to m_2 producing a positive value of y. This may be represented by a secondary diagram:

$$\boxed{\ell_0, m_1} \xRightarrow{P_2} \boxed{\ell_0, y > 0}$$

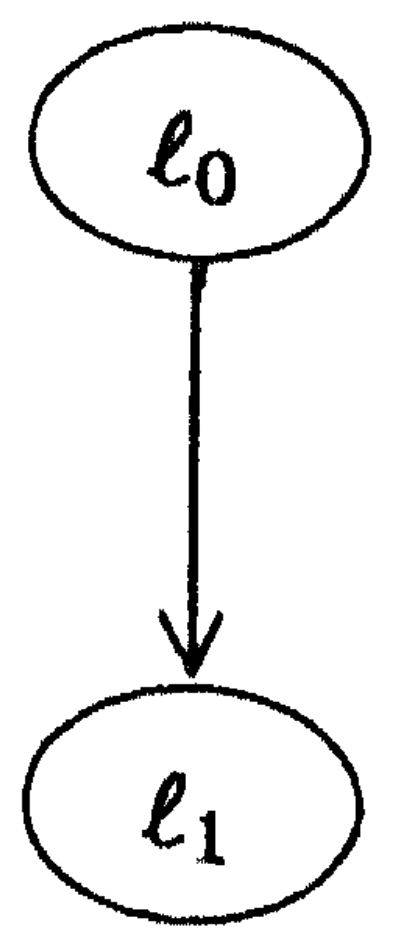
The diagram representation of a proof according to the CHAIN principle is very similar to the proof lattices introduced in [OL] as a concise presentation of a proof of a liveness property. A superficial difference is that they choose to represent as edges the *consequences* of the EVNT rule, while in our representation edges stand for the premises of the EVNT rule which are also the premises to the CHAIN rule. To illustrate this difference, consider the following trivial program:

$\ell_0 :\ y := y$ $\ell_1 :$	$m_0 :\ go\ to\ m_0$
$- P_1 -$	$- P_2 -$

The liveness property to be proved is $\ell_0 \supset \diamond L_1$. Below are diagram representations of the CHAIN principle and a proof lattice according to [OL].



CHAIN Diagram



Proof Lattice

As we see, the CHAIN diagram contains a self-edge, labelled by P_2 (this time drawn explicitly). and a helpful edge labelled by P_1. The process P_1 is guaranteed to get us to ℓ_1. As a consequence of this, by the EVNT rule, $\ell_0 \supset \Diamond \ell_1$. This conclusion is represented in the proof lattice by a single edge from ℓ_0 to ℓ_1. Thus, the different choices of representation lead to the following minor syntactical differences between CHAIN diagrams and proof lattices:

(a) Proof lattices are acyclic, whereas CHAIN diagrams are only weakly acyclic, i.e., may contain self-loops.

(b) In CHAIN diagrams, edges are labelled by the processes responsible for the transition. Special identification is provided for edges traversed by the helpful process. In proof lattices, we no longer care about the identities of the processes since progress along the lattice has already been established.

However these differences are minor and a simple procedure for translation between CHAIN diagrams and proof lattices exists. The important part in both is the identification of the intermediate assertions that are represented as nodes. In constructing a proof, this is usually the creative and most demanding process. Both graph presentations provide a natural and intuitive representation of these assertions and the precedence relations between them.

The chain-reasoning principle assumed a finite number of links in the chain. It is quite adequate for finite-state programs, i.e., programs whose variables range over finite domains. However, once we consider programs over the integers it is no longer sufficient to consider only finitely many assertions. In fact, sets of assertions of quite high cardinality are needed. The obvious generalization of a finite set of assertions $\{\varphi_i \mid i = 0, \ldots, r\}$ is to consider a single assertion $\varphi(\alpha)$, parametrized by a parameter α taken from a well-founded ordered set $(\mathcal{A}, \prec)$. Obviously, the most important property of our chain of assertion is that program transitions eventually lead from φ_i to φ_j with $j < i$. This property can also be stated for an arbitrary well-founded ordering. Thus a natural generalization of the chain reasoning rule is the following:

The Well Founded Liveness Principle — WELL

Let $(\mathcal{A}, \prec)$ be a *well-founded set*. Let $\varphi(\alpha) = \varphi(\alpha; \overline{x}; \overline{\pi}; \overline{y})$ be a parametrized state formula. Let $h : \mathcal{A} \rightarrow [1 \,..\, k]$ be a helpfulness function identifying for each $\alpha \in \mathcal{A}$ the helpful process $P_{h(\alpha)}$ for states in $\varphi(\alpha)$.

$$\begin{array}{ll} \text{A.} & \vdash P \text{ leads from } \varphi(\alpha) \text{ to } \psi \vee (\exists \beta \preceq \alpha \,.\, \varphi(\beta)) \\ \text{B.} & \vdash P_{h(\alpha)} \text{ leads from } \varphi(\alpha) \text{ to } \psi \vee (\exists \beta \prec \alpha \,.\, \varphi(\beta)) \\ \text{C.} & \vdash \varphi(\alpha) \supset \Diamond[\psi \vee (\exists \beta \prec \alpha \,.\, \varphi(\beta)) \vee \mathit{Enabled}(P_{h(\alpha)})] \\ \hline & \vdash (\exists \alpha \,.\, \varphi(\alpha)) \supset (\exists \alpha \,.\, \varphi(\alpha)) \,\mathcal{U}\, \psi \end{array}$$

A justification of this rule can again be conducted, based on induction. Now, however, induction over arbitrary well-founded sets is required.

15. EXAMPLE 4: BINOMIAL COEFFICIENT

As an example for the application of the WELL principle, we consider the following program that computes the binomial coefficient $\binom{n}{k}$ for inputs $0 \leq k \leq n$.

```
                  $(y_1, y_2, y_3, y_4) := (n, 0, 1, 1)$

$\ell_7$ : if $y_1 = (n-k)$ then go to $\ell_1$      $m_3$ : if $y_2 = k$ then go to $m_1$
$\ell_6$ : request$(y_4)$                            $m_2$ : $y_2 := y_2 + 1$
$\ell_5$ : $t_1 := y_3 \cdot y_1$                    $m_9$ : loop until $y_1 + y_2 \leq n$
$\ell_4$ : $y_3 := t_1$                              $m_8$ : request$(y_4)$
$\ell_3$ : release$(y_4)$                            $m_7$ : $t_2 := y_3 / y_2$
$\ell_2$ : $y_1 := y_1 - 1$                          $m_6$ : $y_3 := t_2$
$\ell_8$ : go to $\ell_7$                            $m_5$ : release$(y_4)$
$\ell_1$ : halt                                      $m_4$ : go to $m_3$
                                                     $m_1$ : halt

        $- P_1 -$                                            $- P_2 -$
```

The labelling scheme of the program has been constructed in a way that simplifies the expression of the assertion $\varphi(\alpha)$.

The computation of this program is based on the formula:

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdots (n-k+1)}{1 \cdot 2 \cdots k}.$$

The values of y_1, i.e., $n, n-1, \ldots, n-k+1$, are used to compute the numerator in P_1, and the values of y_2, i.e., $1, 2, \ldots, k$, are used to compute the denominator. The process P_1 multiplies $n \cdot (n-1) \cdots (n-k+1)$ into y_3 while P_2 divides y_3 by $1 \cdot 2 \cdots k$.

The instruction

$$m_9 : \textit{loop until } y_1 + y_2 \leq n$$

guarantees even divisibility of y_3 by y_2. It synchronizes P_2's operation with that of P_1 to ensure that y_3 is divided by i only after $(n-i+1)$ has already been multiplied into it. We rely here on the mathematical theorem that the product of i consecutive integers $n \cdot (n-1) \cdots (n-i+1)$ is always divisible by $i!$ (the quotient actually being the integer $\binom{n}{i}$).

The critical sections $\ell_{3..5}$ and $m_{5..7}$ are mutually protected by the semaphore variable y_4. This protection ensures that y_3 is not updated by P_2 between, say, the computation of $y_3 \cdot y_1$ and the assignment of this value to y_3. Without this protection, the updated value might have been overwritten by P_1.

We start by establishing some invariant properties of this program.

$$I_1: \quad \vdash \ (at\,\ell_{3..5} + at\,m_{5..7} + y_4 = 1) \wedge (y_4 \geq 0).$$

This is the usual semaphore invariant. It can be proven by observing that initially this sum equals 1, and then by considering all possible transitions. For example, the $\ell_6 \rightarrow \ell_5$ transition changes $at\,\ell_{3..5}$ from 0 (*false*) to 1 (*true*), and also decrements y_4 by 1, leaving however the sum constant. From I_1 we can deduce mutual exclusion of the critical sections, i.e.,

$$\vdash \ (\sim\ell_{3..5}) \vee (\sim m_{5..7}).$$

As a consequence of this we can establish:

$$I_2: \quad \vdash \ (\ell_4 \supset t_1 = y_3 \cdot y_1) \wedge (m_6 \supset t_2 = y_3/y_2).$$

This holds due to the impossibility of interference by P_2 while P_1 is at ℓ_4.

$$I_3: \quad \vdash \ (n - k + at\,\ell_{2..6}) \leq y_1 \leq n.$$

This invariance states that y_1 always lies between $n-k$ and n. When P_1 is at $\ell_{2..6}$, $y_1 > n-k$, whereas P_1 is at other locations, $y_1 \geq n-k$. To verify I_3 we need only consider the transitions:

- $\ell_7 \rightarrow \ell_6$ which maintains $n-k < y_1 \leq n$, assuming it was previously known that $n-k \leq y_1 \leq n$.
- $\ell_2 \rightarrow \ell_8$ which results in $n-k \leq y_1 - 1 \leq n$ from $n-k < y_1 \leq n$.

$$I_4: \quad \vdash \ 0 \leq y_2 \leq (k - at\,m_2).$$

This invariance bounds the range of y_2. We need consider the transitions $m_3 \rightarrow m_2$ and $m_2 \rightarrow m_4$ which can be shown to maintain I_4.

$$I_5: \quad \vdash \ at\,m_{7..8} \supset (y_1 + y_2) \leq n.$$

Here we should consider two transitions:

- $m_9 \rightarrow m_8$ which is possible only if currently $y_1 + y_2 \leq n$.
- $\ell_2 \rightarrow \ell_8$ is the only transition modifying y_1. However since it decrements y_1 it certainly preserves $y_1 + y_2 \leq n$.

Let us define the following virtual variables:

$$y_1^* = \textit{if } at\,\ell_{2,3} \textit{ then } y_1 - 1 \textit{ else } y_1$$

$$y_2^* = \textit{if } at\,m_{6..9} \textit{ then } y_2 - 1 \textit{ else } y_2$$

These variables are roughly equal to y_1 and y_2 respectively and differ from them by 1 in certain ranges.

$$I_6: \quad \vdash \; y_3 = [n \cdot (n-1) \cdots (y_1^* + 1)]/[1 \cdot 2 \cdots y_2^*].$$

To verify this invariant we have to check the transitions $\ell_4 \to \ell_3$, $m_6 \to m_5$. Making use of I_2, they can be shown to maintain I_6.

$$I_7: \quad \vdash \; [at\,\ell_1 \supset y_1 = (n-k)] \wedge [at\,m_1 \supset (y_2 = k)].$$

Using I_6, I_7 and the definition of y_1^*, y_2^* we obtain partial correctness of this program, namely

$$\vdash \; (at\,\ell_1 \wedge at\,m_1) \supset [y_3 = \tbinom{n}{k}].$$

To prove termination we will use the WELL rule in order to establish $\vdash \Diamond(at\,\ell_1 \wedge at\,m_1)$. As the well-founded domain we take

$$(\mathcal{A}, \prec) = (N \times N \times N, \prec_{lex}).$$

That is, the set of triplets of nonnegative integers ordered by lexicographic ordering. This ordering defines $(m_1, m_2, m_3) \prec (n_1, n_2, n_3)$ iff for the lowest i, $i = 1, 2, 3$ such that $m_i \neq n_i$, $m_i < n_i$.

For our goal assertion we take $\psi : at\,\ell_1 \wedge at\,m_1$. The parameterized assertion is given by:

$$\varphi(\alpha; \ell_i, m_j; y_1, y_2): \; (y_1 + k - y_2, \; j, \; i) \; = \; \alpha.$$

The helpfulness function is given by:

$$h(\alpha) \; = \; h(r, j, i) \; = \; (\mathit{if}\ i = 1\ \mathit{then}\ 2\ \mathit{else}\ 1).$$

Thus as long as the first process P_1 has not terminated we rely on P_1 to be the helpful process. Once it has terminated, we take P_2 to be the helpful process.

We have to show that all the three premises of the WELL rule are satisfied.

Consider first premise A. We have to show that every transition of P leads to $\varphi(\beta)$ with $\beta \preceq \alpha$ if ψ is not already satisfied. By simple inspection of all the possible transitions we find that they all lead from $\langle \ell_i, m_j \rangle$ to $\langle \ell_{i'}, m_{j'} \rangle$ such that either $i' < i$ or $j' < j$ except for the following transitions:

- $\ell_2 \to \ell_8$. But this transition decrements y_1 producing a strict decrease in $y_1 + k - y_2$ which is the first component in α.
- $m_2 \to m_9$. In a similar way this transition increments y_2, leading to a decrease in $y_1 + k - y_2$.
- $m_9 \to m_9$. This transition leaves α at the same value.

Consider now premise B. As we have shown above, all transitions provide a strict decrease in α. The only exception is $m_9 \to m_9$. However this is a P_2-transition which is considered helpful only when P_1 is at ℓ_1. By I_7, at this point $y_1 = (n-k)$ so that in view of I_4, $y_1 + y_2 \leq k$ and hence the only transition possible from m_9 is $m_9 \to m_8$.

To show premise C we have to prove that P_h is always eventually enabled. Consider first the case that $h = 1$. The only location in which it is not immediately enabled is when P_1 is at ℓ_6 while P_2 is at $m_{5..7}$ (in view of I_1). However by simple chain reasoning it is obvious that in such a case, P_2 will certainly reach m_4 in which y_4 becomes positive and P_1 enabled.

The case $h = 2$ is even simpler because it is only considered when P_1 is at ℓ_1. Consequently, even when P_2 is at m_8, which may potentially raise some problems, we have in view of I_1 and $at\,\ell_1$ that $y_4 > 0$ and P_2 is enabled.

Thus we conclude that $\psi : at\,\ell_1 \wedge at\,m_1$ must eventually be realized and therefore the program must terminate.

16. PRECEDENCE PROPERTIES

The next class of properties we will consider and provide proof principles for is that of precedence properties. These are properties, usually needing the $\mathcal{U}$ operator for their expression, which ensure that some event precedes another event, or that a certain event will not happen until another event happens first. In view of the fact that the basic FAIR and EVNT rules did actually provide a conclusion containing the $\mathcal{U}$ operator, they may be naturally utilized to form precedence proof principles which are generalizations of the corresponding liveness principles.

In the following we will often consider nested *until* expressions in which the nesting always occurs in the second argument. We therefore adopt the convention of representing the nested formula:

$$\varphi_n \,\mathcal{U} \left(\varphi_{n-1} \,\mathcal{U} \left(\ldots (\varphi_1 \,\mathcal{U}\, \varphi_0) \ldots \right)\right)$$

by:

$$\varphi_n \,\mathcal{U}\, \varphi_{n-1} \,\mathcal{U}\, \ldots \varphi_1 \,\mathcal{U}\, \varphi_0.$$

The semantic meaning of this formula is that, starting from the present there is going to be a period in which φ_n continuously holds, followed by another period in which φ_{n-1} continuously holds, ..., followed by a period in which φ_1 continuously holds, until finally φ_0 occurs. Any of these periods may be empty, but the occurrence of φ_0 is guaranteed.

Let us consider first the proper generalization of the CHAIN rule in which we assume a *finite* chain of assertions φ_r, φ_{r-1}, ..., φ_1 leading to the goal $\psi = \varphi_0$.

Let $0 < p_1 < p_2 < \ldots < p_s = r$ be a partition of the index range into s contiguous segments. Then we may formulate the following chain principle for precedence properties:

The Chain Rule for Precedence Properties — P-CHAIN

Let $\varphi_0, \varphi_1, \ldots, \varphi_r$ be a sequence of state assertions, and $0 = p_0 < p_1 < p_2 < \ldots < p_s = r$ a partition of $[1 \,..\, r]$.

A. $\vdash P$ leads from φ_i to $(\bigvee_{j \leq i} \varphi_j)$ for $i = 1, \ldots, r$.

B. For every $i > 0$ there exists a $k = k_i$ such that:

$$\vdash P_k \text{ leads from } \varphi_i \text{ to } (\bigvee_{j<i} \varphi_j)$$

C. For $i > 0$ and $k = k_i$ as above:

$$\vdash \varphi \supset \Diamond[(\bigvee_{j<i} \varphi_j) \vee \mathit{Enabled}(P_k)]$$

$$\vdash (\bigvee_{i=0}^{r} \varphi_i) \supset (\psi_s \,\mathcal{U}\, \psi_{s-1} \ldots \psi_1 \,\mathcal{U}\, \varphi_0)$$

where

$$\psi_\ell \text{ is } \bigvee_{p_{\ell-1} < j \leq p_\ell} \varphi_j \quad \text{for } \ell = 1, \ldots, s.$$

The conclusion states that starting at a state that satisfies one of the φ_i, $i = 0, \ldots, r$, we are guaranteed to have a period in which $(\bigvee_{j=p_{s-1}+1}^{p_s} \varphi_j)$ continuously holds, followed by a period in which $(\bigvee_{j=p_{s-2}+1}^{p_{s-1}} \varphi_j)$ continuously holds, etc., until φ_0 is finally realized. Any of these periods may be empty.

Proof:

To justify the soundness of this conclusion we will first prove it for the most refined partition possible, namely:

$$(\bigvee_{i=0}^{r} \varphi_i) \supset (\varphi_r \,\mathcal{U}\, \varphi_{r-1} \,\mathcal{U}\, \varphi_{r-2} \,\mathcal{U}\, \ldots \varphi_1 \,\mathcal{U}\, \varphi_0).$$

This is proved in a way similar to the justification of the corresponding liveness principle. We show, by induction on n, $n = 0, 1, \ldots, r$, that

$$\vdash (\bigvee_{i=0}^{n} \varphi_i) \supset (\varphi_n \,\mathcal{U}\, \varphi_{n-1} \,\mathcal{U}\, \ldots \varphi_1 \,\mathcal{U}\, \varphi_0).$$

For $n = 0$ we have $\vdash \varphi_0 \supset \varphi_0$ which is the induction statement for $n = 0$.

Assume that the statement above has been proved for a certain n and consider its proof for $n+1$.

Consider the EVNT rule with $\varphi = \varphi_{n+1}$, $\psi = (\bigvee_{i=0}^{n} \varphi_i)$. As shown in the proof of the liveness case, all the premises of the EVNT rule are satisfied. Consequently we may conclude:

$$\vdash \varphi_{n+1} \supset \varphi_{n+1} \,\mathcal{U}\, (\bigvee_{i=0}^{n} \varphi_i).$$

By the induction hypothesis and the $\mathcal{U}\mathcal{U}$ rule this yields

$$\vdash \varphi_{n+1} \supset \varphi_{n+1} \,\mathcal{U}\, (\varphi_n \,\mathcal{U}\, \ldots \varphi_1 \,\mathcal{U}\, \varphi_0).$$

Due to $\vdash v \supset (u \mathcal{U} v)$ which is a consequence of axiom A9, the induction hypothesis can also be written as

$$\vdash (\bigvee_{i=0}^{n} \varphi_i) \supset \varphi_{n+1} \,\mathcal{U}\, (\varphi_n \,\mathcal{U}\, \ldots \varphi_1 \,\mathcal{U}\, \varphi_0).$$

Taking the disjunction of the last two gives

$$\vdash (\bigvee_{i=0}^{n+1} \varphi_i) \supset \varphi_{n+1} \,\mathcal{U}\, (\varphi_n \,\mathcal{U}\, \ldots \varphi_1 \mathcal{U} \varphi_0),$$

which is the required statement for $n+1$.

Consider now a coarser partition:

$$0 = p_0 < p_1 < p_2 < \ldots < p_s = r.$$

By consecutively merging any two contiguous assertions that fall into the same partition cell, using theorem T38:

$$\vdash (\varphi_{i+1} \,\mathcal{U}\, (\varphi_i \,\mathcal{U}\, \varphi)) \supset ((\varphi_{i+1} \vee \varphi_i) \,\mathcal{U}\, \varphi),$$

we obtain the coarser conclusion:

$$\vdash (\bigvee_{i=0}^{n+1} \varphi_i) \supset \Big((\bigvee_{p_{s-1} < j \le p_s} \varphi_j) \,\mathcal{U}\, (\bigvee_{p_{s-2} < j \le p_{s-1}} \varphi_j) \,\mathcal{U}\, \ldots (\bigvee_{0 < j \le p_1} \varphi_j) \,\mathcal{U}\, \varphi_0\Big). \qquad \lrcorner$$

Examples:

As our first example, let us consider the Mutual Exclusion program analyzed above. We have already proven that mutual exclusion is maintained by this program. We have also proven the liveness property that if P_1 wishes to enter its critical section it will eventually gain access to it. A more discriminating question is that of how fair is our algorithm. That is, if P_1 wishes to enter its critical section, how many times will P_2 be able to enter its own critical section before P_1? Is that number bounded? We refer to this question as the problem of bounded overtaking. Namely, how many times can P_2 overtake P_1 before P_1 enters his critical section.

Our first analysis makes use of Fig. 1 without any modifications. We only read from it the stronger conclusion according to the stronger P-CHAIN rule. As a partition we choose $p_1 = 7$, $p_2 = 9$, $p_3 = r = 11$. Consequently, from the diagram of Fig. 1 we conclude by the P-CHAIN rule:

$$\vdash \ \Big(\bigvee_{i=1}^{11} \varphi_i\Big) \supset \Big(\Big(\bigvee_{i=10}^{11} \varphi_i\Big) \ \mathcal{U} \ \Big(\bigvee_{i=8}^{9} \varphi_i\Big) \ \mathcal{U} \ \Big(\bigvee_{i=1}^{7} \varphi_i\Big) \ \mathcal{U} \ \varphi_0\Big).$$

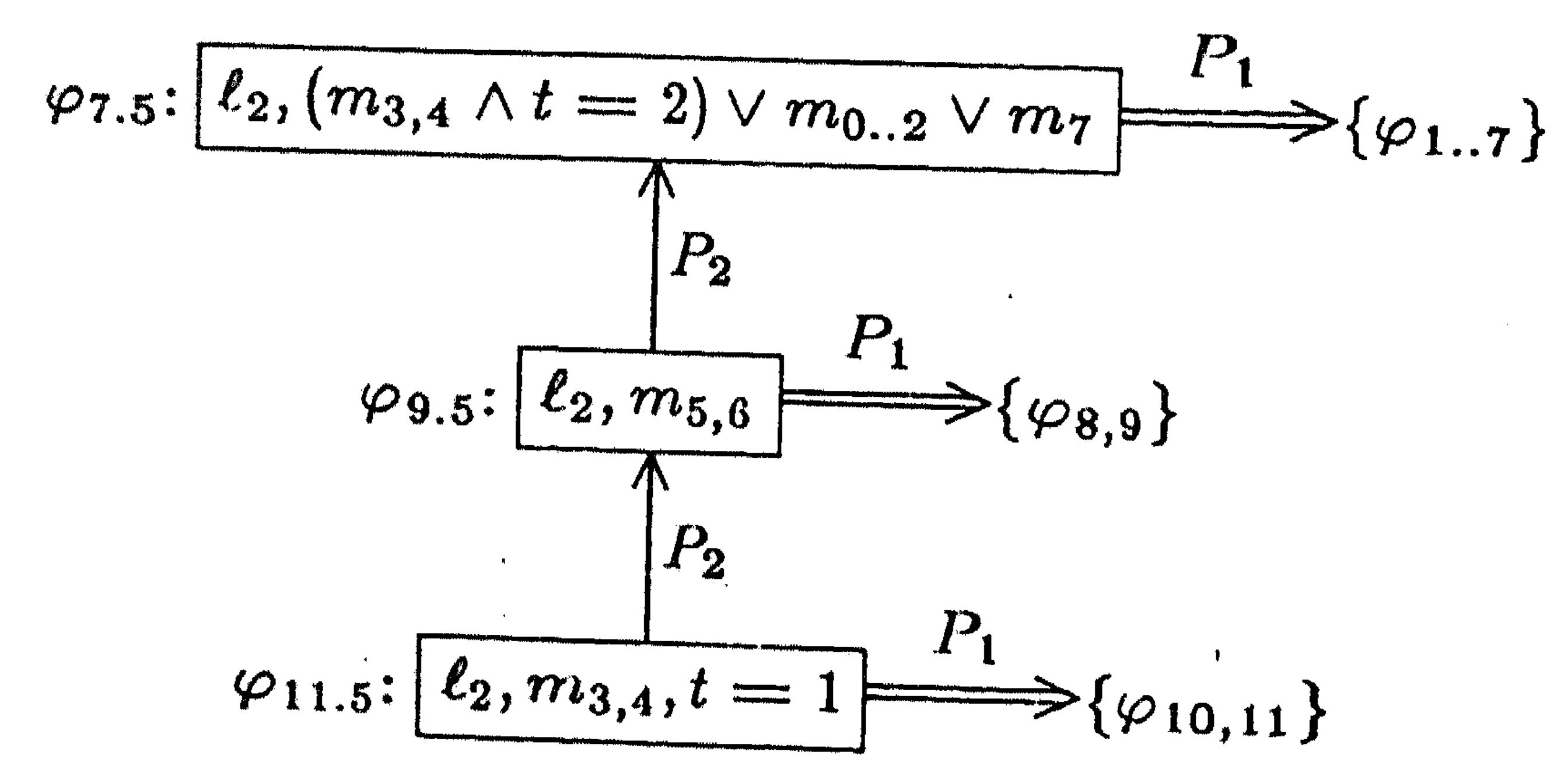
Replacing each of the right hand side disjunctions by a weaker property and the left hand side disjunction by a stronger statement we obtain:

$$\vdash \ \ell_{3,4} \supset \big((\sim m_{5,6}) \ \mathcal{U} \ m_{5,6} \ \mathcal{U} \ (\sim m_{5,6}) \ \mathcal{U} \ \ell_5\big).$$

This implies that if P_1 is at the waiting loop in $\ell_{3,4}$, there will be a period in which P_2 is not in the critical section $m_{5,6}$, followed by a period in which P_2 is inside the critical section $m_{5,6}$ followed by a period in which P_2 is outside the critical section which terminates by P_1 entering his critical section. Since any of these periods may be empty this is a worst-case analysis. But it certainly assures 1-bounded overtaking, i.e., once P_1 is in $\ell_{3,4}$, P_2 may overtake it at most once.

Having successfully analyzed the situation from $\ell_{3,4}$ on we may attempt to obtain a similar analysis from the moment that P_1 enters ℓ_2.

This analysis calls for a refinement of the diagram of Fig. 1. The following is a subdiagram that should replace the node corresponding to φ_{12} in Fig. 1. It consists of three nodes labelled respectively $\varphi_{7.5}$, $\varphi_{9.5}$ and $\varphi_{11.5}$. The fractional indexing indicates that $\varphi_{7.5}$ should be inserted between φ_7 and φ_8 in Fig. 1. The edges out of φ_{13} should enter one of these three nodes. Edges out of $\varphi_{7.5}$ lead to some of $\varphi_1, \ldots, \varphi_7$.



Similarly for edges out of $\varphi_{9.5}$ and $\varphi_{11.5}$. Considering the updated diagram composed of Fig. 1 and the above subdiagram we obtain the following conclusion:

$$\vdash \ \ell_{2..4} \supset \Big(\Big(\bigvee_{i=10}^{11.5} \varphi_i\Big) \ \mathcal{U} \ \Big(\bigvee_{i=8}^{9.5} \varphi_i\Big) \ \mathcal{U} \ \Big(\bigvee_{i=1}^{7.5} \varphi_i\Big) \ \mathcal{U} \ \varphi_0\Big).$$

This again leads to

$$\vdash \ \ell_{2..4} \ \supset \ ((\sim m_{5,6}) \ \mathcal{U} \ m_{5,6} \ \mathcal{U} \ (\sim m_{5,6}) \ \mathcal{U} \ \ell_5),$$

which ensures 1-bounded overtaking even from ℓ_2. Encouraged by this, we may next ask whether a similar result can be obtained from ℓ_1. Unfortunately this is not the case. P_2 may enter its critical section an arbitrary number of times while P_1 is at ℓ_1. This is obvious since while being at ℓ_1, P_1 has not yet modified any variable in a way that will show that it is not still in ℓ_0. And naturally while P_1 is at ℓ_0, P_2 may enter the critical section any number of times if the algorithm is correct.

THE WELL-FOUNDED PRINCIPLE FOR PRECEDENCE PROPERTIES

A natural extension of the P-CHAIN rule to programs that require infinite chains of assertions again uses well founded ordered sets.

Let $(\mathcal{A}, \prec)$ be a well founded ordered set. We require however that the ordering is total (or linear). That is, for every two distinct elements $\alpha_1, \alpha_2 \in \mathcal{A}$ either $\alpha_1 \prec \alpha_2$ or $\alpha_2 \prec \alpha_1$.

Well Founded Precedence Rule — P-WELL

Let $\varphi(\alpha) = \varphi(\alpha; \overline{\pi}; \overline{y})$ be a parametrized state assertion with $\alpha \in \mathcal{A}$.
Let $h : \mathcal{A} \rightarrow [1 \ .. \ k]$ be a helpfulness function.
Let $\alpha_1 \prec \alpha_2 \prec \ \ldots \ \prec \alpha_s$ be a sequence of elements of $\mathcal{A}$.

$\vdash P$ leads from $\varphi(\alpha)$ to $\psi \vee (\exists \beta \preceq \alpha \ . \ \varphi(\beta))$

$\vdash P_{h(\alpha)}$ leads from $\varphi(\alpha)$ to $\psi \vee (\exists \beta \prec \alpha \ . \ \varphi(\beta))$

$\vdash \varphi(\alpha) \ \supset \ \Diamond[\psi \ \vee \ (\exists \beta \prec \alpha \ . \ \varphi(\beta)) \ \vee \ \mathit{Enabled}(P_{h(\alpha)})]$

$\vdash \ (\exists \alpha \preceq \alpha_s \ . \ \varphi(\alpha)) \ \supset \ (\psi_s \ \mathcal{U} \ \psi_{s-1} \ \mathcal{U} \ \ldots \psi_1 \ \mathcal{U} \ \psi)$

where

ψ_ℓ is $\exists \beta (\alpha_{\ell-1} \prec \beta \preceq \alpha_\ell) \ . \ \varphi(\beta)$ for $\ell = 2, \ \ldots s,$ and

ψ_1 is $\exists \beta (\beta \preceq \alpha_1) \ . \ \varphi(\beta)$

Note that while the range of the parameter in the assertions is infinite, the partition is still finite.

Acknowledgement:

We thankfully acknowledge the help extended to us by Yoni Malachi, Ben Moszkowski, Stuart Russell, and Frank Yellin in reading the manuscript. Special thanks are due to Evelyn Eldridge-Diaz for TEXing the manuscript and to Carol Weintraub for typing its first draft.

REFERENCES

[H] Hoare, C.A.R., "Communicating Sequential Processes," CACM 21 (1978) pp. 666-677.

[ILL] Igarashi, S., London, R.L., Luckham, D.C., "Automatic Program Verification I: A Logical Basis and Its Implementation," *Acta Informatica*, Vol. 4, No. 2 (1975), pp. 145-182.

[KR] Kuiper, R. and de Roever, W.P. "Fairness Assumptions for CSP in a Temporal Logic Framework," TC2 Working Conference on the Formal Description of Programming Concepts, Garmisch (June 1982).

[L1] Lamport, L., "Proving the Correctness of Multiprocess Programs," IEEE Trans. Soft. Eng. SE-3, 2 (Mar. 1977), pp. 125-143.

[L2] Lamport, L., " 'Sometime' is Sometimes 'Not Never': On the Temporal Logic of Programs," 7th Annual ACM Symposium on Principles of Programming Languages (1980), pp. 174-185.

[LPS] Lehmann, D., A. Pnueli, and J. Stavi, "Impartiality, justice and fairness: the ethics of concurrent termination," in *Automata Languages and Programming*, Lecture Notes in Computer Science 115, Springer Verlag (1981), pp. 264-277.

[M] Manna, Z., "Verification of Sequential Programs: Temporal Axiomatization," Theoretical Foundations of Programming Methodology (M. Broy and G. Schmidt, eds.), NATO Scientific Series, D. Reidel Pub. Co., Holland (1982), pp. 53-102.

[MP1] Manna, Z. and A. Pnueli, "Verification of Concurrent Programs: The Temporal Framework," in *The Correctness Problem in Computer Science* (R.S. Boyer and J S. Moore, eds.), International Lecture Series in Computer Science, Academic Press, London (1982), pp. 215-273.

[MP2] Manna, Z. and A. Pnueli, "Verification of Concurrent Programs: Temporal Proof Principles," Proc. of the Workshop on Logic of Programs (D. Kozen, ed.), Yorktown-Heights, N.Y. (1981). Springer- Verlag Lecture Notes in Computer Science 131, pp. 200-252.

[MP3] Manna, Z. and A. Pnueli, "Verification of Concurrent Programs: Proving Eventualities by Well-Founded Ranking," TOPLAS (1983, to appear).

[MP4] Manna, Z. and A. Pnueli, "How to Cook a Temporal Proof System for Your Pet Language," in the Proc. of the Symposium on Principles of Programming Languages, Austin, Texas (Jan. 1983).

[MP5] Manna, Z. and Pnueli, A., "Verification of Concurrent Programs: a Temporal Proof System," Computer Science Report, Stanford University, Stanford, CA (1983).

[OL] Owicki, S. and L. Lamport, "Proving Liveness Properties of Concurrent Programs," ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3 (July 1982), pp. 455-495.

[Pe] Peterson, G.L., "Myths about the Mutual Exclusion Problem," Information Processing Letters, Vol. 12, No. 3 (June 1981), pp. 115-116.

[PS] Pnueli, A. and R. Sherman, "Semantic Tableau for Temporal Logic," Technical Report, CS81-21, The Weizmann Institute (Sept. 81).