# Towards a Proof of the Kahn Principle for Linear Dynamic Networks

Arie de Bruin & Shan-Hwei Nienhuys-Cheng
{arie, cheng}@cs.few.eur.nl
Department of Computer Science
Erasmus University Rotterdam
H4-19, P.O. Box 1738, 3000DR Rotterdam

**Abstract**

We consider dynamic Kahn-like data flow networks, i.e. networks consisting of deterministic processes each of which is able to expand into a subnetwork. The *Kahn principle* states that such networks are deterministic, i.e. that for each network we have that each execution provided with the same input delivers the same output. Moreover, the principle states that the output streams of such networks can be obtained as the smallest fixed point of a suitable operator derived from the network specification.
This paper is meant as a first step towards a proof of this principle. For a specific subclass of dynamic networks, linear arrays of processes, we define a transition system yielding an operational semantics which defines the meaning of a net as the set of all possible interleaved executions. We then prove that, although on the execution level there is much nondeterminism, this nondeterminism disappears when viewing the system as a transformation from an input stream to an output stream. This result is obtained from the *graph of all computations.* For any configuration such a graph can be constructed. All computation sequences that start from this configuration and that are generated by the operational semantics are embedded in it.

**Keywords** : the Kahn principle, dynamic data flow networks, process creation, the fork statement, operational semantics, nondeterministic transition systems

# 1 Introduction

A dataflow network consists of a number of parallel processes which are interconnected by directed channels. Processes communicate with each other only through these channels, there is no sharing of variables. The channels act as possibly infinite FIFO queues and communication is asynchronous.

In his seminal paper [K74] Kahn describes such networks in which the processes are deterministic. He characterizes the processes as functions, transforming input histories into output histories, where a history models a stream of values which has appeared on a channel during a computation. He then states a result which has become known as the *Kahn principle* since then: a network consisting of deterministic nodes as a whole also computes a history function, and for each set of input histories on the input channels the output histories can be obtained as the smallest solution of a set of equations derived from the network.

Stated somewhat differently: the nondeterminism caused by the asynchronicity of the computing processes does not lead to global nondeterminism in the history level I/O behaviour of the network. In essence there is only one computation possible, modeled by a function transforming input histories into output histories (although certain unfairly interleaved computations might not deliver the full output histories but only prefixes thereof).

Kahn's paper was quite influential, it has been the basis of much subsequent research. Work has been done to define evaluation strategies or implementations of dataflow networks, e.g. [KM77, AG78, F82], and several authors have proved the Kahn principle for certain subsets of networks [C72, A81, LS89].

Two extensions have been proposed to the framework sketched above. First of all the restriction can be omitted that the processes must be deterministic. In that case the Kahn principle is no longer true [BA81]. Much effort has been devoted to a study of this phenomenon and several remedies have been proposed [BA81, B88, J85, K86, K78, SN85, etc.], for an overview, cf. [JK90].

Another extension was already present in Kahn's original paper [K74], namely, to allow recursive definitions of history functions. This leads to a more intricate set of equations defining the system, in which not only variables occur denoting histories, but also variables denoting functions from histories to histories. In a subsequent paper [KM77], an idea is suggested to implement this, viz. reconfiguration or expansion: a node may be replaced by a subnetwork, connected to the rest of the network using the original channels. In [BB85] a simple programming language is presented using which expansions like these can be formulated, and a full denotational semantics for this language is given.

Although the Kahn principle has been justified for static networks, to our knowledge such a justification is lacking for the dynamic case, where it is possible that one process expands into a new network of processes. In this paper we take a first step to remedy this. We propose a simple language in which it is possible to create dynamically expanding linear arrays of processes, not unlike a unix pipeline which can be built up using the unix primitives pipe and fork. The meaning of a program in this language can be specified as a function from one input history to one output history.

In [B86] this language has been introduced and a denotational semantics has been defined, along the lines of [K74]. In [BBB93] an operational semantics has been given for this language. This semantics is based on the demand driven approach, it is deterministic and it formalizes the so called coroutine model proposed in [KM77]. In the same paper a proof is given of the equivalence of this operational semantics with the denotational one.

In this paper we introduce a transition system defining a full nondeterministic interleaving semantics. For this semantics we will prove the first half of the Kahn principle, i.e. that there is essentially one outcome possible for all computations (executions).

## 1.1 Overview of the proof method used in this paper

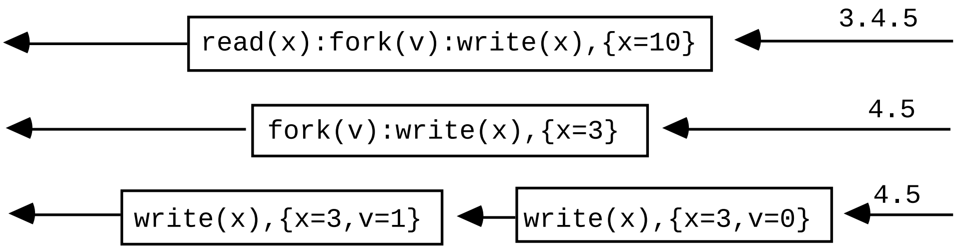
As we said earlier, in this article we consider the simple imperative language **L** also studied in [B86, BBB93]. For a given input stream and initial state, the execution of a program in **L** produces an output stream. Initially the program is executed by one so-called process, using precisely one input and one output channel. Execution of the statement read(x) will fetch the next value from the input channel and assign it to the variable x. Execution of the statement write(e) will evaluate the expression e and write the resulting value to the output channel.

However, a process can split up into two nearly identical subprocesses. One process, the mother, will have the input channel of the original process as its own input channel. The other process, the daughter, will be connected to the output channel of the original process. The output channel of the mother will be the input channel of the daughter. This channel will originally be empty. This effect is achieved by a statement of the form fork(v). Both processes proceed with execution of the statement following fork(v). There is one difference: in the mother process the variable v will be set to 0 and in the daughter v becomes 1.

Consider for example the program

read(x); fork(v); write(x)

to which an input stream with values 3, 4 and 5 is supplied. Execution of the first two statements of the program can be depicted as follows:




After the fork the system has evolved into a combination of two subprocesses. In the next step the leftmost one should write 3 to the global output channel while the rightmost one should write 3 to the newly created channel. We will model the parallelism by arbitrary interleaving, i.e., it is not determined which action should first be executed. This leads to nondeterminism which will be visible in our transition system. However, both executions will eventually lead to the same last stage as shown in the following figures.

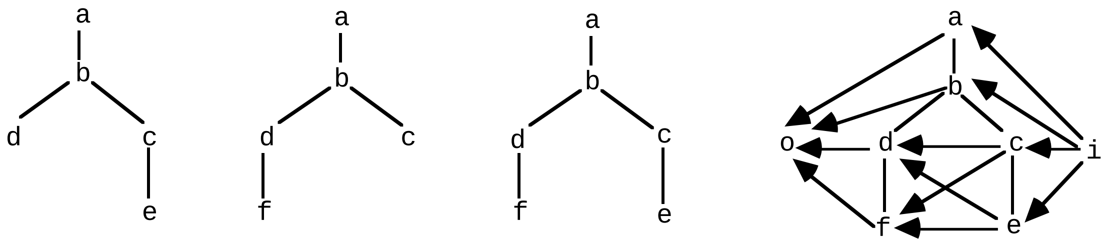



fig. A fig. B fig.C fig. D

The nodes a, b, c, etc. correspond with snapshots of the processes as given in the steps explained above. That is, node a corresponds with

`read(x):fork(v):write(x),{x=10}`

Node b corresponds with

`fork(v):write(x),{x=3}`

Nodes c and d correspond with the following snapshots of the two subprocesses, respectively.

`write(x),{x=3,v=1}` `write(x),{x=3,v=0}`

Fig. A symbolizes one execution in which the rightmost process writes first. Fig. B symbolizes another one in which the leftmost process writes first. However, both executions will arrive at a situation described in fig. C because the first execution will then take a transition from node d to node f and the second execution will take a transition from node c to node e. Thus the values written on the channels in these two executions will also be the same.

Therefore, one might say that fig. C gives an overview of all possible snapshots of subprocesses in executions of the program. If we add channels to this overview we obtain fig. D. Here, two nodes have been added, the input node i and the output node o. Node i is the starting point of the input channel into the whole system and node o is the end point of the output channel leaving the whole system. Now the resulting picture indeed provides an overview of all executions, because each path from i to o corresponds with a stage in an execution of the program, and all possible stages of all possible executions are captured in this way. For both executions we have that path i, a, o, path i, b, o and path i, c, d, o represent the first three stages. The first execution will arrive at stages described by path i, e, d, o and i, e, f, o. The second execution will arrive at the stages described by i, c, f, o and i, e, f, o. Thus the path i, f, e, o is the last stage of both executions.

Kahn's principle states that in general if we have a choice of actions executed by different subprocessesss, we can freely choose without effect on the output stream. This is intuitively clear but it is not so easy to prove it when we consider dynamically expanding networks. An execution can expand many subprocesses in one direction and neglect the other direction for some time wheras another execution might behave quite differently. This gives rise to quite different tree structures describing these executions. Even if we are able to prove that in the end trees of the same structure will have been obtained, we still cannot be sure whether corresponding nodes in these resulting trees will denote process snapshots with the same state (values of variables) and the same program still to be executed. If the states differ, then a write-statement might output different values on a channel. And if these different values are being input by a read-statement, then another process may obtain different states. So some care is needed when comparing two execution trees, especially where read and write statements are involved.

Our intention is to generalize graphs like the one given in fig. D. We want to systematically construct such a graph which contains all executions. A few questions which have to be answered are

- How should we construct a tree like the one in fig. C and how can it be proven that it represents the tree structure of all executions? The approach in this paper will be that we start to construct a graph corresponding with a special execution. This will be the backbone of the graph of all executions.
- How to construct all paths as given in fig. D which describe the stages of different executions?
- How to prove that all (fair) executions reach the same last path?

## 1.2 Overview of the paper

In the next section we fully define our language by giving its syntax and an operational semantics. In section 3 we will show how a computation graph can be defined for each computation as described by a transition sequence defined by transition system in section 2. After that we will define a special computation the corresponding graph of which will be used to construct the graph of all computations. This will be the first result of section 4. We then proceed by establishing a few properties of the horizontal paths (i.e. the paths from the input node to the output node) of this graph of all computations, one of which will be the one to one correspondence of these paths with the intermediate stages of all computations. This will enable us to show that there is essentially one maximal computation which proves the first half of Kahn's principle. The last section will state some conclusions and discuss work that is still ahead of us.

# 2 A Nondeterministic Transition System

## 2.1 Preliminaries

- Let **L** be a language which contains the following syntactical units:
  s ::= v:=e | skip | write(e) | read(v) | fork(v) | s ; s |
  if b then s else s fi | while b do s od
  Every unit s∈**L** is called a *statement*.

- Let (v∈)**Var**, (α, β, γ∈)**Val** , (e∈)**Exp** and (b∈)**Bexp** be given sets. They are usually called the set of *variables,* the set of *values*, the set of *expressions* and the set of *boolean expressions*, respectively. A *state* is a function $\sigma$: **Var** → **Val**. Let **State** be the set of all states. The notation $\sigma\{\beta/v\}$ is used to denote a state like $\sigma$ with the exception that $\sigma(v)=\beta$ now. Two meaning functions V : **Exp** → **State** → **Val** and B: **Bexp**→ **State** → {true, false} are assumed to be available.

- Let ($\eta$, $\zeta$, $\xi \in$)**Val**$^{\infty}$ be the set of finite or infinite sequences of elements from **Val**. Such a sequence is called a *stream*. We use $\varepsilon$ to denote the *empty stream*. For a finite stream $\xi=\alpha_1\ldots\alpha_m$ and a finite or infinite stream $\zeta=\beta_1\ldots\beta_n\ldots$, let $\xi.\zeta=\xi\zeta=\alpha_1\ldots\alpha_m\beta_1\ldots\beta_n\ldots$ . If $\xi$ is infinite, then $\xi\zeta=\xi$. We say $\xi$ is a *subsequence* or a *substream* of $\zeta$, denoted by $\xi\leq\zeta$ if there is a stream $\eta$ such that $\zeta=\xi\eta$. That means $\xi$ is a prefix of $\zeta$. We use $\xi\subset\zeta$ if $\xi$ is a postfix of $\zeta$, i.e., there is a finite stream $\eta$ such that $\zeta=\eta\xi$. We use *rest*($\xi$) to denote the stream $\xi$ without the first element.

- A *resumption* r is recursively defined by
  $$r ::= E \mid s : r,$$
  where $s\in$**L**. We use E to denote termination.
  **Config** is the set of all configurations where a *configuration* $\rho$ is recursively defined in the following way :
  $$\rho ::= \xi \mid \langle r, \sigma, \eta, \rho' \rangle,$$
  where $\xi, \eta \in$ **Val**$^{\infty}$ and $\rho'$ is a configuration.
  A configuration is thus a nested structure $\langle r, \sigma, \eta, \langle r', \sigma', \eta', \langle\ldots\rangle\ldots\rangle$. Here the r,$\sigma$-pairs model snapshots of processes (r consists of the statements still to be executed and $\sigma$ is the state). The $\eta$'s are called *buffer streams* modeling the values on the intermediate channels (i.e. written but not yet read). In the configuration $\langle r, \sigma, \eta, \langle r', \sigma', \eta', \langle\ldots\rangle\ldots\rangle$, $\eta'$ models the input channel of r',$\sigma'$-process and $\eta$ its output channel. If $\rho = \langle r, \sigma, \eta, \ldots\langle r', \sigma', \eta', \zeta'\rangle\ldots\rangle$, then $\zeta'$ is called the *input stream* of $\rho$.

- A *process* is a function P: **State** $\to$ **Val**$^{\infty}$ $\to$ **Val**$^{\infty}$. The set of all processes is denoted by **Proc**. In this paper we will define an *operational semantics* as a function O: **L** $\to$ **Proc**.

## 2.2 A Nondeterministic Transition Systems NT for L

A transition system is usually defined as a relation $\to$ on **Config**. i.e., $\to\subseteq$ **Config**x**Confi**g. In general, such a relation is not total, there may be configurations $\rho$ such that for every $\rho'$, $(\rho, \rho')\notin\to$. Such a $\rho$ is called a *terminal.* In this paper we will use a different approach. We introduce a special symbol $\otimes$. A terminal configuration $\rho$ will be characterized by the fact that $\otimes$ is its image. In that case we end up with a total relation $\to\subseteq$ **Config**x(**Config**$\cup\{\otimes\}$). We use $\rho \longrightarrow \rho'$ to denote $(\rho,\rho') \in \to$. If $\rho'\neq\otimes$, then it is called a *transition* from $\rho$ to $\rho'$. Such a transition may be accompanied by a label $\alpha\in$ **Val**. We then write $\rho \xrightarrow{\alpha} \rho'$. We define this relation by induction on the nesting depth of configurations. The proof that the relation is indeed total will be omitted. We also omit the proof that if the image of $\rho$ is $\otimes$, then it is indeed a terminal., i.e., no other images are possible.

(1) $\varepsilon \longrightarrow \otimes$
 If $\rho \longrightarrow \otimes$ the[redacted]$\langle E, \sigma, \eta, \rho\rangle \longrightarrow \otimes$
 If $\rho \longrightarrow \otimes$ t[redacted]$\langle read(v) : r, \sigma, \varepsilon, \rho\rangle \longrightarrow \otimes$
(2) If $\rho \xrightarrow{\alpha}$ $\rho$[redacted]n $\langle r, \sigma, \eta, \rho\rangle \longrightarrow \langle r, \sigma, \eta.\alpha, \rho'\rangle$
 If $\rho \longrightarrow$ $\rho$[redacted], then $\langle r, \sigma, \eta, \rho\rangle \longrightarrow \langle r, \sigma, \eta, \rho'\rangle$
(3) $\beta . \zeta' \xrightarrow{\beta} \zeta'$
(4) $\langle (v := e) : r, \sigma, \eta, \rho\rangle \longrightarrow \langle r, \sigma\{\beta / v\}, \eta, \rho\rangle$, where $\beta$=V(e)($\sigma$).
(5) $\langle skip : r, \sigma, \eta, \rho\rangle \longrightarrow \langle r, \sigma, \eta, \rho\rangle$
(6) $\langle write(e) : r, \sigma, \eta, \rho\rangle \xrightarrow{\beta} \langle r, \sigma, \eta, \rho\rangle$, where $\beta$=V(e)($\sigma$).
(7) $\langle read(v) : r, \sigma, \beta.\eta, \rho\rangle \longrightarrow \langle r, \sigma\{\beta / v\}, \eta, \rho\rangle$
(8) $\langle fork(v) : r, \sigma, \eta, \rho\rangle \longrightarrow \langle r, \sigma\{1 / v\}, \varepsilon, \langle r, \sigma\{0 / v\}, \eta, \rho\rangle\rangle$
(9) $\langle (s_1 ; s_2) : r, \sigma, \eta, \rho\rangle \longrightarrow \langle s_1 : (s_2 : r), \sigma, \eta, \rho\rangle$
(10) if B(b)($\sigma$)=true, then $\langle$ if b then $s_1$ else $s_2$ fi : r, $\sigma$, $\eta$, $\rho\rangle \longrightarrow \langle s_1 : r, \sigma, \eta, \rho\rangle$
 if B(b)($\sigma$)=false, then $\langle$ if b then $s_1$ else $s_2$ fi : r, $\sigma$, $\eta$, $\rho\rangle \longrightarrow \langle s_2 : r, \sigma, \eta, \rho\rangle$
(11) $\langle$while b do s od : r, $\sigma$, $\eta$, $\rho\rangle \longrightarrow \langle$if b then s; while b do s od else skip fi : r, $\sigma$, $\eta$, $\rho\rangle$

# 3 Graph of One Computation

Given a configuration, there may be different transition sequences starting from it. To find the relation between the streams of labels produced by these transition sequences, we first analyse how one such sequence is built up. To this end we introduce the notion *graph of one computation*.

## 3.1 Enabledness and computations

Given a configuration in **Config**,

$\rho = \langle r_m, \sigma_m, \eta_m, \ldots, \langle r_i, \sigma_i, \eta_i, \langle \ldots \langle \rho_1, \sigma_1, \eta_1, \zeta \rangle \ldots \rangle$,

for i=1, …, m, we define the i-th *subconfiguration* as

$\langle r_i, \sigma_i, \eta_i, \langle \ldots \langle r_1, \sigma_1, \eta_1, \zeta \rangle \ldots \rangle$.

Here $r_i$, $\sigma_i$ and $\eta_i$ are called the *resumption, state* and *buffer of* the i-th subconfiguration. It is clear from the definition of the transition rules, that a transition is determined by a transition on some i-th subconfiguration. That means that either the input stream of ρ undergoes a change (i=0) or $r_i \neq r_i'$ (i>0). In the transition

$\langle \ldots, \eta_{i+1}, \langle r_i, \sigma_i, \eta_i, \langle \ldots \langle r_1, \sigma_1, \eta_1, \zeta \rangle \ldots \rangle \longrightarrow \langle \ldots, \eta_{i+1}', \langle r_i', \sigma_i', \eta_i', \langle \ldots \langle r_1, \sigma_1, \eta_1, \zeta \rangle \ldots \rangle$,

we say this transition is *caused by* the i-th subconfiguration. We say also the transition is *caused by* the input stream or by the resumption $r_i$. The i-th subconfiguration can change only if it is enabled. We define

- ζ is *enabled* if it is not empty. $\langle r, \sigma, \eta, \varepsilon \rangle$ is *enabled* if one of the rules from (4) to (11) can be applied to it.
- The i-th subconfiguration of ρ given above is *enabled* if $\langle r_i, \sigma_i, \eta_i, \varepsilon \rangle$ is enabled.

If a subconfiguration is not enabled, then it is called *disabled*. It is clear that when a subconfiguration is enabled, then it performs a transition independently from the rest. For example, it can write without considering if the value written is needed. This gives a more uniform seamantics than *call by need* [BBB93]. This isalso the reason why buffer streams occur in configurations.

For a given configuration ρ, due to the nondeterminism there may be different maximal transition sequences possible from ρ. Each such sequence is called a *computation*.

$c(\rho): \quad \rho = \rho_0 \longrightarrow \rho_1 \ldots \longrightarrow \rho_{n-1} \longrightarrow \rho_n \longrightarrow \ldots$

The transition from $\rho_{n-1}$ to $\rho_n$ is called the n-th *step* of the computation. $\rho_n$ is called the n-th *stage* of the computation. The *output stream* of c(ρ) is the sequence of labels produced by c(ρ) .

Notice that a sequence can only be finite when there is no more transition possible after some stage. As we will prove later, the lengths of different computations from the same configuration are the same. However, if the length of these computations is infinite, then the output streams of different computations may not be the same. Consider for instance computations starting in

$\rho = \langle \mathrm{write}(1) : E, \sigma, \varepsilon, \langle \text{while true do skip}: E, \sigma, \varepsilon \rangle \rangle$

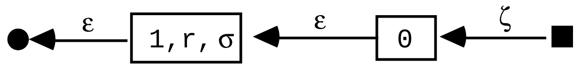
If a computation involves an execution of write(1), then the output stream will be nonempty. If every transition is caused by the first subconfiguration, then there will be no output at all. In this article we will also prove for the output streams of any two computations that always one is a substream of the other.

## 3.2 The graph of one computation

Given a configuration $\rho = \langle r, \sigma, \varepsilon, \zeta \rangle$ and a computation c(ρ), we will construct the computation graph of c(ρ). The nodes in such a graph are snapshots of processes corresponding to r,σ-pairs in configurations. Later when analyzing computation graphs it will be useful to have identification numbers for each node. For this purpose we will use a Dewey-like number system. Therefore nodes in our graphs will be p,r,σ-triples where p is the identification number of the node. (In the

sequel we will often identify nodes and their identification numbers). There will be two types of edges. Horizontal edges model channels between processes. They will be labeled by streams, having the same meaning as buffers $\eta$ in configurations. The other type of edge are vertical edges. These correspond with transition steps a process can take. If process p,r,$\sigma$-triple takes a transition, changing into p',r',$\sigma$', then there will be a vertical edge from node p,r,$\sigma$ to node p',r',$\sigma$'. The subsequent transitions starting from a subprocess will define a tree in the computation graph.
For any computation we have that the 0-th stage $\rho_0$ equals $\rho$. The stages in c($\rho$) are represented by *horizontal paths* consisting of horizontal edges. Notice that the following graph construction can be easily generalized to a computation starting in an arbitrary configuration.

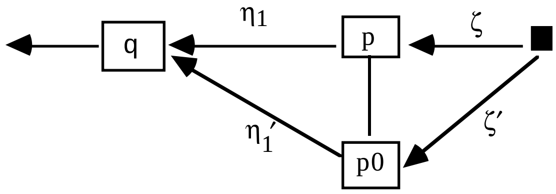
*Beginning of the graph – the 0-th step*




The solid black square symbolizes the input generator. The black circle represents the output receiver. These two nodes have identification numbers [-1] and [2] respectively. The interior nodes with identification number 0 and 1 are the roots of two trees to be constructed. In node 1 we also add r and $\sigma$. The *horizontal arrows* in this stage are denoted by <2,1>, <1,0> and <0,-1>. They have respectively $\varepsilon$, $\varepsilon$, $\zeta$ as labels. If there is no confusion we will write <q,p>=$\zeta$ to indicate that the arrow from p to q has $\zeta$ as label. For example, we have here <0,-1> = $\zeta$ and <1,0> = $\varepsilon$. The sequence of horizontal arrows <2,1>, <1,0>, <0,-1> is called the *horizontal path* of $\rho_0$.

*Extension of the graph – the (n+1)-th step*
Suppose the graph is already drawn up to the n-th stage $\rho_n$. Assume all horizontal paths representing $\rho_1,\dots,\rho_n$ are known. Let $\rho_n$=<$r_m$, $\sigma_m$,$\eta_m$,...<$r_i$, $\sigma_i$, $\eta_i$, ..., <$r_1$, $\sigma_1$, $\eta_1$, $\zeta$>...>.
We now define how to extend the graph. This is determined by the (n+1)-th step $\rho_n$ ——> $\rho_{n+1}$ from c($\rho$).

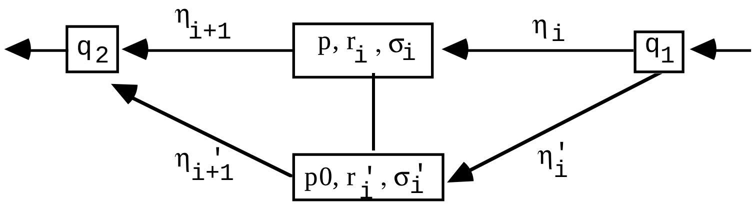
***a*** Suppose $\rho_n$ ——> $\rho_{n+1}$ is caused by $\zeta \xrightarrow{\beta} \zeta'$. Then
$\rho_{n+1}$=<$r_m$, $\sigma_m$, $\eta_m$,...,<$r_1$, $\sigma_1$, $\eta_1'$, $\zeta'$>...>.
where $\eta_1' = \eta_1.\beta$. Let p correspond with the 0-node (i.e., it is connected to the input generator) in the horizontal path of $\rho_n$ and suppose <q,p> exists. We extend the graph by adding a new node p0, a vertical connection from p to p0, and horizontal arrows <p0,-1>=$\zeta'$ and <q,p0>= $\eta_1'$.




The *horizontal path* of $\rho_{n+1}$ is obtained by replacing <q,p> by <q,p0> and <p,-1> by <p0,-1> in the horizontal path of $\rho_n$.

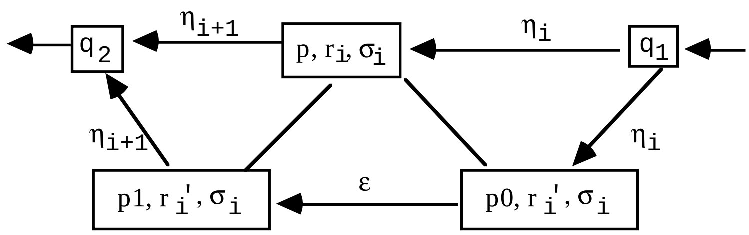
***b*** Suppose $\rho_n$ ——> $\rho_{n+1}$ is caused by the i-th subconfiguration and suppose $r_i$ does not begin with a fork- statement. We then have
$\rho_{n+1}$=<..., $r_{i+1}$, $\sigma_{i+1}$, $\eta_{i+1}'$, <$r_i'$, $\sigma_i'$, $\eta_i'$...<...>...>.
If $r_i$ begins with a read-statement, then $\eta_i'$ = rest($\eta_i$). If $r_i$ writes a value $\alpha$, then $\eta_{i+1}'$ = $\eta_{i+1}.\alpha$. In the other situations $\eta_i$ and $\eta_{i+1}$ do not change.
Let [p, $r_i$, $\sigma_i$] be the corresponding node in $\rho_n$ and let <$q_2$,p>=$\eta_{i+1}$ and <p,$q_1$>=$\eta_i$ be the outgoing and incoming arrows in the horizontal path of $\rho_n$. We extend the graph

by adding a new node [p0,$r_i$',$\sigma_i$'] and vertical and horizontal connections in the following way




The *horizontal path* of $\rho_{n+1}$ is obtained by replacing <$q_2$,p> and <p,$q_1$> in the horizontal path of $\rho_n$ by <$q_2$,p0> and <p0,$q_1$>.

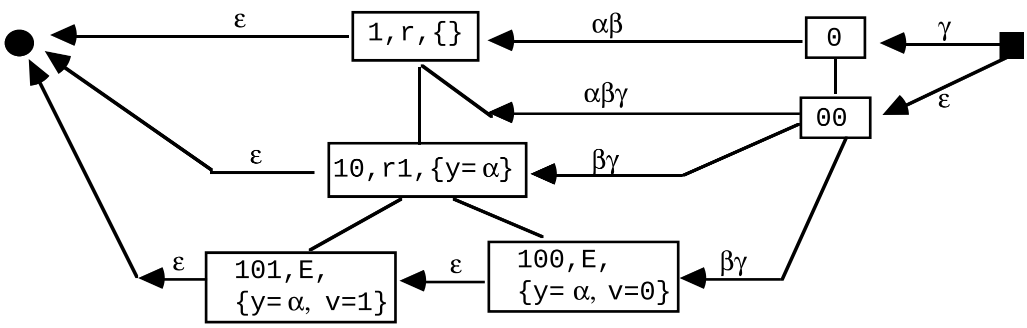
***c*** Suppose $\rho_n$ ——> $\rho_{n+1}$ is caused by a change of $r_i$ and $r_i$ begins with a fork-statement.
Let $\rho_{n+1}$=<..., <$r_i$', $\sigma_i$', ε, <$r_i$', $\sigma_i$'', $\eta_i$, ...<...>...>.
Let <$q_2$,p>=$\eta_{i+1}$ and <p,$q_1$>=$\eta_i$ be horizontal arrows in $\rho_n$. We extend the graph in the following way:




The horizontal path of $\rho_{n+1}$ is obtained by replacing <$q_2$,p> and <p,$q_1$> in the horizontal path of $\rho_n$ by <$q_2$,p1>, <p1,p0> and <p0,$q_1$>.

**3.2.1 Example.** We show the computation graph of the following computation.

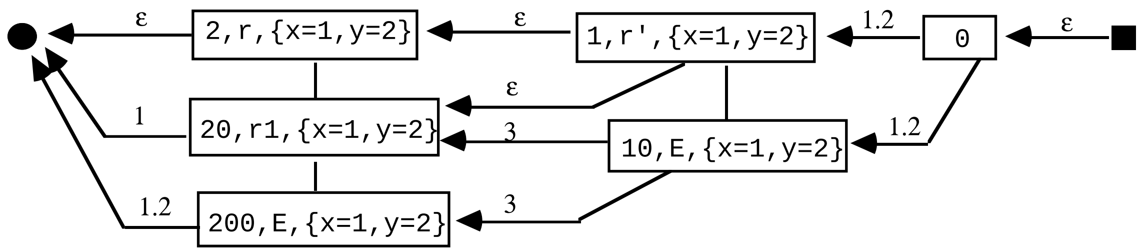
<read(y) : (fork(v)) : E), {}, αβ, γ> = <r, {}, αβ, γ>
——> <read(y) : (fork(v): E), {}, αβγ, ε> = <r, {}, αβγ, ε>
——> <fork(v) : E, {y=α}, βγ, ε> = <r1, { y=α}, βγ, ε>
——> <E, { y=α, v=1}, ε, <E, {y=α, v=0}, βγ, ε>>



**3.2.2 Example.** Consider the computation c(ρ):

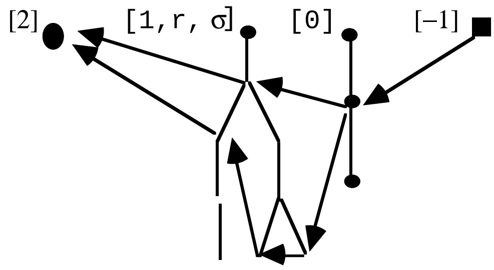
<write(x) : (write(y) : E), {x=1, y=2}, ε, <write(x+y) : E, {x=1,y=2}, 1.2, ε>>
$\xrightarrow{1}$ <write(y) : E, {x=1, y=2}, ε, <write(x+y) : E, {x=1,y=2}, 1.2, ε>>
$\longrightarrow$ <write(y) : E, {x=1, y=2}, 3, <E, {x=1,y=2}, 1.2, ε>>
$\xrightarrow{2}$ <E, {x=1, y=2}, 3, <E, {x=1,y=2}, 1.2, ε>>

Let r=write(x) : (write(y) : E), r1=write(y) : E, r'=write(x+y) : E. Below we show the computation graph of c(ρ). Notice that in this case node 2 is not the output receiver because the initial configuration was a nested one.




### 3.2.3 Horizontal paths and trees in the graph of a computation

Consider the graph of a computation starting in ρ=<r, σ, ε, ζ>. For any node p, there is a tree with p as root such that for any node q in the tree there is a unique direct or indirect vertical connection from p to q. The tree structure and two horizontal paths in some graph are illustrated in the following figure.



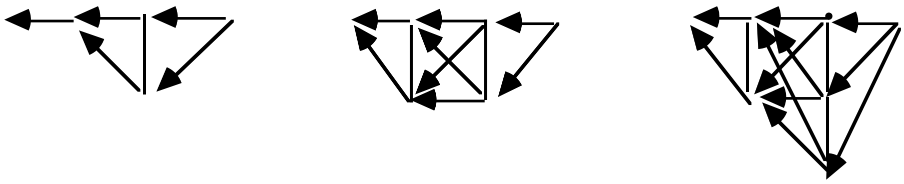

**Terminology.** We will use the terms *father and son* for two nodes between which a vertical connection exists. We will use the terms *mother and daughter* for two nodes connected by a horizontal arrow. If p is a node, then any q which contains p as a subsequence is called a *descendant* of p. In other words, p is in the tree rooted in q. We use pTq to denote this relation.

# 4 The Graph of all Computations

For each initial configuration ρ=<r, σ, ε, ζ> our transition system specifies many computations c(ρ). It is our intention to define the operational meaning of ρ as the maximal output stream over all such c(ρ). This definition makes sense only if such a maximal stream exist. We will actually prove a stronger result than this.

For each computation starting in some configuration we have defined a computation graph in which all transition steps are recorded and in which all intermediate stages are modelled as horizontal paths. We will show that it is possible to combine all such computation graphs in one diagram, the graph of all computations **C.**

We will specify a construction of **C**. This construction will be based on a special computation, starting from ρ, called C(ρ). The backbone of the construction of **C** will be the construction of the graph of only the computation C(ρ). However, in each step of the construction, every time a new node has been created together with the vertical connection with its father and the new horizontal arrows defining the next stage in C(ρ), many more horizontal arrows to and from the new node will be added as well. These additional arrows can be combined into horizontal paths which correspond with stages in other computations than C(ρ). The following picture sketches the initial phases in the construction of such a **C**.



## 4.1 Constructing the special computation C(ρ)

Let $\rho=<r, \sigma, \varepsilon, \zeta>$. We now define a special computation C(ρ) serving as the basis for the graph of all computations. This computation begins with changing the input stream if it is enabled. In the next steps it changes the resumptions and states of subconfigurations from right to left if they are enabled. This process repeats as often as possible.

1 Intial step: let $\rho_0 =<r, \sigma, \varepsilon, \zeta>$. We say the 0-th subconfiguration is ready to proceed.

2 Suppose we have arrived at the stage

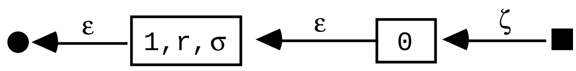
$$<r_m,\sigma_m,\eta_m,\ldots, \eta_{i+1}, <r_i, \sigma_i, \eta_i, <\ldots,<r_1, \sigma_1, \eta_1, \zeta>\ldots>$$

and suppose the i-th subconfiguration is ready to proceed. If this subconfiguration is enabled, then perform the transition caused by this subconfiguration which yields the next stage. If it is disabled, then do nothing. If i=m, then define that now the 0-th subconfiguration will be ready to proceed. If i<m and $r_i$ does not start with a fork, then we define that the (i+1)-th subconfiguration is ready to proceed. If i<m and $r_i$ starts with a fork-statement, then we define that the (i+2)-th subconfiguration is ready to proceed.

## 4.2 An algorithm constructing the graph C of all computations

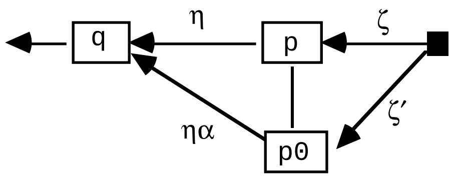
The construction of **C** is based on the construction of the graph of C(ρ) as described in subsection 4.1. Every step in the construction of C(ρ) entails the creation of one or two new nodes, and one or two new vertical and horizontal connections. However, the construction of **C** will specify that many more horizontal arrows must be added. This will be described below. Notice that among those new horizontal arrows the ones that would have been generated in the construction of C(ρ) are included.

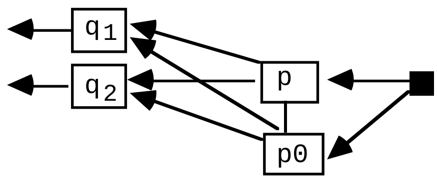
*Start of the construction of* ***C.*** We draw ρ as in C(ρ).



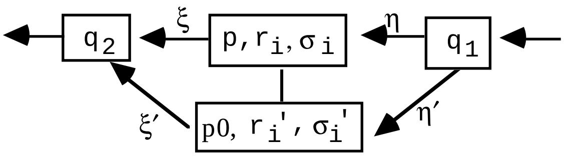

*Extension of the graph.* Suppose we have constructed the graph **C** using the first n steps of C(ρ). We use now the (n+1)-th step of C(ρ) to construct the new node(s) and vertical connections. Then we add new horizontal arrows according to the different cases described below.

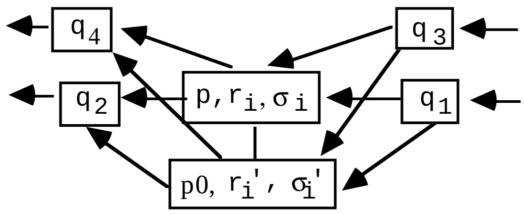
***a*** In the graph of C(ρ) a new 0-node [p0] is created. From the fact that this transition was enabled we infer that <p,-1>= $\zeta = \alpha\zeta' \neq \varepsilon$ .
Draw <p0,-1> with label $\zeta'$. For every horizontal arrow <q,p> we add a new horizontal arrow <q,p0>. If <q,p>=η, then <q,p0>=ηα. Notice that the new arrows for C(ρ) are also added in this way. This step is depicted in the rightmost picture below. The leftmost picture sketches how two of the horizontal arrows and their labels are added.



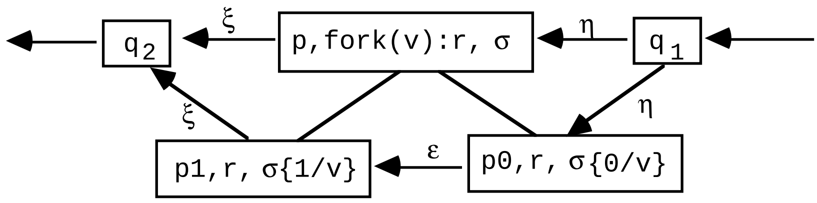




***b*** In the graph of C(ρ) a new node [p0, $r_i$', $\sigma_i$'] is created. Suppose its father is [p, $r_i$, $\sigma_i$] and suppose $r_i$ does not begin with a read- or write-statement. For every <p,$q_1$>, add a new arrow <p0,$q_1$> with the same label. For every <$q_2$,p>, add a new arrow <$q_2$,p0> with the same label. Thus in the picture below we have $\eta=\eta'$, $\xi=\xi'$.
If $r_i$ causes a value α to be written, then for every arrow from or to p, we add a new arrow similar to the case above. However, now we define <$q_2$,p0>=<$q_2$,p>α. That means in the picture $\eta=\eta'$, $\xi\alpha=\xi'$.
If $r_i$ begins with a read-statement, then for every <$q_2$,p> we add a new arrow <$q_2$,p0> with the same label. However, we now only add an arrow <p0,$q_1$> if there exist an arrow <p,$q_1$>≠ε. In that case we set <p0,$q_1$>=η′=rest(<p,$q_1$>).
In the pictures below the left one gives the construction only in C(ρ), while the right one sketches the full construction.




***c*** In the graph of C(ρ) two new nodes [p0, r, σ{0/v}] and [p1, r, σ{1/v}] are added because the node p started with a fork-statement.
First of all, the arrow <p1,p0>=ε should be added. Furthermore, we add for every <$q_2$,p>, an arrow <$q_2$,p1>. Also, for every <p,$q_1$>, we add <p0,$q_1$>. The corresponding labels are defined by <$q_2$,p1>=<$q_2$,p>=ξ and <p0,$q_1$>=<p,$q_1$>=η.

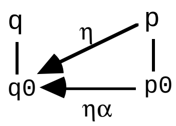


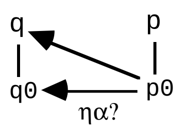
## 4.3 Horizontal paths in C

In this subsection we will prove that the horizontal paths in **C** characterize all stages of all computations. A *horizontal path* in **C** is a sequence of arrows $\langle 2,p_m\rangle$, $\langle p_m,p_{m-1}\rangle$,..., $\langle p_2,p_1\rangle$, $\langle p_1,p_0\rangle$, $\langle p_0,-1\rangle$ in the graph. Having proven this property we then will use the labels of output arrows $\langle 2,p_m\rangle$ to find the output streams of the computations. This will provide the basis for defining the operational semantics.

**Induction and proofs.** In the sequel we will often prove properties of nodes and paths by induction on the time of creation of the youngest node. This amounts to proving the desired property for a graph after adding new nodes and connections, from the induction hypothes that this property holds for the graph just before the additions were made.

If <q,p0> is constructed from <q,p>, then there are certain relations between <q,p> and <q,p0>. For example, if p begins with a statement writing α, then <q,p0>=<q,p>α. Similarly if <q0,p> is constructed from <q,p>, there are also relations between them. For example if q begins with a read statement, then <q,p>≠ε and <q0,p>=rest(<q,p>). Now consider two nodes p and q with sons p0 and q0 and suppose <q0,p0> exists. This arrow may have been constructed in two ways. If p0 is younger than q0, then this arrow and its label are constructed from <q0,p>, otherwise they are constructed from <q,p0>. We can ask ourselves whether the label of a new arrow depends on the arrow it is derived from. For example if p begins with writing α, then the left picture shows the first situation and <q0,p0>=ηα. However, it is not immediately clear whether the same label ηα will be obtained if q0 is constructed after p0 (cf. the right picture).




The next lemma states that this is indeed the case.

**4.3.1 Lemma.** Consider the graph of computations **C** of ρ. Suppose<q0,p0> exists.
If <q0,p> exists then

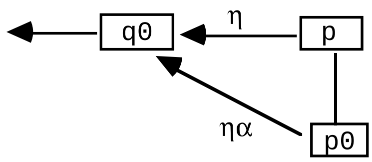
*a* if p begins with a statement which is not a write statement then <q0,p0>=<q0,p> .
*b* if p begins with writing α, then <q0,p0>=<q0,p>α.

Similar properties hold for a son of p which is of the form p1.
If <q,p0> exists, then

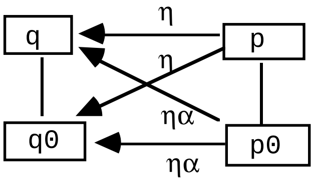
*c* if q does not begin with a read-statement, then <q0,p0>=<q,p0>.
*d* if q begins with a read statement, then <q,p0>≠ε and <q0,p0>=rest(<q,p0>).

**Proof.** Consider the graph corresponding only with configuration ρ which is the initial stage in the construction of **C**. There is at most one arrow from a node to its left neighbour and from its right neighbour. For this case the lemma is trivially true. Suppose these properties are true before some new node and new arrows are constructed. We want to prove they are still true after the creations. We check *b* completely, distinguishing three cases, and then check *d* only for the situation where the transition from p to p0 does not involve a write.

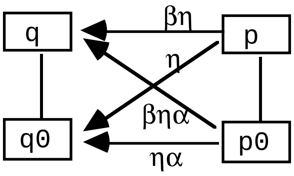
- Suppose <q0,p0> is constructed from <q0,p>. From the definition of the labels for the new arrows in the construction of **C**, we have <q0,p0>=<q0,p>α.



- Suppose <q0,p0> is constructed from <q,p0> and suppose furthermore that q does not start with a read statement. Then <q0,p0>=<q,p0> and q0 is constructed later than p0. We know <q0,p> is constructed from <q,p> because q0 is constructed later than p. Let <q,p>=$\eta$. Then <q0,p>=$\eta$. By induction we know also <q,p0>=<q,p>$\alpha$=$\eta\alpha$. This implies <q0,p0>=<q,p0>=<q0,p>$\alpha$=$\eta\alpha$.

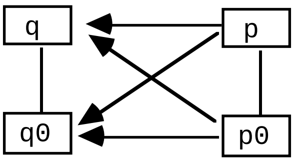



- Suppose <q0,p0> is constructed from <q,p0> and q starts with a read statement.




  When q0 is constructed, <q0,p> and <q0,p0> are also constructed from <q,p> and <q,p0> respectively. Since q starts with a read statement and <q0,p> exists, by the construction of **C** we have that <q,p>=$\beta\eta\neq\varepsilon$ and that <q0,p>=$\eta$. On the other hand, by induction <q,p0>=<q,p>$\alpha$=$\beta\eta\alpha$. Thus <q0,p0>=$\eta\alpha$.

- Now we check *d* in the situation when the transition from p to p0 does not involve a write statement. If <q0,p0> is constructed from <q,p0>, then *d* is true by the algorithm constructing **C**. Let us now consider the situation where <q0,p0> is constructed from <q0,p>, i.e. p0 is constructed later than q0. We want to prove that <q,p0>$\neq\varepsilon$ and <q0,p0>=rest(<q,p0>).




  Since there is no writing in the transition from p to p0, we have <q0,p0>=<q0,p>. Since p0 is constructed later than q, <q,p0> is constructed from <q,p>. That means <q,p> exists and <q,p>=<q,p0>. By induction, we know <q,p>$\neq\varepsilon$, <q0,p>=rest(<q,p>). So we know <q,p0>$\neq\varepsilon$. From <q0,p0>=<q0,p>=rest(<q,p>), we have <q0,p0>=rest(<q,p0>).

**4.3.2 Lemma.** If <p,q> and <p,q'> exist in **C**, then either qTq' or q'Tq. A similar property holds for <q,p> and <q',p>.

**Partial proof.** We only prove the first situation. At the beginning of the construction of **C**, there is at most one arrow to each node so this property is trivially true. We consider only the situation that p0 is the only new node created as the son of p. Suppose this property is true before creating p0 and the new connections. We have to check whether the property still holds for the new arrows to p0 as well as the new arrows from p0 to some old nodes p'.


Consider <p0,q> and <p0,q'>. The arrows have been created because <p,q> and <p,q'> already exist. By induction, this property is true.

Now consider a new arrow <p’,p0> to an old node p'. The existence of <p’,p0> implies the existence of <p’,p>. We should compare q occurring in an old arrow <p',q> with p0. From induction pTq or qTp. The first situation implies p0Tq. The second situation implies p=q or p0=q. Since p0 is the youngest node and q is an old node we have that p0=q is impossible. Thus p=q and p0Tq.

**Remark.** Let p0 be the youngest node in a certain stage during the construction of **C**. Then <p’,p0> is lower than any other arrow going to p’, i.e., p0Tq if <p’,q> exists. Similarly for <p0,p’>.

**4.3.3 Lemma** Let q’Tq. If for every node q" in the path from q to q’, <p,q”> exist, then <p,q’>≥<p,q>. If for every node q” in the path from q to q’, <q”,p> exists, then <q’,p>⊂<q,p> .

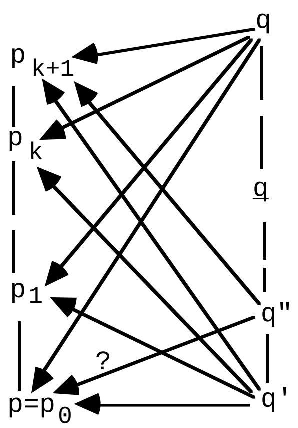
**Proof.** We consider only <p,q> and <p,q’>. Let $q_0$=q,…,$q_k$=q’ be the nodes in the path from q to q’ such that $q_i$ is the father of $q_{i+1}$. If p is not a real descendant of a node, i.e. p is in the initial configuration ρ, then there is only one way to construct $<p,q_{i+1}>$, namely from $<p,q_i>$. We have $<p,q>\le<p,q_1>\le\ldots\le<p,q_k>$ by the algorithm constructing **C**. If p is a descendant of some other node, then we can apply lemma 4.3.1 stepwisely to get $<p,q>=<p,q_0>\le<p,q_1>\le\ldots\le<p,q_k>=<p,q'>$

**4.3.4 Lemma.** Let q'Tq. If <p,q> and <p,q’> exist in **C**, then for every node q” in the path from q to q’, <p,q”> exists. Similar relations hold for <q,p>, <q’,p> and the path between q and q’.

**Proof.** We prove this by induction on the length of the path from q to q’. If this length is 0, then q=q’=q” and the lemma is trivially true. Now suppose the lemma is true for paths with length n. We will prove that the lemma also holds for paths with length n+1.

If q’ is constructed later than p, then <p,q’> is constructed from <p,q”> where q” is the father of q’. For the path from q to q” we have the property of the lemma from induction. Together with the existence of <p,q’> we have the property for the nodes in the whole path.

If p is constructed later than q’, then there are $p_0$=p, $p_1$,…, $p_k$, $p_{k+1}$ where $p_i$ is the father of $p_{i-1}$ for 1≤i≤k+1 and $p_0,\ldots p_k$ are all younger than q’ but $p_{k+1}$ is older than q’. A node $p_{k+1}$ with this property must exist because all nodes in the initial configuration ρ are older than q’. It is clear that $<p_i,q'>$ is constructed from $<p_{i+1},q'>$ for i=0,…,k. However, $<p_{k+1},q'>$ is constructed from $<p_{k+1},q''>$ where q” is the father of q’ because $p_{k+1}$ is constructed earlier than q’.



Since $\langle p,q\rangle$ exists and q is older than p, we can find $\langle p,q\rangle$, $\langle p_1,q\rangle$, …,$\langle p_{k+1},q\rangle$. Because $\langle p_{k+1},q\rangle$ and $\langle p_{k+1},q''\rangle$ exist we know by induction that all $\langle p_{k+1},\underline{q}\rangle$ exist where $\underline{q}$ is any node on the path from q to q”.

We will now show that $\langle p,q''\rangle$ must exist. If the transition from $p_{k+1}$ to $p_k$ does not involve a read, then $\langle p_k,q''\rangle$ surely can be constructed from $\langle p_{k+1},q''\rangle$. If the transition from $p_{k+1}$ to $p_k$ does involve a read, we have $\langle p_{k+1},q\rangle\neq\varepsilon$ from the existence of $\langle p_k,q\rangle$. By lemma 4.3.3 we know $\langle p_{k+1},q''\rangle\geq\langle p_{k+1},q\rangle\neq\varepsilon$. Thus $\langle p_k,q''\rangle$ can be constructed from $\langle p_{k+1},q''\rangle\neq\varepsilon$. From lemma 4.3.1 and the existence of $\langle p_k,q''\rangle$ and $\langle p_k,q\rangle$ we can assume the existence of all $\langle p_k,\underline{q}\rangle$ where $\underline{q}$ is any node in the path from q to q”. This line of reasoning can be carried on until we have proved the existence of $\langle p_0,q''\rangle=\langle p,q''\rangle$. From the existence of $\langle p,q''\rangle$ and $\langle p,q\rangle$ using the induction hypothesis we have the lemma.

**4.3.5 Theorem.** Let q ‘Tq in the graph **C** of all computations.

***a*** If $\langle p,q\rangle$ and $\langle p,q'\rangle$ exist, then $\langle p,q\rangle\leq\langle p,q'\rangle$.

***b*** If $\langle q,p\rangle$ and $\langle q',p\rangle$ exist, then $\langle q',p\rangle\subset\langle q,p\rangle$.

**Proof.** Consider $\langle p,q\rangle$ and $\langle p,q'\rangle$ only. From lemma 4.3.4 we know $\langle p,q''\rangle$ exist for all q” in the path from q to q’. From lemma 4.3.3 we have the result.

**4.3.6 Theorem.** Every horizontal path in **C** corresponds with a stage of a computation.

**Partial proof.** We will show that the theorem holds for every subgraph of **C** during the construction of **C**. Whenever a new node and new edges are added to **C**, new horizontal paths are created. It is sufficient to prove the properties for these paths, using induction on the time of creation of the youngest node(s) in C(ρ). Initially **C** contains only the horizontal path corresponding with ρ, i.e., the 0-th stage of all computations. Suppose all horizontal paths in the graph before the creation of the new node(s) correspond with a stage of some computation.

- *A new 0-node p0 is constructed as the son of 0-node p.*
  It is only necessary to consider a new horizontal path after this construction, a path which uses the new node p0. Let the new path be
  $$\langle 2,p_m\rangle, \langle p_m,p_{m-1}\rangle,\ldots, \langle p_1,p0\rangle, \langle p0,-1\rangle.$$
  The existence of $\langle p_1,p0\rangle$ and $\langle p0,-1\rangle$ implies the existence of $\langle p_1,p\rangle$ and $\langle p,-1\rangle$. Moreover from the fact that the construction of p0 was possible we have that $\langle p,-1\rangle\neq\varepsilon$ by the algorithm constructing **C**. Thus we have an old horizontal path
  $$\langle p_m,p_{m-1}\rangle,\ldots, \langle p_1,p\rangle, \langle p,-1\rangle$$
  By induction, this corresponds with a stage $\rho_n$ of some computation c(ρ):
  $$\langle r_m,\sigma_m,\eta_m,\ldots\langle r_1,\sigma_1,\eta_1,\zeta\rangle\ldots\rangle$$
  where $\zeta=\langle p,-1\rangle\neq\varepsilon$ and $\eta_1=\langle p_1,p\rangle$. If $\zeta=\alpha\zeta'$, define
  $$\rho_{n+1}=\langle r_m,\sigma_m,\eta_m,\ldots\langle r_1,\sigma_1,\eta_1.\alpha,\zeta'\rangle\ldots\rangle$$
  Then the transition sequence
  $$\rho \xrightarrow{*} \rho_n \longrightarrow \rho_{n+1} \longrightarrow \ldots$$
  is a computation with $\rho_{n+1}$ as one of its stages. The given path corresponds with $\rho_{n+1}$.
- *A new node p0 is constructed from p and the first statement in p is a read-statement*
  Let a new path be
  $$\langle 2,p_m\rangle,\ldots, \langle p_{i+1},p0\rangle, \langle p0,p_{i-1}\rangle, \ldots, \langle p_0,-1\rangle.$$
  The existence of $\langle p_{i+1},p0\rangle$ and $\langle p0,p_{i-1}\rangle$ implies the existence of $\langle p_{i+1},p\rangle$ and $\langle p,p_{i-1}\rangle\neq\varepsilon$.
  Thus we have an old horizontal path
  $$\langle 2,p_m\rangle, \ldots, \langle p_{i+1},p\rangle , \langle p,p_{i-1}\rangle ,\ldots, \langle p_1,p_0\rangle, \langle p_0,-1\rangle$$
  By induction this corresponds with a stage $\rho_n$ of some computation c(ρ):
  $$\langle r_m,\sigma_m,\eta_m,\ldots, \eta_{i+1}, \langle r_i, \sigma_i, \eta_i, \langle\ldots\rangle\ldots\rangle$$

where $\eta_{i+1}$=<$p_{i+1}$,p> and $\eta_i$=<p,$p_{i-1}$>≠ε and $r_i$ begins with read. The result of this read is a new state $\sigma_i$' and a new buffer $\eta_i$'=rest($\eta_i$). Let the statement in p0 be $r_i$' and let

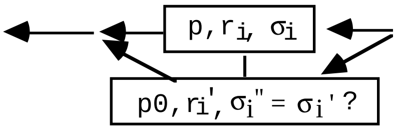
$$\rho_{n+1}=<r_m,\sigma_m,\eta_m,\ldots,\eta_{i+1}, <r_i', \sigma_i', \eta_i', <\ldots>\ldots>.$$

Then there is a transition $\rho_n \longrightarrow \rho_{n+1}$ and the transition sequence

$$\rho \xrightarrow{*} \rho_n \longrightarrow \rho_{n+1} \longrightarrow \ldots$$

is a computation which has $\rho_{n+1}$ as one of its stages.

We still have to prove that the given path in **C** corresponds with $\rho_{n+1}$. More specifically, we should prove that the state in node p0 in graph **C** is the same as $\sigma_i$'. According to the construction of the graph **C**, we use the first element of some <p,q>≠ε in C(ρ) to read from. Suppose reading from <p,q> transform $\sigma_i$ into $\sigma_i$''. Do we have $\sigma_i$'= $\sigma_i$''?



(constructed by read in C(ρ))

Apparently we have two arrows to p, <p,q> and <p,$p_{i-1}$>, which are both non-empty. By lemma 4.3.5 one arrow is a subsequence of the other. Therefore they have the same value as the first element so which one we use to read from makes no difference. Thus we have the same new state.

**4.3.7 Lemma.** Consider the graph **C** of computations of ρ. Suppose p does not begin with a fork-statement and p0 exists. Then

*a* If <q,p> exists, then <q,p0> exists.

*b* If <p,q> exists and p does not begin with a read-statement, then <p0,q> exists.

*c* If <p,q> exists and <p,q>≠ε and p begins with a read-statement, then <p0,q> exists.

Suppose p begins with a fork statement and p0, p1 exist. Then <p1,p0>=ε exists. Furthermore, the existence of <q,p> implies the existence of <q,p1> and the existence of <p,q> implies the existence of <p0,q>.

**Proof.** We will prove *a* and *c*, the rest being similar.

- *Proof of a* (See fig. A)
  If p0 is constructed later than q, then by the algorithm constructing **C** we have that <q,p0> is constructed from <q,p>. Now suppose q is constructed later than p0. Let $q_0$=q,$q_1$,…$q_k$, $q_{k+1}$ be a path where $q_{i+1}$ is the father of $q_i$ for i=0, …, k such that $q_0$,…$q_k$ are constructed later than p0 but $q_{k+1}$ is constructed earlier than $p_0$. Thus <$q_i$,p> is constructed from <$q_{i+1}$,p> for i=0,…, k. Since $q_{k+1}$ is constructed earlier than $p_0$, <$q_{k+1}$,p0> must have been constructed from <$q_{k+1}$,p>. If $q_{k+1}$ does not begin with a read statement, then <$q_k$,p0> will be constructed from <$q_{k+1}$,p0>. If $q_{k+1}$ begins with a read statement, then by using <$q_{k+1}$,p>≤<$q_{k+1}$,p0> (theorem 4.3.5) we know <$q_k$,p0> will be constructed. Using similar arguments we can prove the existence of <$q_{k-1}$,p0>,…,<$q_1$,p0>, <$q_0$,p0>.

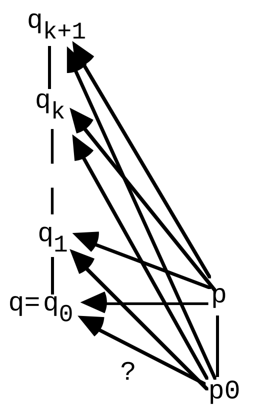



fig. A

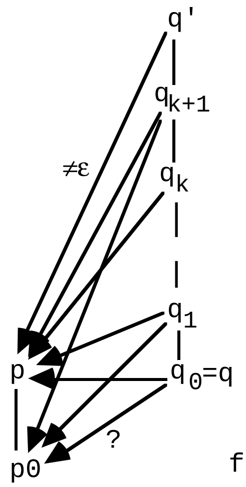



fig. B

- *Proof of c* (See fig. B)
  Suppose p0 is constructed later than q, then the existence of $<p,q>\neq\varepsilon$ implies the existence of $<p0,q>$. This follows from the algorithm constructing **C**. Now suppose p0 is constructed earlier than q. Since p begins with a read statement, p0 must have been constructed by reading from some $<p,q'>\neq\varepsilon$ in $C(\rho)$. This implies also p0 is constructed later than q'. Let $q_0=q,q_1,\ldots,q_k,q_{k+1}$ be a path such that $q_{i+1}$ is the father of $q_i$ for $i=1,\ldots,k$ ; $q_0,\ldots,q_k$ are constructed later than p0 and $q_{k+1}$ is constructed earlier than p0. Then $<p,q>=<p,q_0>,\ldots,<p,q_{k+1}>$ exist. Since both $<p,q_{k+1}>$ and $<p,q'>$ exist, we have $q'Tq_{k+1}$ or $q_{k+1}Tq'$. However, q' can never be a real descendant of $q_{k+1}$ because $q_k$ is a son of $q_{k+1}$ and $q_k$ is constructed later than p0 which is again constructed later than q'. Thus $<p,q_{k+1}>\geq<p,q'>\neq\varepsilon$. Thus $<p0,q_{k+1}>$ can be constructed from $<p,q_{k+1}>\neq\varepsilon$. Then we can construct stepwisely $<p0,q_k>$, $\ldots,<p0,q_0>=<p0,q>$.

**4.3.8 Theorem.** Every stage of a computation can be found as a horizontal path in **C**.

**Partial proof.** Let $c(\rho)$ be any chosen computation. By induction on n we will prove that every stage $\rho_n$ in $c(\rho)$ corresponds with a horizontal path in **C**. So, suppose the n-th stage $\rho_n$ of $c(\rho)$ corresponds with a horizontal path in **C**. We want to find a horizontal path in **C** corresponding with $\rho_{n+1}$.

Let $\rho_n=<r_m,\sigma_m,\eta_m,\ldots,\ \eta_{i+1},\ <r_i,\ \sigma_i,\ \eta_i,\ <\ldots<r_1,\sigma_1,\eta_1,\zeta>\ldots>$. Let $\rho_{n+1}$ be caused by the i-th subconfiguration. Suppose $\rho_n$ is represented by the horizontal path

$$<2,p_m>,\ldots,\ <p_{i+1},p>,\ <p,p_{i-1}>\ ,\ldots,<p_0,-1>$$

We consider only two cases : $r_i$ begins with a write- or read-statement.

- Suppose $r_i$ begins with write. Consider the horizontal path of $C(\rho)$ which passes p when p is constructed. In the next round from right to left in $C(\rho)$, the write-statement in $r_i$ in node p will enable the construction of a new node p0 of $C(\rho)$. So we have p0 in **C**. By lemma 4.3.7 we can construct $<p_{i+1},p0>$ and $<p0,p_{i-1}>$. By lemma 4.3.1 we have $<p_{i+1},p0>=<p_{i+1},p>\alpha$ and $<p0,p_{i-1}>=<p,p_{i-1}>$.
  Now consider $\rho_{n+1}$ of $c(\rho)$:
  $$\rho_{n+1}=<r_m,\sigma_m,\eta_m,\ldots,\ \eta_{i+1}\alpha,\ \ <r_i',\ \sigma_i,\ \eta_i,\ <\ldots>\ldots>.$$
  This stage of $c(\rho)$ can be represented by
  $$<2,p_m>,\ldots,<p_{i+1},p0>,\ <p0,p_{i-1}>\ ,\ldots,<p_0,-1>$$
  because $<p_{i+1},p0>=<p_{i+1},p>\alpha$ and $<p0,p_{i-1}>=<p,p_{i-1}>$.

- If $r_i$ begins with a read-statement, then $<p,p_{i-1}>\neq\varepsilon$. Consider the moment that node p has been generated in the construction of **C**. At that moment a horizontal arrow $<p,q>$ has been added to **C** which also appears in a stage in the computation $C(\rho)$. Therefore we have $<p,q>$ and $<p,p_{i-1}>$ in **C**. We can infer that either $qTp_{i-1}$ or $p_{i-1}Tq$ from lemma 4.3.2 We first consider the situations $qTp_{i-1}$ (so that $<p,p_{i-1}>\leq<p,q>$ and thus $<p,q>\neq\varepsilon$, (cf. fig. A) and $p_{i-1}Tq$ where $<p,q>\neq\varepsilon$ (cf. fig. B). Now p0 will be constructed in $C(\rho)$. We distinguish two cases. If in the next round of $C(\rho)$ from right to left we can construct q' as the son of q, then p0 will be constructed by reading from $<p,q'>\geq<p,q>\neq\varepsilon$ (see fig. A, B). If in the next round q is disabled, then p0 will be constructed reading from $<p,q>$ (In fig. A, B we should in such a case identify q with q').
  If $p_{i-1}Tq$ and if $<p,q>=\varepsilon$, then in the path from q to $p_{i-1}$ there will be a highest node q' such that $<p,q'>\neq\varepsilon$. Consider the nodes in the path between q and q' and the arrows from these nodes to p: $<p,q>,\ldots, <p,q'>$ . All these arrows occur in stages of $C(\rho)$. This can be inferred from the fact that p remains disabled until q' appears and the fact that $<p,q>$ occurs in some stage of $C(\rho)$. The next node after constructing q' will be p0 because$<p,q'>\neq\varepsilon$.
  Since p0 can be constructed and $<p,p_{i-1}>\neq\varepsilon$, from lemma 4.3.7 we can construct arrows $<p0,p_{i-1}>$ and $<p_{i+1},p0>$. By lemma 4.3.1 we have $<p0,p_{i-1}>=rest(<p,p_{i-1}>)$ and $<p_{i+1},p0>= <p_{i+1},p0>$.
  Now consider $\rho_{n+1}$ of $c(\rho)$:

  $$\rho_{n+1}=<r_m,\sigma_m,\eta_m,\ldots, \eta_{i+1}, <r_i', \sigma_i', \eta_\iota', <\ldots>\ldots>$$

  where $\eta_{i'}=rest(\eta_i)$. We can prove $\sigma_i'=\sigma_i$ in the same way as we have done in the last part of lemma 4.3.6. This stage of $c(\rho)$ can be represented by

  $$<2,p_m>,\ldots,<p_{i+1},p0>, <p0,p_{i-1}> ,\ldots,<p_0,-1>.$$

**Figures.** In the following figures $<p,p_{i-1}>$, $<p0,p_{i-1}>$ are arrows in $c(\rho)$ and $<p,q>$, $<p,q'>$, $<p0,q'>$ are arrows in $C(\rho)$. In figure A, B it is possible that q=q' if q is disabled.

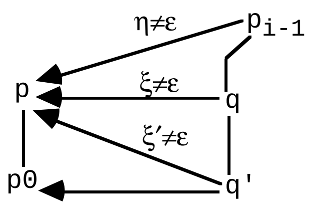



fig.A

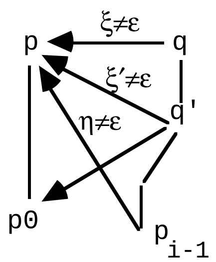



fig.B

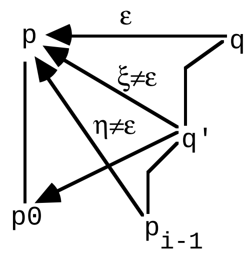



fig. C

## 4.4 Operational semantics

After the characterization of the horizontal paths in **C** we will use the labels of output arrows $<2,p_m>$ to find the output streams of computations. For defining the operational semantics we are going to use the maximal output over all computations. This definition makes sense because we will prove that output streams of different computations are always subsequences of each other

**4.4.1 Lemma.** Let p be in some horizontal sequences $\rho_n$ of $c(\rho)$. Then every ancestor q of p is also in some computation stage $\rho_k$, $k\leq n$ of $c(\rho)$.

**Proof outline** Suppose this is true for $\rho_n$. For $\rho_{n+1}$ one or two new nodes are added. The old nodes are in $\rho_k$, $k \leq n$. The new nodes are in $\rho_{n+1}$.

**4.4..2 Theorem.** For any stage $\rho_n$ of any computation $c(\rho)$, the label of $<2,p>$ in the horizontal path of $\rho_n$ is the output stream of this computation to this stage.

**Proof.** We prove this by induction on n. For $\rho_0=\rho$ we have empty output. This is the same as the label $<2,1>$. Let $\rho_n$ be represented by a horizontal path in **C** with the leftmost arrow $<2,p_m>$. By induction the label on $<2,p_m>$ equals the output stream of $c(\rho)$ until $\rho_n$. It is only necessary to check the difference between $<2,p_m>$ and the leftmost arrow in the path of $\rho_{n+1}$.

- If the transition $\rho_n \longrightarrow \rho_{n+1}$ is caused by some i-th subconfiguration, i<m, then $<2,p_m>$ is still in the path of $\rho_{n+1}$. The output stream does not change either.
- Let the transition from $\rho_n$ to $\rho_{n+1}$ be caused by the resumption r in $p_m$ and and let the son of $p_m$ be p. If r does not begin with a write-statement, then $<2,p>=<2,p_m>$. The output stream until $\rho_{n+1}$ also does not change. If r causes a value $\alpha$ to be written, then $\alpha$ is appended both to the output stream and to $<2,p>$ .

**4.4.3 Corollary.** Let $c(\rho)$ be a computation and $<2,p_n>$ be the arrow in its n-th stage. Then

output stream of $c(\rho)=\cup_n<2,p_n>$,

where n ranges over all the computation steps of $c(\rho)$.

**4.4.4 Corollary.** Consider two computations $c(\rho)$ and $c'(\rho)$. Then the output stream of one of the two computations is a subsequence of the output stream of the other.

**Proof.** We need only to compare $\cup_n<2,p_n>$ and $\cup_m<2,q_m>$ which are the output streams of $c(\rho)$ and $c'(\rho)$ respectively. By lemma 4.3.2, for any n, m, either $<2,p_n>\leq<2,q_m>$ or $<2,p_m>\leq<2,p_n>$. Two cases should be considered:

- For every n there is an m such that $<2,p_n>\leq<2,q_m>$. In that case $\cup_n<2,p_n>\leq\cup_m<2,q_m>$.
- For some n there is no m such that $<2,p_n>\leq<2,q_m>$. In that case for every m we have $<2,q_m>\leq <2,p_n>$. Thus $\cup_m<2,q_m>\leq\cup_n<2,p_n>$.

**4.4.5 Corollary.** The computation $C(\rho)$ yields the maximal output stream.
**Proof.** Consider some $c(\rho)$. The output stream of $c(\rho)$ is $\cup_n<2,p_n>$ for all $p_n$ in the n-th stage of $c(\rho)$. On the other hand, every $p_n$ is a node in $C(\rho)$ because **C** is constructed by using $C(\rho)$. Thus the output stream of $C(\rho)$ is the union of all possible $<2,p>$ in **C**. This shows that the output stream of $C(\rho)$ is maximal in all computations.

**4.4.6 Operational semantics** . Let O: **L** $\rightarrow$ **Proc** be defined as follows. For any $s\in$ **L**, $\sigma\in$ **State** and $\zeta\in$ **Val**$^\infty$,

$O(s)(\sigma)(\zeta)$ = maximal output stream in the graph of all computations of $<s : E, \sigma, \varepsilon, \zeta>$.

## 4.5 All computations have the same length

In this subsection we will show that all computations starting in the same configuration take the same number of steps. Moreover, if they are finite then they reach precisely the same last stage.

**4.5.1 Lemma.** Let $c(\rho)$ be some computation. Then $c(\rho)$ is finite iff all computations are finite and the last stages of all computations are the same.

**Proof.** Let the computation $C(\rho)$ be $\rho_0 \longrightarrow \rho_1 \longrightarrow \ldots \longrightarrow \rho_n \longrightarrow \ldots$ which may be finite or infinite. Suppose some computation $c(\rho)$ is finite and let the last stage of $c(\rho)$ be given by the sequence

$<2,p_m>, <p_m,p_{m-1}>,\ldots, <p_i,p_{i-1}> ,\ldots , <p_0,-1>$.

We want to prove that every node in this path has no children in **C** anymore. In other words we reach also the end of C(ρ). Suppose nodes $p_0,\ldots,p_{i-1}$ have no children in **C** (or C(ρ)) but $p_i$ has a child p. We claim that the transition which brings $p_i$ to p in C(ρ) can also induce a transition from the last stage of c(ρ), which leads to a contradiction. For i>0, if the resumption of $p_i$ does not begin with read, then the transition of c(ρ) is surely enabled. Suppose the resumption begin with a read-statement. We now prove that $<p_i,p_{i-1}>$ is not empty which will lead to a contradiction. Since $p_i$ must have been enabled during the construction of C there must be a node q such that $<p,q> \neq\varepsilon$. Furthermore, $p_{i-1}$ has no son in **C** so$<p_i,q>\leq<p_i,p_{i-1}>$. This means $<p_i,p_{i-1}>\neq\varepsilon$. Similar arguments hold also for i=0. The discussion above implies $p_i$ has also no children in **C** and this is a contradiction. Since every stage of every computation is to be found in **C**, all other computations can be compared with C(ρ) in the same way. This proves that if one computation is finite then so are all the other ones and that they all reach the same final stage.

**4.5.2 Theorem.** All computations of ρ have the same length.

**Proof.** We have seen from the last lemma that computations are either all finite or all infinite. We want to prove if the computations are finite then they take precisely the same number of steps.

If C(ρ) is finite, then c(ρ) reaches also the same horizontal path in the last stage. From lemma 4.4.1 all computations have passed the same nodes. Every step of the computation passes a new node or two new nodes, depending on the tree structure of **C**. If n is the number of nodes in the graph not including $\rho_0$ and k is the number of nodes with two sons, then the total number of steps of an arbitrary computation is n-k.

# 5 Conclusions and Future Work

In this paper we have defined a nondeterministic transition system for a language with which dynamic linear arrays of processes can be specified. For each computation as defined by the transition system we can define a computation graph, in which the nodes are (snapshots of) processes, characterized by a state and the remainder of program still to be executed, and in which two kinds of connections feature, vertical and horizontal edges. The vertical edges correspond with the transitions within one process,the horizontal edges model the channels between the processes. Each 'horizontal path', consisting of horizontal edges only, corresponds with an intermediate configuration in the computation. We showed that it was possible to define the 'graph of all computations', **C**, the horizontal paths of which correspond exactly with the set of all intermediate configurations of all computations. The construction of this graph was based on a special fair right-to-left computation C(ρ). In this graph we have the property that, going downwards, the contents of all horizontal edges into the output receiver form a nondecreasing sequence of streams. We defined the operational semantics of a program as the least upper bound of this sequence of streams.

Using this graph of computations we were able to derive that all finite computations take the same steps, although possibly in a different order. All this proves the first half of the Kahn principle for linear dynamic networks, i.e. that every (fair) computation yields the same output. In order to prove the second half of the Kahn principle we have to show that this output stream equals the smallest solution of a system of equations to be derived from the initial configuration. This smallest solution can be obtained from a suitably defined denotational semantics, cf. [BB85, BBB93].

At the moment it is not clear which denotational semantics is the best choice. The semantics in [BB85] corresponds exactly with the operational semantics as defined here. It is however defined using CPO's. The paper [BBB93] offers a denotational semantics based on metric spaces. The advantage of such a semantics is that Banach's theorem can be used, the disadvantage is that in order to make the metric machinery work silent steps $\tau$ had to be introduced, which are not around in the operational semantics presented in this paper.

Several extentions of the results derived here come to mind. A natural idea is to study the general case introduced in [K74], i.e. to allow processes to expand into arbitrary networks, and to allow feedback loops, (sequences of) channels starting from and arriving at the same node. Apart from notational inconveniences, we do not see many problems. The advantage of using linear arrays of processes is that the graph of all computations can be depicted as two dimensional structure. In general case a configuration, a snapshot of the systems, will be a two dimensional graph in itself. This means that the new 'vertical edges' should now be drawn in the third dimension, and therefore computation graphs will be three dimensional. We expect that all our results will carry over this general case.

Another topic for future investigation is non-deterministic nodes. We expect that also for this case the graph of all computations can be drawn. However, the nice results that for each node in the graph its succesor(s) is (are) unique no longer holds, because a process now make a choice. Whether the notion 'graph of all computations' is useful in this setting remains to be seen.

**Acknowledgements.** Thanks go to Joost Kok, who offered a few essential pointers into the litterature on dataflow nets. We are also indebted to the other members of the Amsterdam Concurrency Group, headed by Jaco de Bakker, for their stimulating remarks during a presentation of our results.

## References

[AG78] Arvind and K.P. Gostelow, Some relationships between asynchronous interpreters of a data flow language, in: Formal description of programming concepts, W.J. Neuhold (ed.), pp.95-119, North-Holland, 1978.

[A81] A. Arnold, Sémantique des processus communicants, RAIRO Theor.Inf. 15,2, pp.103-139, 1981.

[BA81] J. Brock and W. Ackerman, Scenarios: a model of non-determinate computation, in: Formalization of programming concepts, LNCS 107,pp.252-259, Springer, 1981.

[BBB93] J.W. de Bakker, F. van Breugel and A. de Bruin, Comparative semantics for linear arrays of communicating processes, a study of the UNIX fork and pipe commands, in: A.M. Borzyzszkowski and S. Sokolowski (eds.) Proc. Math. Foundations of Computer Science, LNCS 711, pp.252-261, Springer, 1993.

[BB85] A.P.W. Böhm and A. de Bruin, The denotational semantics of dynamic networks of processes, ACM Trans. Prog. Lang. Syst, 7, pp.656-679,1985.

[B86] A. de Bruin, Experiments with continuation semantics: jumps, backtracking, dynamic networks, Ph.D. Thesis, Free University Amsterdam, 1986.

[B88] M. Broy, Nondeterministic data flow programs: how to avoid the merge anomaly, Science of Computer Programming, 10, pp.65-85, 1988.

[C72] J.M. Cadiou, Recursive definitions of partial functions and their computations, Ph.D. thesis, Stanford Univ., 1972.

[F82] A.A. Faustini, An operational semantics for pure data flow, in: Automata, languages and programming, 9th. Coll., LNCS 140, M.Nielsen and E.M. Schmidt (eds.), pp.212-224, 1982.

[JK90] B.Jonsson and J.N.Kok, Towards a complete hierarchy of compositional dataflow networks, in: Proc. theoretical aspects of computer software, LNCS 526, pp.204-225, Springer, 1990.

[J85] B. Jonsson, A model and proof system for asynchronous networks, in: Proc. 4th ACM PoDC, pp.49-58, 1985.

[KM77] G. Kahn and D.B. MacQueen, Coroutines and networks of parallel processes, in: Proc. IFIP77, B. Gilchrist (ed.), North-Holland, pp.993-998, 1977.

[K74] G. Kahn, The semantics of a simple language for parallel programming,in: Proc. IFIP74, J.L. Rosenfeld (ed.), North-Holland, pp.471-475, 1974.

[K78] P. Kosinski, A straight-forward denotational semantics for nondeterminate data flow programs, in: Proc. 5th ACM PoPL, pp.214-219, 1978.

[K86] J. Kok, Denotational semantics of nets with nondeterminism, in: European Symp. Programming, Saarbrücken, LNCS 206, pp.237-249, Springer, 1986.

[LS89] N. Lynch and E. Stark, A proof of the Kahn principle for input/output automata, Information and Computation, 82, 1, pp.81-92, 1989.

[SN85] J. Staples and V. Nguyen, A fixpoint semantics for nondeterministic data flow, JACM, 32 (2), pp.411-444, 1985.