Nets and Data Flow Interpreters

A. Rabinovich
B. A. Trakhtenbrot

School of Mathematical Sciences
Raymond and Beverly Sackler Faculty of Exact Sciences
Tel-Aviv University
Tel-Aviv, Israel 69978
e.mail: rabino@taurus.bitnet, trakhte@Math.Tau.Ac.IL

Abstract

We investigate and compare two ways of specifying stream relations (in particular - stream functions).

The first one uses relational programs, i.e. net-like program schemes in which the signature primitives are interpreted as relations over a given CPO. No stream domains are assumed; semantics is in fixed point style.

The second one is through data flow nets i.e., nets whose nodes are interpreted as processes (computational stations).

We prove the existence of an adequate data flow interpreter for relational programs over all relations (not only functional) and (essentially) its uniqueness. When dealing with functions the interpreter is modular and obeys the Kahn Principle.

On the other hand analyzing the deviations from Kahn's Principle we identify two kinds of anomalies:

The first one ("meagerness" anomaly) is caused by the defect of the used processes (computational stations) and holds in fact for arbitrary (even for very simple functional) input-output behaviors. This anomaly may always be avoided through the use of appropriate processes.

The second ("ambiguity" anomaly) is rooted in the semantics of relational nets over arbitrary CPO (and not specifically over stream domains). It is unavoidable in any extension beyond functional behaviors.

1. Introduction

We report on the further development of data flow semantics begun in [RT]; our main source of inspiration is in the well known papers by Kahn [Ka] and Brock-Ackerman [BA]. In [RT] we focused on stream functionals and on nets with determinate primitives, called there - unambiguous processes. It is our goal now to pursue the problem in the most general format, where nonfunctional stream relations may come out as input-output behavior of nets over non-determinate primitives. In doing so we need more insight into the syntax and semantics of languages for the specifications of input-output relations between streams.

Let us first recall how matters stand for stream functionals. Here Kahn's principle focuses on the fitness of two approaches, recursive equations and data flow networks. However, a more careful examination identifies four levels at which the specification of stream functionals is (or might be) dealt with.

At the first level one may start with a functional programming language $L_1$ which is sequential par excellence and has nothing to do with concurrency. $L_1$ may be an appropriately sugared version of ISWIM, i.e., λ-calculus with letrec-declarations and suitable constants; at this stage standard denotational semantics may be used. LUCID [WA] is a good illustration of such a language. Under some restrictions on the types in $L_1$ it is a matter of routine to describe a pure syntactic translation of $L_1$-programs into appropriate systems of recursive equations; these systems are just the programs of the next level language $L_2$. The translation from $L_1$ into $L_2$ has little to do with stream domains; as a matter of fact it is valid for a very broad class of...
domains. Note also that $L_2$-programs allow nice graphical representations which suggest the transition to the next level language, $L_3$. The role of programs in $L_2$ is played by data flow networks which reproduce exactly the graphical representations of the original $L_2$-programs. Moreover, the implicit parallelism of the languages $L_1$ and $L_2$ is now exhibited explicitly through communication and cooperation among the computing stations of the network. Kahn's principle just affirms that (under appropriate conditions) the translation from $L_2$ to $L_3$ is semantically correct, i.e. that the network provides an adequate interpreter [PA] of the original program. The semantics of data flow was originally formulated in an operational way, namely, in terms of a token game. However, it is easy to realize that this operational semantics is exactly captured by synchronization of the component processes (computational stations) and of the buffers associated with the channels of the network. This observation brings us to the next level which may be represented by an appropriate concurrent language $L_4$, say in the style of CCS-CSP [Hoare, Milner].

Let us now address the more general task of specifying input-output stream relations (many-valued functions). We observe first that $L_3$ and $L_4$ as above are still adequate for this task. What one needs are some kind of relational languages (call them $L'_1$, $L'_2$) which can do for stream-relations the same job that $L_1$ and $L_2$ did for stream functionals. At present we find it difficult to point to an appropriate candidate for $L'_1$ (perhaps experience in logic programming, or relational data bases and their generalizations could be helpful for developing such a language). But things look better for level 2. In [Br] M. Broy describes a simple net oriented language NET for the specification of stream functionals; this language relies on three fundamental constructs, sequential composition, parallel composition (aggregation), and feedback (looping); there are also some more combinators like permuters of input lines and of output lines, etc.. The syntax from [Br] may obviously be adapted for the specification of relations (and not just functions), but the semantics should be formulated anew. In particular, one has to look for a proper representation of functions and multivalued functions by means of relations. Our choice is to deal with specific relations (we call them observable I-O-relations) between n-tuples and m-tuples of finite elements of the domain under consideration; as to functions $f$, we represent them by corresponding approximation relations $R^f(x,y)$ = def $f(x) \geq y$. This way we come to our language $L'_2$. The present paper aims at a satisfactory explanation for the connection between $L'_2$ and $L_1$. We prove the existence (and in a natural sense the uniqueness) of an adequate data flow interpreter for the full language $L'_2$ (and not only for functional programs as was shown in [RT]). When dealing with functions the interpreter is modular and obeys the Kahn Principle.

As observed in [RT] the pomset (scenario) approach used in [BA] is not essential for the explanation of the Brock-Ackerman Anomaly; linear interleaving processes do the job as well. A further careful analysis of the Brock-Ackerman Anomaly brings us to the identification of two different kinds of anomalies:

The first one ("meagerness" anomaly) is caused by defects of the processes (computational stations) used and holds in fact for arbitrary (even for very simple functional) input-output behaviors. This anomaly may always be repaired through the use of appropriate processes (we call them fat processes).

The second ("ambiguity" anomaly) is rooted in the semantics of relational programs over arbitrary CPO (and not specifically over stream domains). This anomaly is unavoidable in any extension beyond functional behaviors.

In this sequel the exposition is organized as follows.

Section 2 deals with the conceptual background of those kind of relations we identify with (may be nondeterminate) behaviors of data flow networks. As in [RT] we proceed from the assumption that only finite behavior is observable; this view is in full agreement with domain-theory methodology. Accordingly, we define the notion of observable input-output relation and illustrate it through several examples. Though we do not consider aspects of infinite behavior, such as fairness, completeness of computations etc., we believe that these phenomena may be examined on the firm basis of observable relations. For observable relations we consider and extensively illustrate the crucial operation of looping which is intended to generalize the least fixed point operator on functions. It occurs, however, that unlike the functional case, simultaneous looping does not fit
Section 3 presents some of the process theory notation and terminology needed to make precise the formulations in the next sections. Part of this material is borrowed from [RT], to which we refer for more details and examples. But note that some important notions which are relevant for the proofs but not for the formulation of the results are omitted here and will appear in the full paper together with the proofs. Special attention is paid to the role of buffers and phenomena that appear around them. We pick out five facts which help explain the importance of considering buffered processes. This section may be omitted in the first reading and consulted only when needed for definitions (say of fat processes).

Section 4 presents the main results of this work, among them the two pairs of contrasting theorems which reveal the essence of two anomalies and the way they may (or may not) be avoided.

In the concluding remarks we comment about possible improvement of the results, about related and further research.

2. Relations

2.1. Observable Relations

Though the theory we are investigating is inspired by (and mainly developed for) stream processing there is no reason to focus on specific stream domains. So, we consider an arbitrary CPO domain D which obeys the Finiteness condition. Each element \( \eta \) in D is the least upper bound (lub) of a sequence \( \eta_1 \leq \eta_2 \leq \cdots \leq \eta_n \cdots \) of finite elements in D.

Recall that a finite element \( x \) is characterized by the following condition: assume that \( x \leq a \), where \( a \) is the lub of a sequence \( a_1 \leq a_2 \leq \cdots \); then \( x \leq a_n \) for some \( n \).

We shall extensively use flat domains (for example \( \text{BOOL} = \{ \text{true, false} \} \) ) and domains of streams.

The stream domain \( D = \text{STREAM} (\Delta) \) over a set \( \Delta \) consists of all finite and infinite strings over \( \Delta \), including the empty string. It is partially ordered by the relation \( "x \) is a prefix of \( y\)".

Obviously \( < D, \leq > \) is a complete partial order (CPO).

Clearly, flat domains and stream domains satisfy the finiteness condition.

For a monotonic and continuous function \( f \) from \( < D^*, \leq > \) to \( < D^*, \leq > \) one can define an approximation relation \( R_f \) between input \( n \)-tuples \( x \) and output \( m \)-tuples \( y \) of finite elements in \( D \).

\[
R_f(x;y) = \text{approx}(f(x) \geq y).
\]

It is easy to see that \( f = g \) iff \( R_f = R_g \) and that \( R_f \) has the following properties

(1) for every \( x \) there is a \( y \) such that \( R_f(x;y) \).

(2) \( R_f \) increases on inputs and decreases on outputs. Formally:

\[
R_f(x,y), x \leq x', y \leq y' \implies R_f(x', y').
\]

(3) \( R_f \) is functional if it is the approximation relation for a function from \( D \) into \( E \).

Warning about the types of relations. Though \( (D \times D) \times E \) is isomorphic to \( D \times (D \times E) \) these types are different. Even if a relation \( R \subset (D \times D) \times E \) and its image in \( R' \subset D \times (D \times E) \) may be considered as I-O relations they have to be dealt as with different I-O relations.
An observable I-O-relation $R$ may be extended to a full I-O relation $\hat{R}$ between arbitrary elements (not only between finite ones) through closure under limits. Just require that $\hat{R}$ be the minimal extension of $R$ such that

$$x_\alpha \rightarrow x, y_\beta \rightarrow y, \hat{R}(x_\alpha, y_\beta) \text{ imply } \hat{R}(x, y).$$

Clearly, $\hat{R}$ still obeys the conditions (2), (3) stipulated originally for observable I-O-relations. Define also the roof of an observable I-O-relation as follows:

$$\langle \text{roof } R \rangle (x, y) = \text{def } \hat{R}(x, y) \text{ and there is no } y' \text{ such that } y < y' \text{ and } \hat{R}(x, y').$$

Note that if $R$ is the approximation relation for a continuous function $f$ then roof $R$ is just the graph of $f$. In general roof $R$ may be considered as the graph of a many-valued function $f_R$: namely roof $R(x, y)$ holds for "complete" values of $f_R$ on the argument $x$.

In this sequel we use "observable relation" for "observable I-O-relation"

Example 1.

Let merge $(x_1, x_2; y)$ be the observable relation which holds for given finite streams $x_1, x_2$ if $y$ merges some prefixes of $x_1, x_2$. Then roof merge represents just the so called angelic merge [Park].

Later we shall make use of the following observable relations between $n$-tuples and $m$-tuples over BOOL. Notations like $R_{X}$ point on the type of the relation: $n, m$ are respectively the number of inputs and outputs. We include in the tables only some of the tuples, but they are chosen so that one can reconstruct the full table through input increasings and output-decreasings. Note also that $x$ stands for true or false (but not for $\bot$).

### Functional relations

<table>
<thead>
<tr>
<th>$R^1_\alpha (x_1; y_2)$</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_2$</td>
<td>0</td>
<td>$y_2$</td>
</tr>
<tr>
<td>$\bot$</td>
<td>$\bot$</td>
<td>false</td>
</tr>
<tr>
<td>$x$</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R^0_\alpha (\bot)$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>

### Ambiguous relations

<table>
<thead>
<tr>
<th>$G^1_c (c, x_1, x_2; y_1, y_2)$</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>$x_1$</td>
<td>$x_2$</td>
</tr>
<tr>
<td>true</td>
<td>$y_1$</td>
<td>$y_2$</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

### Unambiguous but nonfunctional relation

<table>
<thead>
<tr>
<th>$H^1_\alpha (x; y_1, y_2)$</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$y_1$</td>
<td>$y_2$</td>
</tr>
<tr>
<td>true</td>
<td>$\bot$</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>$\bot$</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
2.2. Looping

The definition of this operation is best illustrated and fully captured by the following examples.

**Looping without output hiding (LOOP):**

\[
\text{LOOP } R^3_{y_1} (x_1, y_2, y_1, y_2)
\]

is the relation \( S^3_{\lambda y_1 \eta_1 \eta_2} (x_2, \eta_1, \eta_2) \) which is defined as the conjunction:

\[
R^3_{y_1} (y_1, x_2, y_1, y_2) \land \text{Iter (appropriate parameters)}.
\]

The first conjunct points on \( y_1 \) as on a fixed point of \( R^3_{y_1} \), whereas the second one (when equipped with appropriate parameters) states that this fixed point is inductive, i.e., it is achievable via iterations. In more details \( \text{Iter} \) express the following: There exists a finite sequence of finite approximation for \( y_1 \) (remind that \( y_1 \) itself is finite)

\[
\downarrow = y_1^0 \leq y_1^1 \leq \ldots \leq y_1^n = y_1
\]

such that for each \( i < n \)

\[
R^3_{y_1} (y_1^i, x_2, y_1^{i+1}, \downarrow) \quad (**)
\]

**Comments**

1. Below is an equivalent definition of \( \text{LOOP} \). (Note that checking this equivalence essentially relies on the input-increasing and output-decreasing properties of observable relations.)

\[
S^3_{\lambda y_1 \eta_1 \eta_2} (x_2, \eta_1, \eta_2) \text{ if there exists an increasing sequence of approximations}
\]

\[
\{y_1^i, y_2^i, \sigma^i \}_{i=1}^n \quad \text{for } y_1, y_2, \sigma \text{ such that } \downarrow = y_1^0, \quad y_1^n = y_1, \quad y_2^n = y_2, \quad \sigma^n = \sigma \text{ and}
\]

\[
R^3_{y_1} (y_1^i, x_2, y_1^{i+1}, \downarrow) \quad \text{for } i < n
\]

2. Simultaneous looping is defined in a very similar way. For example

\[
S^3_{\lambda y_1 \eta_1 \eta_2} (y_1, x_2, y_1, \eta_2) = \text{LOOP } R^3_{y_1} (x_1, x_2, y_1, y_2)
\]

holds if there exists an increasing sequence of approximations \( \{y_1^i, y_2^i, \sigma^i \}_{i=1}^n \) for \( y_1, y_2, \sigma \) such that \( \downarrow = y_1^0, \quad \downarrow = y_2^0, \quad y_1^i = y_1, \quad y_2^i = y_2, \quad \sigma^i = \sigma \) and

\[
R^3_{y_1} (y_1^i, y_2^i, \sigma^i, y_1^{i+1}) \quad \text{for } i < n
\]

3. Looping of observable relations returns observable relations; moreover it is consistent with the least fixed point construct on functions. Let \( f \) be a function from \( D \times D \) into \( D \) let \( R_f (x, y) \) be its approx-

**Looping with hiding of output (loop):**

\[
\text{loop } R^3_{y_1} (x_1, x_2, y_1, y_2)
\]

In this case both \( y_1 \) and \( x_1 \) are bound and the result \( S^3_{\lambda y_1 \eta_1 \eta_2} (x_2, \eta_1, \eta_2) \) of the looping is defined as

\[
S^3_{\lambda y_1 \eta_1 \eta_2} (x_2, \eta_1, \eta_2) \supseteq \exists \eta_1 S^3_{\lambda y_1 \eta_1 \eta_2} (x_2, \eta_1, \eta_2).
\]

**Example 2.**

For the relations of Fig. 1-3, it is easy to check that

\[
\text{loop } R^3_{y_1} (x_1, x_2, y_1, y_2) = R^3_{y_1} (x_2, y_2)
\]

\[
\text{loop } R^3_{y_2} (x_2, y_2) = R^3_{y_2} (x_2, y_2)
\]

\[
\text{loop } R^3_{y_1} (x_1, x_2, y_1, y_2) \land \text{loop } (\text{loop } R^3_{y_1} (x_1, x_2, y_1, y_2))
\]

3. Processes

**Action alphabet.** Each process \( P \) is equipped with a set \( \text{alph}(P) \) - the alphabet of the actions which are available in this process. Note that the runs of \( P \) use actions only from \( \text{alph}(P) \), but not necessarily all of them.

A linear process is a prefix closed set of action-strings.

For linear processes the definition of synchronization is very simple:

\[
s \in \text{Synch} (P_1, P_2, \ldots) \text{ iff for all } i
\]

\[
s | \text{alph} (P_i) \subseteq P_i
\]

where \( s | A \) is the notation for the string one gets from the string \( s \) omitting all the actions that are not in \( A \).

3.1. Input-Output Processes

When dealing with data flow one has to be more specific about the action alphabet of the processes. Here are the relevant stipulations:

**Communications.** The action alphabet is structured as a Cartesian product \( CH \times DATA \). An action \( \langle \text{ch, } d \rangle \) is said to be a communication through channel \( \text{ch} \) which passes the data value \( d \). Also a partition of \( CH \) into \( CH_{\text{inp}} \) (input channels), \( CH_{\text{out}} \) (output channels) and \( CH_{\text{int}} \) (internal channels) must be explicitly displayed.
We call this partition the type of the process. Correspondingly we use the terminology: external type (for the ports \(CH_{\text{in}} \cup CH_{\text{out}}\)), internal type, etc.

The status of input, output and internal channels in a data flow network will be formalized in detail below (see 3.2) in terms of buffering.

Without loss of generality we assume below that \(DATA\) is fixed, so we have only to specify the types of the processes under consideration. If \(CH_{\text{in}}=[I_1, \ldots, I_n]\) and \(CH_{\text{out}}=[O_1, \ldots, O_m]\) we refer to the process as to a \((I_1, \ldots, I_n, O_1, \ldots, O_m)\)-process. A linear \(I-O\)-process is a prefix closed set of finite communication strings and these strings inherit the type of the process. In other words, everywhere below when a string is considered we assume that it is equipped with a type of channels; often we refer only to the external channels of the type.

**Equivalences and Approximations between Processes.**

First agree about the following notations and terminology.

(i) For a string \(s\) and a subset \(CH'\) of the set of channels let \(s \setminus CH'\) be the string which results when all the communications through channels not in \(CH'\) are deleted from \(s\). In particular \(s \setminus \text{ext}=\text{def} s \setminus (CH_{\text{in}} \cup CH_{\text{out}})\).

Define external equivalence of strings:

\[
s_1 \sim_{\text{ext}} s_2 \iff s_1 \setminus \text{ext}=s_2 \setminus \text{ext}.
\]

Accordingly we write \(x \in_{\text{ext}} P\) if \(x\) is externally equivalent to a string in \(P\).

For 1-O processes \(P_1, P_2\) the relation \(P_1 \sim_{\text{ext}} P_2\) holds if they have the same external type and

\[
(*) \quad [s \setminus \text{ext} : s \in P_1]=[s \setminus \text{ext} : s \in P_2]
\]

\(P_1 \leq_{\text{ext}} P_2\) (\(P_1\) externally approximate \(P_2\)) holds if \((*)\) is replaced by:

\[
(**) \quad [s \setminus \text{ext} : s \in P_1] \subseteq [s \setminus \text{ext} : s \in P_2]
\]

(ii) Given a string \(s\) and a sequence \(CH'=(ch_1, \ldots, ch_k)\) of channels, we use the notation \(\text{stream}(s, CH')\) for the sequence <\(s \mid ch_1, \ldots, s \mid ch_k\)>.

Now, we define the relational behavior \(\text{rel}(P)\) of a 1-O-process \(P\), relational equivalence \(=_{\text{rel}}\), and relational approximation \(\leq_{\text{rel}}\) for processes:

\[
\text{rel}(s) = \langle \text{stream}(s, \bar{I}), \text{stream}(s, \bar{O})\rangle
\]

\[
P_1 =_{\text{rel}} P_2 \iff \text{rel}(P_1)=\text{rel}(P_2)
\]

\[
P_1 \leq_{\text{rel}} P_2 \iff \text{rel}(P_1) \subseteq \text{rel}(P_2)
\]

Obviously \(=_{\text{ext}}\) implies \(=_{\text{rel}}\) and \(\leq_{\text{ext}}\) implies \(\leq_{\text{rel}}\).

We say that a process is unambiguous (respectively functional, or ambiguous) if its I-O-behavior is an unambiguous (respectively functional, or ambiguous) relation.

### 3.2. Buffering

**Definition (Buffer).** \(\text{buf}(I; O)\) is the set of strings over \((I, O) \times DATA\) such that \(s\) is in \(\text{buf}(I; O)\) if for every prefix \(s_1\) of \(s\) the projection of \(s_1\) on the channel \(O\) is a prefix of projection of \(s_1\) on the input channel \(I\).

**Definition (Buffering of a Process on an channel \(Ch\)).**

Given a process \(P\) with an external channel \(Ch\):

a) rename in \(P\) the channel \(Ch\) to a fresh channel name \(J\)

b) If \(Ch\) is an input channel synchronize the resulting process with \(\text{buf}(Ch; J)\)

If \(Ch\) is an output channel synchronize the resulting process with \(\text{buf}(J; Ch)\).

![Fig. 4]

**Notation.** \(\text{Buf}(P)\) designates the result of buffering \(P\) on all external channels. We call this operation buffering of \(P\).

It is easy to see that \(\text{Buf}(\text{Buf}(P))=_{\text{ext}} \text{Buf}(P)\)

**Definition.** A process \(P\) is buffered if \(P=_{\text{ext}} \text{Buf}(P)\).

In order to give a combinatorial characterization of buffered processes let us first define some operations on strings.

a) **Input extension of \(s\).** Extend \(s\) appending to the right arbitrary many input communications.

b) **Input (output) permutation.** In the string \(a_1 \cdots a_k a_{k+1} \cdots\) permute the adjacent \(a_k, a_{k+1}\) if they are both input (output) communications which refer to different channels.
c) **Input anticipation.** Permute the adjacent $a_k, a_{k+1}$ whenever $a_{k+1}$ is an input communication and $a_k$ is an output communication.

Claim (Combinatorial characterization). A process is buffered iff its projection on the external channels is closed under input extension, input and output permutations and input anticipation.

The importance of buffers and buffered processes.

Consider an arbitrary process $P$; its I-O-behavior $rel(P)$ is not necessarily an observable I-O-relation (it may violate the input-increasing or output-decreasing properties). But for every buffered process its I-O-behavior is an observable relation over streams. Moreover, for arbitrary process $P$ there holds:

1. $rel(Buf(P))$ is the minimal observable I-O-relation which extends $rel(P)$

Let $N$ be a data flow net. Assume that $N'$ is obtained from $N$ by replacing some component process $P$ of $N$ by its buffering $Buf(P)$; then $N$ and $N'$ have the same I-O-behavior. So,

2. The input-output behavior of a data flow network is not sensible to the replacement of its component processes ("computational stations") by their bufferings.

That is why without loss of restriction we may consider only data flow networks over buffered processes.

Let $R$ be a relation over streams and let $s$ be a string. We say that $s$ is consistent with $R$ if for every prefix $s_i$ of $s$ the stream generated by $s$ on inputs is $R$-related to the stream generated on outputs, i.e., $R(\text{stream}(s_1, p^i), \text{stream}(s_1, q))$.

**Definition.** A process $P$ is said to be fat if its buffering consists of all strings which are consistent with $rel(P)$.

A process is meager if it is not fat.

**Extremality of fat processes.** A buffered process $P$ is fat iff for all processes $Q$ $Q = rel P$ implies $Q \leq_{\text{ext}} P$.

It is obvious from the definitions that

3. for every buffered process $P$ there is a fat process $P'$ with the same I-O-behavior as $P$.

(4) for every observable relation $R$ over streams there is a fat process $P$ such that $rel(P) = R$; Moreover there is a unique (up to $=_{\text{ext}}$) buffered fat process (we denote it by $\text{fat}(R)$) such that $rel(\text{fat}(R)) = R$.

**Comment.** If $R$ is the approximation relation for a function then the process $\text{fat}(R)$ consisting of all string consistent with $R$ is smooth in the sense of [RT].

In process theory one uses sometimes the operation looping [Den, PS] on a process $P$. However, this is actually a derived operation;

5. Looping of a process $P$ may be expressed by appropriate synchronization of $P$ with buffers.

4. Nets of Relations versus Nets of Processes

4.1. Syntactical Provisos

For simplicity we consider only finite nets. Let $N$ be a net; its arcs (channels) are divided into input, output and internal channels.

Fig. 5 suggests itself. $p^3_1, p^0_1$ are names for nodes of the appropriate types. For example $p^3_1$ has three input channels (one of them is the only input channel of the net) and three output channels (two of them are the output channels of the net). Actually such a net is a piece of syntax and one can impose semantics on nets via appropriate interpretation of the nodes.
A net may be composed from more elementary nets through three fundamental constructions.

a) Aggregation: (notation ||) several nets are viewed as a single net but there is no connection between them.

b) Looping: amounts to connecting some output channels to some input channels. (For simplicity we assume that different output channels are connected to different input channels and vice versa).

c) Sequential composition: (notation - semicolon) connects output channels of one net to input channels of the other. It can be viewed as an aggregation followed by an appropriate looping, so one could manage without it.

The standard composition of a net \( N \) prescribes that atomic nets (single nodes with input and output channels) be aggregated with subsequent connection through simultaneous looping.

Textual representations for nets with prescribed composition are provided by programs of the language NET [Br], which uses the three net-constructs above and some other useful combinators. For example, the following are some possible compositional prescriptions for the net on Fig. 6. (the last prescription is the standard one).

\[
\text{loop } (C^0_1;G^3_3)
\]
\[
C^0_1;\text{loop}(G^3_3)
\]
\[
\text{loop } (C^0_1||G^3_3)
\]

Here in order to avoid cumbersome notations we deviate from the full NET formalism. In this sequel we use the notations : \( \pi, \rho \) for programs in (our coarsened version of) NET, \( N_{\pi}, N_{\rho} \) for the net prescribed by \( \pi \) and \( \rho \); \( \pi=\text{net} \rho \) means that the programs \( \pi, \rho \) prescribe the same net (i.e., \( N_{\pi}=N_{\rho} \)).

4.2. Semantical Provisos

Below, by abuse of notation we use \( \pi, \rho, \pi=\text{net} \rho, N_{\pi}, N_{\rho} \) for interpreted nets and relational programs as well.

4.2.1. Data Flow Nets

In the data flow approach the nodes \( P^3_3, P^0_1, \ldots \) are interpreted as "computational stations" \( P_1, P_2, \ldots \) of the respective types. In this paper the formalization of computational stations is in terms of I-O linear (i.e., pure interleaving) processes. Data flow semantics assigns observable input-output relations \( R_1=\text{rel}(P_1), R_2=\text{rel}(P_2) \) of the proper types to the component processes. It assigns also to the net as whole a global process \( Pr(N) \) and a global relation \( \text{rel}(N) \). Both may be defined operationally in terms of a token game as suggested in the original work of Kahn [Ka], (see also [Den]). On the other hand it is well known that:

1. the process \( Pr(N) \) may be obtained by synchronization of the processes assigned to the nodes of the net and of the buffers assigned to the arcs of the net. The input (respectively output, internal) actions of \( Pr(N) \) are actions on the input (respectively output, internal) channels of the net.

2. \( \text{rel}(N) \) as defined by the token game coincides with the relation assigned to \( Pr(N) \).

It follows (see section 3) that the Input-Output behavior of a data flow network is not sensible to the replacement of its component processes ("computational stations") by their bufferings. That is why without loss of restriction we consider only data flow networks over buffered processes. The crucial point about data flow
semantics is that this semantics does not depend on the textual representation $p$ and is uniquely determined by the net $N$ (or what is the same by its standard composition). Hence, it is convenient to ignore the specific textual representation and to refer directly to $rel(N(P_1,P_2))$.

**Fact 1.** $P_1 = rel Q_1$ does not necessarily imply $rel(N(P_1)) = rel(N(Q_1))$.

**Fact 2.** There exists $sup(rel(N(Q)))$ over all $Q$ with $rel(Q_i) = R_i$. Moreover this supremum is reachable if the $Q_i$ are fat.

**Consequence.** On a schematological level each net $N$ specifies an operation (a functional) on relations over streams (of the corresponding types).

### 4.2.2. Relational Programs

Consider NET-programs $\pi, \rho \ldots$ whose atoms $A_{n,m}^k$ $(n,m=0,1,\ldots)$ are interpreted as observable relations with $n$ inputs and $m$ outputs. The semantics assigns in a denotational (compositional) way observable relations of the proper types to relational programs $\pi, \rho$. The semantics of aggregation is evident. For example

$$Rel(P_1 \| P_2)(x_1 x_2 x_3 y_1 y_2 y_3 y_4) = def \exists z_2. (Rel(P_1)(x_1 y_1 y_2) \& Rel(P_2)(x_2 z_2 y_3 y_4)).$$

The semantics for looping was already explained in section 2 and the semantics of sequential composition is very simple. For example (see Fig. 7)

$$Rel(P_1 \triangleright P_2)(x_1 y_1 y_2) = def \exists_{1,2} \exists z_2. (Rel(P_1)(x_1 z_2, y_2) \& Rel(P_2)(z_2 x_1 y_3 y_4)).$$

Referring to the Examples 1 and 2 one can see that $R_{\tau} = (C^{G_{\tau}})$ such that $R_{\tau} = (C^{G_{\tau}})$.

**Fact 1'.** $N_{\pi} = N_{\rho}$ does not necessarily imply $rel(\pi) = rel(\rho)$

Proof: Recall the example in section 2, where

$$loop \ R_{\tau} \neq \exists x_1 y_1 y_2. (Rel(\pi)(x_1 x_2 y_1, y_2) \& Rel(\rho)(x_1 x_2 y_1, y_2)).$$

**Fact 2'.** There exists $inf Rel(\pi)$ over all programs $\pi$ which prescribe a given net $N$. Moreover this infimum is reachable by the standard prescription for this net $N$.

From now on we focus on standard relational programs $\pi, \rho$ and call them also relational nets, having in mind the corresponding interpreted nets $N_{\pi}, N_{\rho}$. We consider also relational net contexts i.e. nets in which not all nodes (atomic subformula) are interpreted. The uninterpreted nodes are called the holes of the context. Actually in this paper we shall need only contexts with one hole. The notations for such a context and for the relational net which arises when the unique hole is replaces by a relation $R$ are respectively $N[R]$ and $N[N]$. A net (a context) is called functional if all its (interpreted) nodes are functional. A relational net $N$ is said to be robust if all relational; programs $\rho$ with $N_{\rho} = N$ have the same semantics (as the standard program).

### 4.3. Main Results

We formulate first some important facts about the semantics of relational nets

Due to the well-known theorem about the equality of nested and simultaneous fixed points for continuous function one can deduce

**Fact 3.** The meaning of functional net is also functional. It can be characterized also as the least solution of the Kahn system of equations which is associated with the net.

Hence:

**Theorem (Robustness).** Nets over functional relations are robust.

This fact sharply contrasts with

**Theorem (Ambiguity anomaly).** Given an arbitrary ambiguous relation $R$, there is a functional context $N_{\pi/R}$ such that the meaning of $N_{\pi/R}[R]$ is a functional relation and $N_{\pi/R}[R]$ is not robust.

Let $C$ be a class of observable relations closed under the net constructs. Say that $C$ is modular if for nets $N, N', N''$ over relations in $C$ the following condition holds:

By replacing in net $N$ any subnet $N'$ by a net $N''$ with the same meaning as $N'$ one preserves the meaning of the whole net.

As consequences from these theorems we have the following facts:

**Fact 4.** The class $F$ of functional relations is modular.

**Fact 5.** Let $C$ be a closed (under the net construct) class of relations; if $C \supseteq F$ and $C$ con-
tains an ambiguous relation then modularity fails for it.

All the facts above hold for observable relations over arbitrary domains. Now, let us assume that the underlying domain is actually a stream domain. Under this assumption we are going to clarify how specifications via data flow and relational nets correlate with each other.

We classify processes into \textit{fat} and \textit{meager} (see Section 3). For each observable relation \( R \) there are both meager and fat processes with \( R \) as their input-output-behavior.

Given a net \( N \) interpreted as a relational net \( RN \), consider the corresponding data flow network \( PN \) such that the process assigned to a node \( p \) in \( PN \) computes the relation assigned to this very node in \( RN \). \( PN \) is called a data flow interpreter for \( RN \). We say that \( PN \) is an adequate data flow interpreter for \( RN \) if \( rel(PN)=Rel(RN) \).

**Theorem (Meagerness Anomaly).** Let \( P \) be an arbitrary meager process and let \( R \) be its 1-O behavior. Then there exists a functional net context \( N_p[ ] \) such that every its data flow interpreter in which \( R \) is interpreted by \( P \) is not adequate.

This anomaly may be avoided if no use is made of meager processes.

**Theorem (Adequate Data Flow Interpretation).** Assume that in \( PN \) only fat processes are chosen; then it provides an adequate data flow interpretation for \( RN \).

Let us now summarize and complete our considerations above. Assume that we start with a given net \( N \) and a given assignment of component relations to the nodes of \( N \). Actually we dealt with two ways of specifying a resulting relation:

The first is through relational programs which may differ from each other in how they prescribe the construction of the net \( N \).

The second is through data flow networks which may differ from each other through the choice of processes (computational stations) for the component relations.

Now, the message we get from Facts 2, 2’ and the Adequateness Theorem is reflected in

**Theorem (Min-Max).**

1. Among relational programs the standard one provides the minimal relation.
2. Among the data flow interpreters the \textit{fat} one provides the maximal relation.
3. The minimum provided by relational programs coincides with the maximum provided by interpreters.

5. Concluding Remarks

5.1. Looping of Processes and Looping of Relations

Looping of processes is expressed via synchronization with buffers (see 3.2 (5)). Therefore, looping of processes is a robust operation in data flow. On the other hand looping of relations as defined in section 2 is not robust. One might suspect that this is a consequence of the particular definition we used; but that is not the case, as we can prove the following:

It is impossible to define an operation on relations (call it \textit{your-favorite-loop}) such that:

1. When applied to functional relations it provides the least fixed point (as explained in comment (3) section 2.2)
2. It is consistent with sequential composition and aggregation (as explained in section 4.2.2).
3. Simultaneous and nested \textit{your-favorite-loop} provide the same result.

5.2. Uniqueness of the Adequate Universal Interpreter

A function \( U \) from observable relations to processes is said to provide an Universal Interpreter if \( rel(U(R))=R \) for all observable relations \( R \). It provides an adequate universal interpreter if for every relational net \( RN \) the corresponding data flow net \( PN \) has the same meaning. One can show that \textit{fat} (see 3.2 (4)) is unique in the following sense

1. \textit{fat} is an adequate universal interpreter.
2. For every adequate universal interpreter \( U \)

\[
\text{Buf}(U(R))= \text{extfat}(R)
\]

5.3. Are Infinite Streams Relevant?

Most of our arguments do not presume essentially processing of \textit{infinite} streams. Hence, the following modification (considered as an illustrative example) makes sense:
Consider only data streams of a fixed finite length $k$ and the corresponding data flow version which is supported by restricted buffering (buffers of capacity $k$). It is an easy exercise to adapt most of our results to this model. In particular, taking $k=1$, i.e., reducing buffers to simple delays, one can exhibit Kahn's principle and both anomalies for asynchronous boolean circuits.

5.4. The Fairness Approach

In our investigation we found it convenient to use the machinery of observable relations between finite elements of CPO's, which is tailored along the usual domain theoretical approach. Other authors [LS] prefer to rely on complete computations and on fairness arguments. However it is not hard to show that our techniques may be adapted to prove some similar results for the fair computation format.

5.5. About Fixed Point Reasoning

Though the theorem about adequate data flow interpretations is quite general we are feeling uneasy with it when dealing with nonfunctional behavior. Anxiety is caused by the inherent lack of robustness and modularity in relational nets; this makes hard reasoning about nonfunctional behavior. May be the remedy is in alternative approaches, which replace a relation by a bunch of functions that uniformize this relation [Br],[Pa].

Acknowledgement. We are grateful to Albert Meyer for hospitality and helpful discussions.

6. References


