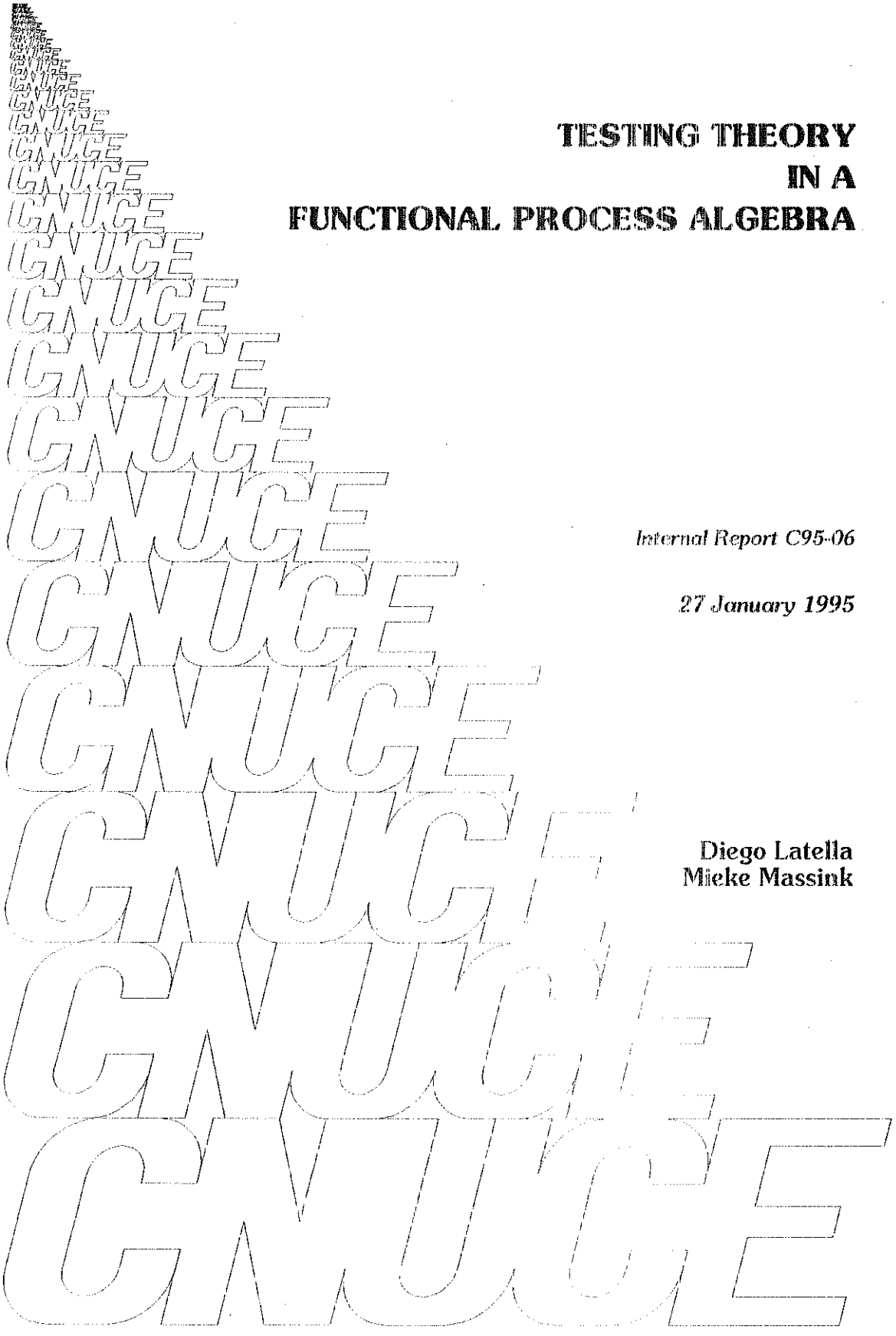

TESTING THEORY IN A FUNCTIONAL PROCESS ALGEBRA

Internal Report C95-06

27 January 1995

Diego Latella
Mieke Massink

TESTING THEORY IN A FUNCTIONAL PROCESS ALGEBRA

Internal Report C95-06

27 January 1995

Diego Latella
Mieke Massink

Testing Theory in a Functional Process Algebra*

Diego Latella † Mieke Massink ‡

Abstract

In this paper we present an algebra of processes with input and output. The key point is the conceptual distinction between input and output actions. We give an operational semantics and a notion of testing where the experimenter is not allowed to restrict the output possibilities of the process under test, as it instead could be the case if the standard *synchronization*-based approach of process algebra would be used. We also give two denotational models, one based on a kind of *acceptance trees* and the other based on sets of *functions* and we prove them *fully abstract* with respect to the testing equivalence induced by the new notion of experimenter. The model based on sets of functions provides a simple and powerful way for combining the process algebra approach with the functional one. In process algebras notions for dealing with non-determinism and progress properties of processes - deadlock, etc - has been studied deeply. In the functional approach clean proof styles - like the transformational one - and proof techniques - like induction principles for minimal as well as maximal fixpoints - have been proved successful. The combination of these approaches gives an interesting framework for the specification and verification of concurrent systems.

KEYWORDS: Process Algebra, Operational Semantics, Testing Equivalence, Acceptance Trees, Denotational Semantics, Functional and Data-flow Systems, Stream-Processing Functions.

1 Introduction and Related Work

A number of formal models and languages for the specification of concurrent systems and for formally reasoning about them have been proposed in the literature [Rei85, Hen88, Hoa85, Mil89]. Many of them are based on the notion of process algebra and are provided with an operational semantics and various notions of equivalences [vG90].

In the theory of formal semantics for sequential (i.e. non-concurrent) systems, a functional approach has shown many advantages. This is mainly due to its sound and well understood mathematical basis which allows for the direct use of rigorous proof styles like transformational reasoning [DC89]. Moreover, when objects are defined as fixed-points of recursive equations, suitable induction principles [BW88, MNV73] can be used.

*This work has been partially supported by C.N.R. - Italy (Progetto Coordinato "Metodologie, Architetture, Ambienti di Progetto e Valutazione per Sistemi di Elaborazione Distribuiti" and Progetto Bilaterale "Estensioni Probabilistiche dell'Algebra di Processi LOTOS, Basate su Strutture di Eventi per la Specifica e Analisi Quantitative di Sistemi Distribuiti") and N.W.O. - NL (Project num. 612-16-023)

†C.N.R. - Ist. CNUCE - Via S. Maria 36 - Pisa - Italy - phone:+39 50 593230 - fax: +39 50 904052 - e-mail: d.latella@cnuce.cnr.it

‡University of Nijmegen - Toernooiveld 1 - 6525 ED Nijmegen - The Netherlands - phone:+31 80 653289 - fax:+31 80 553450 - e-mail:mieke@cs.kun.nl

In [Kah74] a fixed-point semantics for a class of distributed systems has been proposed. The behaviour of a system is modeled as a function from sequences (sometimes called "streams") of input messages to sequences of output messages. Of course such an approach applies only to deterministic specifications of system behaviour. On the other hand, using non-determinism as a way of modeling is essential when reasoning about concurrent systems, so, recently, interesting approaches have been proposed for extending stream semantics with non-determinism [Abr84, Abr89, Bro90, Bro92a, Bro92b, Dyb86, DS89, Mis90, Ong93, San92].

In particular we consider Broy's and Sander's proposals very promising. In these approaches a specification is a set of continuous sequence processing functions which is by itself defined as a higher order boolean function. In the approach of Broy a number of combinators on specifications are defined which reflect the architectural aspects of the system. The main combinators in this approach are based on function composition (pipeline or cascade), parallel composition and feedback. Central in this work are notions of refinement of specifications which are compositional with respect to these combinators. In Sander's approach the topology of the system is given by a higher order function that is defined as the minimal fixed point solution of a set of recursive equations. Central to this approach is a higher order logics for verifying that a function satisfies a given specification.

Of all the above mentioned works only [Ong93] addresses the issue of algebraic relations between specifications that take progress properties into account (deadlock etc.) like observational or testing equivalences do for process algebras. Indeed, low expressive power w.r.t. progress properties in the presence of non-determinism has been an argument against stream based approaches in the concurrency theory community. In particular, it is commonly maintained that such approaches can be *at most* as much powerful as *trace semantics* in the context of process algebra. Indeed, proposals for trace semantics for data-flow networks have appeared in the literature, like [Abr89, Mis90, Jon87]. In [Ong93] a notion of Testing is introduced that is shown to be equivalent to applicative bisimulation which is a variant of observational bisimulation defined by Milner [Mil89]. This notion of Testing takes the internal structure of process expressions into account.

In our opinion, instead, Hennessy Testing Equivalence [Hen88] looks more appealing since it is only based on externally observable behaviour of processes.

In this paper we start by defining, in Section 3, a simple process algebra of *input-output processes*, PA_{io}, where a clear conceptual distinction is made between input messages and the output systems generate as a reaction to such inputs.

Such a distinction is quite natural when modeling a wide variety of concurrent systems, although it is usually disregarded in the process algebra approach. Moreover, keeping the distinction clear results into a step forward in the harmonization of the functional approach to concurrency and process algebra.

As just stated above, the functional approach can offer several advantages. On the other hand, we think that an approach to the study of concurrent systems which would disregard process features like *progress* properties and which would not allow to compare processes on the basis of such features would not be that much useful.

The operational model of PA_{io} is that of *Transition Systems* where transitions are labeled by input-output pairs. This, in principle, could allow the application to PA_{io} of the theory of concurrency based on transition systems provided that all actions are interpreted as input-output pairs. Unfortunately, our experience shows that this is indeed *not* the case; at least it is not the case for Testing Equivalence: there are PA_{io} expressions of which the transition systems are *not* Testing Equivalent in the Hennessy sense but which we would like to consider

equivalent in the light of the conceptual distinction between input and output mentioned above.

In Section 4 we introduce a new notion of *experimenter*. The essential point is that an experimenter can only *observe* the output produced by the system under test (and then behave according to it). In particular the observer cannot constrain such an output in any way and it can influence it only by means of the input *previously* provided to the system. In other words, it is possible that for the same input, the description of the system may give as an output one out of *many* possible outputs, due to internal non-determinism. The choice of the actual output should obviously not be influenced in any way by the experimenter. Based on this notion of experimenter we define our notion of experimental system and the related Testing Equivalence, which we call *I/O-Testing Equivalence*. A denotational semantics of PA_{io} can be given which maps expressions into *Finite Functional Acceptance Trees*, a slight modification of Hennessy Finite Acceptance Trees. This model is shown *fully abstract* with respect to I/O-Testing Equivalence.

In Section 5 we introduce FA, a functional model for the specification of concurrent systems, and we state the main result relative to bridging the functional approach and process algebra, namely that FA is *fully abstract* for PA_{io} with respect to I/O-Testing Equivalence.

For the functional model we follow the approach of Broy: a possible behaviour of a system is modeled by a *continuous* function from sequences of input messages to sequences of output messages, while a system is described by a *Functional Specification*, i.e. a predicate over sequence processing functions.

We propose a set of combinators on sequence processing functions and on functional specifications which are strongly inspired by process-algebraic ones and which together constitute a *Functional Algebra* which allows for the systematic and compositional construction of functional specifications.

Finally in Section 6 some conclusions are drawn and some hints for further research are outlined.

In the present paper we give a quite informal treatment of the subject. The reader interested in formal definitions and detailed transformational proofs of the results stated here is referred to [LMP94, Mas95].

2 Notation

In this paper we use a *Funmath*-like notation. Funmath (*Fun*ctional *Math*ematics) is a formalism that is based mainly on the mathematical concepts of function and sets and on predicate calculus allowing for a transformational proof style. In the following we give a short informal introduction to those features of Funmath which we use in this paper. A more detailed description of the notation can be found in [Bou92, Bou94].

Funmath has four constructs; identifiers, function application, tuple denotation and abstraction.

Besides predefined constants, new constants can be introduced using a definition of the form: **def** $a : A$ **with** P. In this definition a denotes a single identifier or a tuple of identifiers, A is an expression denoting a type, and P is a defining proposition. The existence and uniqueness of the newly defined constant must be proven by the definer. Recursive definitions are allowed. In this context they are intended in the fixpoint/domain-theoretic interpretation. Function application denotes an object, $f\ x$, as the image of an object, x, under a function,

f. Partial application allows for functions returning functions as result.

A tuple denotes a function with domain $\{0,..,n-1\}$ for n in $\mathbb{N}$. For example: a,b,c denotes the function $(a,b,c) \in \{0,1,2\} \to \{a,b,c\}$ such that $(a,b,c)0 = a$, $(a,b,c)1 = b$, and $(a,b,c)2 = c$. Notice that in the above formula $\{0,1,2\}$ denotes the range of the function $0,1,2$. That is, curly brackets together, $\{\}$, denote the operator *range* on functions. Normal brackets "(" and ")" are instead used for grouping in case of ambiguity. The tuple consisting only of the element a is denoted by τa. Considering tuples as functions is extremely useful in transformational reasoning. A small disadvantage is however that in this way we lost the traditional notation for the one-element set. We will write ιa when we want to denote the one-element set containing only a. Given set M, the set of n-tuples over M is denoted by M^n; for $n = 0$ we have $M^0 = \iota\epsilon$, that is the set containing only the empty tuple, i.e. the tuple with empty domain. $M^* = \bigcup_{n\geq 0} M^n$ is the set of all finite sequences over M. M^∞ denotes the set of infinite sequences over M, i.e. the total functions from natural numbers to M, and we let $M^\omega = M^* \cup M^\infty$. On sequences we shall also use the usual constructor, which we will denote by $\succ$:

def $- \succ - : M \to M^\omega \to M^\omega$
with $(x \succ \sigma)\, j =$ *if* $j = 0$ *then* x *else* $\sigma\ (j-1)$ *fi*

We define sequence concatenation as follows:

def $- +\!\!+ - : M^\omega \to M^\omega \to M^\omega$
with $(\sigma_1 +\!\!+ \sigma_2)\, j =$ *if* $j < (\#\sigma_1)$ *then* $\sigma_1 j$ *else* $\sigma_2(j - (\#\sigma_1))$ *fi*

where $\#\sigma$ is the lenth of σ

The only kind of abstraction we use in this paper is of the form $x : X.E$ where x is a single identifier or a tuple of identifiers, X is an expression denoting a type. The semantics of this formula is that it denotes the function mapping x (in X) to E. For shortness we sometimes leave out the type information X in formulas if no confusion can arise.

All other necessary constructs are defined by means of the previous four. For instance, the mathematical formula $\forall x \in X.Px$ in Funmath to be interpreted as $\forall(x : X.Px)$ where $\forall$ is a function over boolean functions, which is applied to the abstraction $x : X.Px$. Thus, the way constructs are defined coincides exactly with the common mathematical notation, and also the mathematical interpretation of those formulas remains the same. The advantage becomes clear when manipulating formulas in a transformational proof style.

3 The Process Algebra

In this section we introduce and extremely simple algebra of finite processes. The syntax is defined by the following grammar, where, given countable set M of *messages*, $m \in M$ and $\sigma \in M^*$:

$$P ::= \mathbf{STOP} \mid ?m!\sigma; P \mid P \,[\!]\, P$$

STOP is the process refusing any input and generating no output. Process $?m!\sigma; P$, when receiving input m generates related output σ and then behaves like P; it does not accept any input other then m. Finally $P_1 [\!] P_2$ models the non-deterministic choice between P_1 and P_2, namely the system which may behave like P_1 or P_2.

The operational semantics of PA_{io} is defined by the labeled transition system: $\langle PA_{io}, M \times M^*, - \rightarrow_{pr} \rangle$. Predicate $- \rightarrow_{pr}$ defines the transitions that can take place within the labeled transition system. We define this predicate in a style introduced in [MR92]:

$$\begin{aligned}
&\textbf{def} \; - \rightarrow_{pr} : PA_{io} \times (M \times M^*) \times PA_{io} \rightarrow \mathbf{B} \\
&\textbf{with} \; \mathbf{STOP} - (m, \sigma) \rightarrow_{pr} p \equiv \mathit{false} \\
&\qquad ?m!\sigma; p - (a, \gamma) \rightarrow_{pr} p' \equiv a = m \wedge \sigma = \gamma \wedge p = p' \\
&\qquad p_0 \,[\!]\, p_1 - (m, \sigma) \rightarrow_{pr} p' \equiv \; p_0 - (m, \sigma) \rightarrow_{pr} p' \\
&\qquad\qquad\qquad\qquad\qquad\quad \vee \\
&\qquad\qquad\qquad\qquad\qquad\quad p_1 - (m, \sigma) \rightarrow_{pr} p'
\end{aligned}$$

The behaviour of a process P in PA_{io} is characterized by the node in the above defined labeled transition system corresponding to expression P. We will sometimes use a graphical tree representation of the labeled transition system of an expression, like in Fig.1. The expression represented in the figure stipulates that a system specified by it can only accept message m. On such an input message the system must respond with output σ. After that, the system is prepared to accept only n or k as input, but, depending on the particular "internal status" chosen, it will be prepared either to accept only n or only k. In the first case it will react by outputting γ on input n and not reacting to any other input. In the second case the situation is analogous, but with k and δ instead.

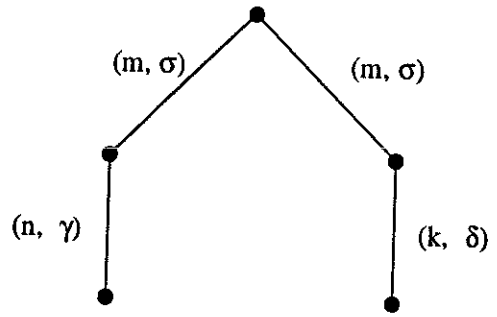

Figure 1: The *LTS* of $?m!\sigma; ?n!\gamma; \mathbf{STOP} \,[\!]\, ?m!\sigma; ?k!\delta; \mathbf{STOP}$

The tree of Fig.2 describes instead a system where, after the input m and relative output σ *both* n and k will be accepted and, according to the particular value submitted, the system will certainly output γ or δ.

This informal reasoning drives us to think that one could use standard testing theory, like for instance developed in [Hen88], by just considering input-output pairs as a particular kind of "actions". In other words one could be attempted to think that the only implication of having made the distinction between input and output explicit would be to structure actions into pairs.

Unfortunately this is not true. The distinction between input and output is much more "semantic", as it can be seen from the following example.

Consider the expression p defined as

$$(?m!\sigma; ?m_1!\sigma_1; \mathbf{STOP}) \,[\!]\, (?m!\sigma; ?m_1!\sigma_2; \mathbf{STOP})$$

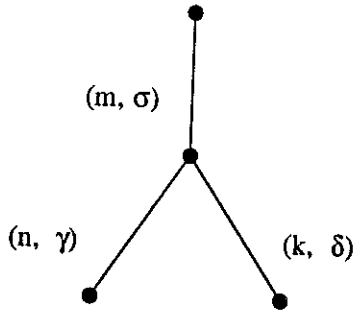

Figure 2: The *LTS* of $?m!\sigma; (?n!\gamma; \mathbf{STOP} \,[\!]\, ?k!\delta; \mathbf{STOP})$

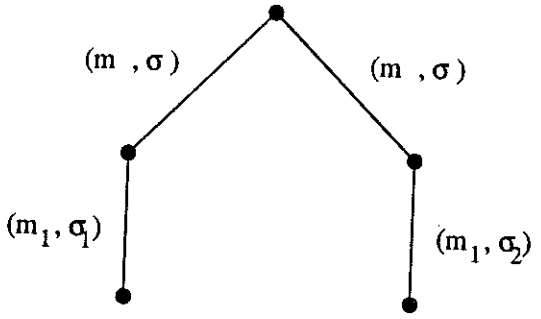

Figure 3: *LTS* of p

This expression can be represented by the *LTS* in Fig. 3.

Now consider the expression q defined as

$$?m!\sigma; (?m_1!\sigma_1; \mathbf{STOP} \,[\!]\, ?m_1!\sigma_2\mathbf{STOP})$$

which has the labeled transition system shown in Fig.4.

Now, if we consider pairs as single actions, processes p and q are *not* testing equivalent. This can easily be seen by showing that the experimenter

$$?m!\sigma; ?m_1!\sigma_1; \mathbf{W}; \mathbf{STOP}$$

distinguishes them. On the other hand, it should be clear that also for process q, after we got output σ on input m, if we provide it with further input m_1, we will either get σ_1 or σ_2 as result, but the choice of the particular *result* should be *up to* the system and then it should not be influenced any longer by the environment. The situation is indeed the same as with p! So, we want an equivalence relation on PA_{io} which is *weaker* than Hennessy Testing Equivalence. In the next section we shall define a testing framework which will induce such an equivalence. Here we want to point out the fact that actually p and q are to be considered equivalent because of the non-determinism present in q due to the fact that there are two *different* transitions labelled with the *same* input m_1 but with *different* outputs. In fact it can be proven [LMP94, Mas95] that the two equivalences coincide only for those processes for which the nodes of the *Acceptance Trees* [Hen88] are labeled by sets of *functions*, whereas those of p and q are labeled by sets containing also *relations*.

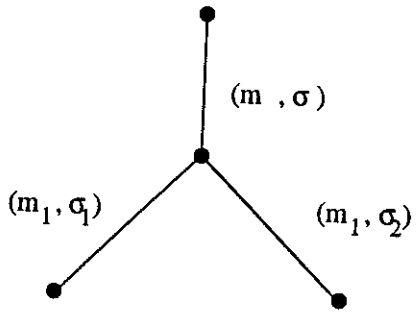

Figure 4: *LTS* of q

Another important fact which is worth noting is that in the traditional testing framework the above experimenter would "force" q to give σ_1 as output. This last point should make it clear also why we do not feel comfortable with the usual process algebraic representation of input and output. In fact, using standard process algebra notation [Hen88], the above situation could be modeled by:

$$p' = \overline{m}\ \sigma\ \overline{m_1}\ \sigma_1 + \overline{m}\ \sigma\ \overline{m_1}\ \sigma_2$$

and

$$q' = \overline{m}\ \sigma\ (\overline{m_1}\ \sigma_1 + \overline{m_1}\ \sigma_2)\ .$$

The two processes p' and q' are indeed Testing Equivalent. On the other hand, choosing to represent the output with actions which semantically are treated in the same way as input ones, as it usually happens in all process algebras we are aware of, brings the problem that the specifier is allowed to model behaviours in which the output of a certain system can actually be controlled by the environment. For instance, the system

$$\overline{m}(\sigma_1 + \sigma_2)$$

would be forced by the experimenter

$$m\overline{\sigma_1}$$

to give σ_1 as output. We consider the above possibility misleading and rather unnatural.

4 A Notion of Testing which Respects Output

In this section we first give a new definition for the notion of *experimenter* and then we study the testing theory it induces.

Definition 4.1 *Output respecting experimenters*
An output respecting experimenter is a process denoted by a term in the set PEO_{io} and the behaviour of which is given by the *LTS* $\langle PEO_{io}, AEO_{io}, - \rightarrow_{exo} \rangle$, where

- $AEO_{io} = M \times M^* \cup \{1, \mathrm{W}\}$

- PEO_{io} is the set of terms generated by the grammar

$$E ::= \mathbf{STOP} \mid ?m!F \mid E \parallel E \mid \mathbf{1}; E \mid \mathbf{W}; E$$

with $m \in M$ and $F \in M^* \rightarrow PEO_{io}$ a *total* function from sequences over M^* to expressions

- $- \rightarrow_{exo} : PEO_{io} \times AEO_{io} \times PEO_{io} \rightarrow \mathbf{B}$ is the transition predicate defined by the following set of equations

$$\mathbf{STOP} - (m, \sigma) \rightarrow_{exo} E \equiv \mathit{false}$$

$$?m!F - (n, \sigma) \rightarrow_{exo} E' \equiv m = n \wedge \sigma \in M^* \wedge E' = F\ \sigma$$

$$E_0 \parallel E_1 - (m, \sigma) \rightarrow_{exo} E' \equiv E_0 - (m, \sigma) \rightarrow_{exo} E' \vee E_1 - (m, \sigma) \rightarrow_{exo} E'$$

$$\mathbf{W}; E - (m, \sigma) \rightarrow_{exo} E' \equiv (m, \sigma) = \mathbf{W} \wedge E' = E$$

$$\mathbf{1}; E - (m, \sigma) \rightarrow_{exo} E' \equiv (m, \sigma) = \mathbf{1} \wedge E' = E$$

□

The key point of the above definition is the fact that an experimenter supplies the process under test with some input and then it must be prepared to receive *any* output the process wishes to return. The subsequent behaviour of the experimenter will depend on such an output. Here we do not care about the particular formalism in which F is specified. We only want to point out that it must be guaranteed, possibly via an additional proof obligation for the specifier of the experimenter, that F is total.

Given the new definition of experimenter we can use it for building up all the machinery of a testing theory [Hen88], where it is intended that two labeled transition systems are *compatible* if they are defined on the same sets of label, with the exception that if one of them is an experimenter, then it can also perform $\mathbf{1}$ and $\mathbf{W}$:

Definition 4.2 *Experimental system respecting output*
Let LTS_P and LTS_E be two compatible labeled transition systems $\langle P, Act, - \rightarrow_{pr} \rangle$ and $\langle E, Act \cup \{\mathbf{1}, \mathbf{W}\}, - \rightarrow_{exo} \rangle$ then $\mathcal{ES}\ (LTS_P, LTS_E)$ is the experimental system $\langle P, E, - \rightarrow_{\parallel}, \mathit{Success} \rangle$ where $- \rightarrow_{\parallel}$ is a predicate defining the interaction relation by the following equation:

$$e \parallel p - (m, \sigma) \rightarrow_{\parallel} e' \parallel p'$$

$\equiv$

$$(e - (m, \sigma) \rightarrow_{exo} e' \wedge p - (m, \sigma) \rightarrow_{pr} p') \vee$$
$$(e - \mathbf{1} \rightarrow_{exo} e' \wedge p = p' \wedge (m, \sigma) = \mathbf{1})$$

and $\mathit{Success} = \{e : E \mid \exists (e' : E\ .\ e - \mathbf{W} \rightarrow_{exo} e')\}$

□

An experimental system models the interaction between an experimenter and the process under test. The test itself is modeled by a sequence of single interactions between the experimenter and the process:

$$e_0 \parallel p_0 - (m_0, \sigma_0) \rightarrow_{\parallel} e_1 \parallel p_1 - (m_1, \sigma_1) \rightarrow_{\parallel} \ldots$$

A *computation* for $e \parallel p$ is a test of maximal length starting from $e \parallel p$ and it is *successful* if and only if there exists i such that $e_i \parallel p_i$ is in the computation and $e_i \in \mathit{Success}$. For process p and experimenter e we say p *may* $_{io}$ e if and only if there exists a successful computation for $e \parallel p$. Analogously we say that p *must* $_{io}$ e if and only if all computations for $e \parallel p$ are successful. Finally, we have $p \underset{\sim}{\sqsubset}_{mayio} p'$ if and only if for all experimenters e, p *may* $_{io}$ e implies p' *may* $_{io}$ e; in the same way, we say $p \underset{\sim}{\sqsubset}_{mustio} p'$ if and only if for all experimenters e, p *must* $_{io}$ e implies p' *must* $_{io}$ e. The testing preorder is defined in the usual way: $p \underset{\sim}{\sqsubset}_{io} p' \equiv p \underset{\sim}{\sqsubset}_{mayio} p' \wedge p \underset{\sim}{\sqsubset}_{mustio} p'$.

I/O-Testing Equivalence, denoted by $\sim_{io}$, is defined as follows:

Definition 4.3 *Testing equivalence*

def $\sim_{io} : P \rightarrow P \rightarrow \mathbf{B}$

with $p \sim_{io} p' = p \underset{\sim}{\sqsubset}_{io} p' \wedge p' \underset{\sim}{\sqsubset}_{io} p$

□

A not so nice feature of our definition of testing equivalence is the fact that we freely used experimenters the behaviour of which is defined in terms of generic total functions. Moreover, given the fundamental impossibility of designing a formalism in which *all* and *only* total computable functions can be defined (see [Rog67] for instance), one possible critic to our proposal is that the very specification of an experimenter is indeed a rather difficult job, since it implies as a "side-obligation" the proof of the totality of all functions involved.

Fortunately, in [Mas95] it is proven that the set of functions of interest is much smaller than that of all total functions from sequences to experimenters. Actually, we need *only* two kinds of functions: constant and conditional ones, which are clearly total. For the experimenters built using these functions we introduce the following short-hands:

$$?m!*; E = ?m!F \textbf{ where } F\ \sigma = E$$

$$?m!\sigma\ [E_1]; E_2 = ?m!F \textbf{ where } F\ \gamma = (\mathit{if}\ \gamma = \sigma\ \mathit{then}\ E_2 \mathit{else}\ E_1)$$

$$[\!] (a : A\ .\ ?a!F) = \ ?a_1!F\ [\!]\ \ldots\ [\!]\ ?a_k!F \textbf{ where } A = \{a_1, \ldots, a_k\}$$

Furthermore, in [Mas95] it is shown that, for characterizing $\sim_{io}$ it is not necessary to consider *all* the experimenters which can be built using the above schemas. In fact, let $\sim_{io}^{E}$ be the testing equivalence which is obtained when only experimenters belonging to a certain set E are considered. Let us now consider the set $\{e\ (s), e\ (s, a), e\ (s, A)\}$ of all the experimenters of the following kinds:

$$
\begin{aligned}
e\ (s) = \ & ?\ m_1!\sigma_1[\mathbf{STOP}]; \\
& (?m_2!\sigma_2[\mathbf{STOP}]; \\
& \cdot \\
& \cdot \\
& ?\ m_n!\sigma_n[\mathbf{STOP}];\mathbf{W})...) \\
e\ (s,(a,\sigma_a)) = \ & \mathbf{1};\mathbf{W} \parallel (?m_1!\sigma_1[\mathbf{1};\mathbf{W};\mathbf{STOP}]; \\
& (\mathbf{1};\mathbf{W} \parallel (?m_2!\sigma_2[\mathbf{1};\mathbf{W};\mathbf{STOP}]; \\
& \cdot \\
& \cdot \\
& (\mathbf{1};\mathbf{W} \parallel (?m_n!\sigma_n[\mathbf{1};\mathbf{W};\mathbf{STOP}]; \\
& (\mathbf{1};\mathbf{W} \parallel (?a!\sigma_a[\mathbf{1};\mathbf{W};\mathbf{STOP}];\mathbf{STOP}))...) \\
e\ (s,A) = \ & \mathbf{1};\mathbf{W} \parallel (?m_1!\sigma_1[\mathbf{1};\mathbf{W};\mathbf{STOP}]; \\
& (\mathbf{1};\mathbf{W} \parallel (?m_2!\sigma_2[\mathbf{1};\mathbf{W};\mathbf{STOP}]; \\
& \cdot \\
& \cdot \\
& (\mathbf{1};\mathbf{W} \parallel (?m_n!\sigma_n[\mathbf{1};\mathbf{W};\mathbf{STOP}]; \\
& \parallel (a:A\ .?\ a!*;\mathbf{W};\mathbf{STOP})...)
\end{aligned}
$$

where s is the sequence $(m_1,\sigma_1) \succ (m_2,\sigma_2) \succ ... \succ (m_n,\sigma_n)$. We have the following

Theorem 4.4
For all p and p' in PA_{io} and $E = \{e\ (s), e\ (s,a), e\ (s,A)\}$ and $s \in (M \times M^*)^*$

$$p \sim_{io}^{E} p' \Rightarrow p \sim_{io} p'$$

Proof
The proof [Mas95] makes use of an auxiliary preorder which is defined directly on processes and which does not make reference to tests. It follows a structure which is similar to the one of the analogous proof in [Hen88].

□

With reference to our first example, we can easily see that the experimenter $e\ ((m,\sigma),\{n\})$ distinguishes the process of Fig.1 from that of Fig.2.

We end this section by mentioning the existence of a *denotational* model for PA_{io}. The model is that of *functional finite acceptance trees* (*ffAT*). Finite acceptance trees have been introduced by Hennessy as a model for finite non-deterministic processes. We refine it for PA_{io} replacing every node-label $\{A_1, ..., A_n\}$ by the set $\{F_{11}, ..., F_{1j}, ..., F_{n1}, ..., F_{nk}\}$ where we replace *relation* $A_i \subseteq M \times M^*$ with all its maximal subsets $F_{i1}, ..., F_{ih}$ which are *functions* from M to M^*.

For example the functional finite acceptance tree denoted by process p of Fig.3 is shown in Fig.5 where the relation $\{(m_1,\sigma_1),(m_1,\sigma_2)\}$ has been replaced by two functions, namely the two singletons $\iota\ (m_1,\sigma_1)$ and $\iota\ (m_1,\sigma_2)$.

On *ffAT* operators like (input/output-)prefixing and choice ($+_{ffAT}$) are defined in a way which takes the distinction between input and output into consideration. Also a partial order

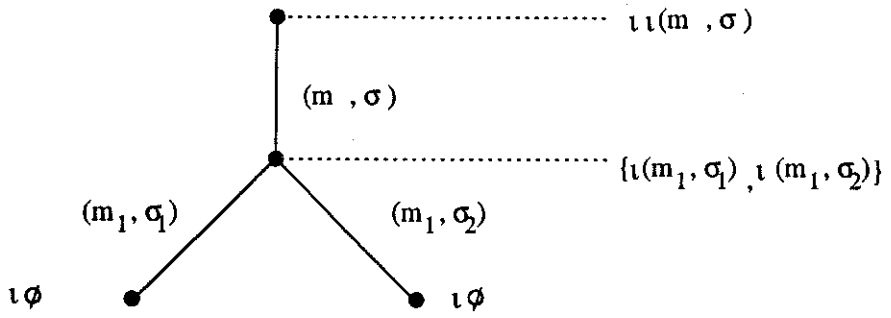

Figure 5: *ffAT*-representation of p

$\leq_{ffAT}$ is defined and, letting $PffAT[\,b\,]$ denote the functional finite acceptance tree of process b, the following pleasant result holds:

Theorem 4.5 *Full Abstraction*
For every b and b' in PA_{io}

$$b \underset{\sim}{\sqsubset}_{io} b' \equiv PffAT[\,b\,] \leq_{ffAT} PffAT[\,b'\,]$$

Proof
Also this proof uses the auxiliary preorder on processes. Moreover, the proof of several peculiar facts about the relation between the union/convex-closure operator [Hen88] and the transformation from relations to functions in the acceptance sets is needed.

□

With reference to our sample processes p and q we find that $PffAT[\,p\,] = PffAT[\,q\,]$, so $p \sim_{io} q$ as we wanted.

Another nice side-effect of our construction is that any functional finite acceptance tree can easily be decomposed into a set of deterministic trees with their edges labeled by input-output pairs. Any of such trees uniquely represents a continuous function from M^ω to M^ω. Fig.6 shows the set of deterministic trees obtained by decomposing $PffAT[\,p\,]$. It should be clear that the two trees represent the two functions p_1, p_2 where, for $i = 1, 2$, p_i gives $\sigma +\!\!+ \sigma_i$ on all sequences $(m \succ (m_1 \succ \gamma))$ for any γ, it gives σ on $(m \succ \varepsilon)$ and on sequences $(m \succ (k \succ \gamma))$ for $k \neq m_1$ and any γ, and it gives ε for all other sequences.

The relation between *ffAT* and continuous functions will be further exploited in the next section.

5 A Functional Algebra

In this section we shall take the *Data-flow* approach to the description of systems [Kah74, Bro92a, Bro92b]. A *possible behaviour* of a system is described by a *continuous* function from M^ω to M^ω where *sequence prefixing* is the partial ordering, which indeed makes M^ω a c.p.o. with the empty sequence ε as bottom. The motivation for continuity is that for instance it rules out functions which would need infinite input before producing any output

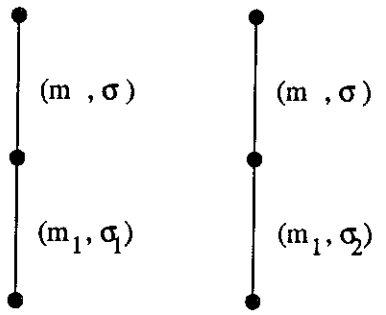

Figure 6: Decomposition of *PffAT* [*p*]

or that when provided with further output could modify the output previously given. Functions like these cannot be computed by machines so, for modeling the latter, we restrict to continuous ones. We also require $f\ \varepsilon = \varepsilon$; this requirement is not essential, but it simplifies our discussion. Moreover we consider only those functions f such that for all σ the set $\{m \mid \exists(\gamma\ .\ f\ (\sigma +\!\!+ (m \succ \gamma)) \neq (f\ \sigma))\}$ is finite. These are exactly the functions which can be represented by *finitely branching* deterministic trees where each edge is labeled by a message m and the increment β in the output corresponding to increment m in the input (β is the value of the so called *derivative* of the function on $\sigma \succ m$ where σ uniquely identify in the tree the node from which the arch whose label has m in the input component comes out). We call *CTSF* the set of the functions meeting the above requirements. Finally, in this paper we consider only those functions which "react" only to a finite set of sequences, i.e. functions the tree representation of which is indeed *finite*. In order to deal with non-determinism we let specifications be just *sets* of functions:$SPEC \equiv CTSF \rightarrow \mathbf{B}$.

We define now a set of *combinators* on functions and then we lift them on specifications so that they will allow to build up specifications in a systematic way giving raise to what we will call the *Functional Algebra* FA.

Definition 5.1 *Terminator*

def $\mathcal{E} : CTSF$
with $\forall(\sigma\ .\ \mathcal{E}\ \sigma = \varepsilon)$

□

Obviously, $\mathcal{E}$ models the behaviour of not reacting to any stimulus.

Definition 5.2 *Input/output prefix on functions*

def $(—, —); — : M \rightarrow M^* \rightarrow CTSF \rightarrow CTSF$
with $((m, \sigma); f)\ \varepsilon = \varepsilon$
$((m, \sigma); f)\ (k \succ \sigma') =$ *if* $k \neq m$ *then* ε *else* $\sigma\ +\!\!+ (f\ \sigma')$ *fi*

□

This is the analogous on functions of *action-prefix* in process algebras. Here an "action" consists in receiving a message m producing output σ. Thus, the first step of a (deterministic) system described by $(m, \sigma); f$ is receiving m as input and producing σ as output. After that, the system behaves like f. Notice that such a system does *not* react on sequences which do not start by m, thus giving ε on them.

The choice-combinator on *CTSF*-functions is defined as follows:

Definition 5.3 *Choice on CTSF-functions ($[\!]$)*

def $- [\!] - : CTSF \to CTSF \to SPEC$
with $(f_1 [\!] f_2)\ f = tree\ f \in dec\ (tree\ f_1 +_{ffAT}\ tree\ f_2)$

□

In the above definition functions *tree* and *dec* are respectively the tree representation of a function in *CTSF* and the decomposition of an element of $ffAT$ into a set of deterministic trees which we mentioned in the previous section. Notice that in general $(f_1 [\!] f_2)$ characterizes *a set* of functions, obviously due to the possible presence of *internal* non-determinism. A couple of simple lemmas easily follow from the definitions:

Lemma 5.4 *Consistency of choice*

$$\forall (f_1, f_2 : CTSF\ .\ \exists (f : CTSF\ .\ (f_1 [\!] f_2)\ f))$$

□

Lemma 5.5 *CTSF closed under i/o-prefix and choice*

$$\forall (f : CTSF, m : M, \sigma : M^*\ .\ (m, \sigma); f \in CTSF) \wedge$$

$$\forall (f_1, f_2, g : CTSF\ .\ (f_1 [\!] f_2)\ g \Rightarrow g \in CTSF)$$

□

The above combinators are lifted to specifications in the obvious way:

Definition 5.6 *FA*

def STOP : $SPEC$
with STOP$f = (f = \mathcal{E})$

def $?\ —!—; — : M \to M^* \to SPEC \to SPEC$
with $(?m!\sigma\ ; S)\ f = \exists (g\ .\ S\ g \wedge f = (m, \sigma); g)$

def $— [\!] — : SPEC \to SPEC \to SPEC$
with $(S_1 [\!] S_2)\ f = \exists (f_1, f_2 : CTSF\ .\ S_1 f_1 \wedge S_2 f_2 \wedge (f_1 [\!] f_2)\ f)$

□

Lemma 5.7 *Consistency of Specification Combinators*
All specifications built using *only STOP*, i/o-prefix on specifications and choice on specifications are consistent.

Proof
Trivial.

□

FA can as well be taken as a denotational model for PA_{io} and the following theorem holds:

Theorem 5.8 *Full Abstraction*
For all p_1 and p_2 in PA_{io}

$$p_1 \sim_{io} p_2 \equiv fun\ [\ p_1] = fun\ [\ p_2]$$

Proof
The proof uses full abstraction of $ffAT$ (Theorem 4.5), some properties of functions *tree*, *dec* and $+_{ffAT}$ and a non-trivial lemma on the the relation between *dec* and $+_{ffAT}$ which is stated below.

□

Lemma 5.9 *Pairwise*
For all a_1, a_2 in $ffAT$

$$dec\ (a_1 +_{ffAT} a_2) =$$

$$\{d \mid \exists (d_1 : dec\ a_1, d_2 : dec\ a_2\ .\ d \in dec\ (d_1 +_{ffAT} d_2))\}$$

Proof
The proof is based on properties of union/convex-closure and of the transformation of relations to sets of functions which are not immediate to prove.

□

6 Conclusions and Future Work

This paper constitutes an attempt to combine the process algebra approach to concurrency with the Functional/Data-flow apprach. The key point is the conceptual semantic distinction between input and output actions. A process algebra PA_{io} together with its operational semantics and related testing equivalence $\sim_{io}$ is given for which the above requirement of dealing with input and out in different ways is met. Two denotational models, one based on *Functional Finite Acceptance Trees* and the other one based on (sets of) *prefix-continuous Sequence Processing Functions* are given and both are shown fully abstract w.r.t $\sim_{io}$. One could argue that the last model is nothing more than a different representation of $ffAT$, which is partially true although proving that the two models coincide is not easy as it requires a rather non-trivial lemma on the convex/union-closure. The importance of the functional model lays on the fact that it constitutes an effective and explicit bridge between the process algebra approach and the Data-flow one. In fact, being fully abstract w.r.t. $\sim_{io}$ it incorporates notions like non-determinism and progress properties of systems and, being completely functional,

it opens the door towards the full exploitation of functional/Data-flow operators and proving techniques based on them. It is worth noting here that typical Data-flow operators like *functional* and *parallel* composition and *feedback* allow the description of any network of communicating agents [Bro92a] which, with the aid of the combinators we proposed in this paper, can be defined in a systematic way and with the inclusion of non-determinism. So, our approach could represent an interesting alternative to the use of synchronization/communication operators typical of process algebra. In this respect it is worth mentioning that we are currently using an extension of the model presented here including recursion with both minimal and maximal fixpoint in conjunction with Data-flow operators for the specification and validation of the Alternating Bit Protocol and our experiments look quite promising in terms of clarity and conciseness of both the specification and its correctness proof, which is written in a completely transformational style.

The above mentioned extension with recursion and with typical Data-flow operators is the first step towards the improvement of the model presented here.

Finally, it is maybe superfluous to point out that in this paper we presented a model more than a really usable specification language. It is then necessary to introduce notational as well as semantical extensions in order to make FA suitable for specifying and reasoning about realistic systems. Moreover, suitable equational laws for the operators of FA and its extensions are due.

Of course, in order for the work presented here to make really sense a suitable kit of automatic tools for the management of specifications and, most of all, for a (semi-)automatic support to theorem-proving is necessary.

7 Acknowledgements

The authors would like to express their gratitude to Dino Pedreschi for his helpful comments and for the fruitful discussions they had on the subject od this paper.

References

[Abr84] S. Abramsky. Reasoning about concurrent systems. In P. Jones Chambers, Duce, editor, *Distributed Computing.* Academic Press, 1984.

[Abr89] S. Abramsky. A generalized kahn principle for abstract asynchronous networks. In *Mathematical Foundations of Programming Language Semantics*, 1989.

[Bou92] R. Boute. A declarative formalism supporting hardware/software co-design. In *IFIP International Workshop on Hardware / Software Co-design*, 1992.

[Bou94] R. Boute. An introduction to funmath. Introductory notes for the course 'Basic Mathematical Complements for Computer Science' - University of Gent, Belgium, 1994.

[Bro90] M. Broy. Functional specification of time sensitive communicating systems. In J. W. de Bakker, W. P. de Roever, and Grzegorz Rozenberg, editors, *Stepwise Refinement of Distributed Systems — Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990.

[Bro92a] M. Broy. Compositional refinement of interactive systems. In *Program Design Calculi - International Summer School* [NAT92].

[Bro92b] M. Broy. (inter-)action refinement: The easy way. In *Program Design Calculi - International Summer School* [NAT92].

[BW88] R. Bird and Ph. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[DC89] E.W. Dijkstra and Scholten C.S. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1989.

[DS89] P. Dybjer and H.P Sander. A functional programming approach to the specification and verifivation of concurrent systems. *Formal Aspects of Computing*, 1:303–319, 1989.

[Dyb86] P. Dybjer. Program verification in a logic theory of construction. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 334–349. Springer-Verlag, 1986.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* The MIT Press, 1988.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prantice Hall, 1985.

[Jon87] B. Jonsson. A fully abstract trace model for dataflow networks. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE — Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the 1974 IFIP Congress.* IFIP, 1974.

[LMP94] D. Latella, M. Massink, and D. Pedreschi. A functional approach to testing equivalence. Internal Report C94-25, Consiglio Nazionale delle Ricerche, Istituto CNUCE, December 1994.

[Mas95] M. Massink. *A Functional Approach to Concurrency.* PhD thesis, Univerity of Nijmegen, 1995. to appear.

[Mil89] R. Milner. *Communication and Concurrency.* Series in Computer Science. Prantice Hall, 1989.

[Mis90] J. Misra. Equational reasoning about nondeterministic processes. *Formal Aspects of Computing*, 2:167–195, 1990.

[MNV73] Z. Manna, S. Ness, and J. Vuillemin. Inductive methods for proving properties of programs. *Communications of the ACM*, 16(8), 1973.

[MR92] M. Massink and L. Rooijakkers. Equational semantics for basic LOTOS and an example of its use in a transformational proof style. In P. Dewilde and Vandewalle J., editors, *Computer Systems and Software Egineering - the 6th Annual European Computer Conference.* IEEE, IEEE Computer Society Press, 1992.

[NAT92] Program design calculi - international summer school, 1992.

[Ong93] C.H.L. Ong. Non-determinism in a functional setting. In *IEEE Symposium on Logic in Computer Science.* IEEE, Computer Society Press, 1993.

[Rei85] W. Reisig. *Petri-Nets - an introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1985.

[Rog67] H. Rogers. *Theory of Recursive Functions and Effective Computability.* Higher Mathematics Series. Mc Graw Hill, 1967.

[San92] H. P. Sander. *A Logic of Functional Programs with an Application to Concurrency.* PhD thesis, Univ. of Göteborg and Chalmers Univ. of Technology, 1992.

[vG90] R.J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions.* PhD thesis, University of Amsterdam, 1990.