Formal Aspects of Computing (1993) 3: 1–000

Communicating Processes with Value-passing and Assignments[1]

M. Hennessy

A. Ingólfsdóttir

University of Sussex
Falmer, Brighton BN1 9QH
ENGLAND

Keywords: communicating processes; denotational models; Occam; testing; value-passing

Abstract. A semantic theory of an imperative language which allows value-passing and assignments as a simple action prefixing is described. Three different semantic approaches are given: denotational based on the mathematical model Acceptance Trees, axiomatic based on inequations and behavioural in terms of testing. The equivalence of these different approaches is shown. The results are compared with similar results for other languages such as *CSP* and *Occam*.

1. Introduction

In this extended introduction we give a detailed description of the contents of the paper, compare it with previous work and outline what we perceive to be the advantages of our approach.

1.1. Languages for Concurrent Systems

In 1978 Hoare published a proposal for a *parallel programming language* called *CSP*, [Hoa 78]. In this language a program consists of a collection of independent processes, each with its private memory or store, which could communicate

[1] This work has partly been supported by the ESPRIT/BRA project Concur
Correspondence and offprint requests to: M. Hennessy and A. Ingólfsdóttir

with each other using a form of instantaneous exchange of messages called *handshake communication*. Each individual process in a program may be viewed as a straightforward imperative program working on its own memory, with assignment statements, boolean tests, iteration, etc.

Subsequently, although some work was done on the semantics of this language, [FLP 84], most of the research effort was devoted to semantic theories of more abstract versions, such as *theoretical CSP, TCSP*, [BHR 84], [Plo 82] and *CCS*, [Mil 89]. These languages, often refered to as *process algebras*, differ significantly from the original *CSP*. For example *TCSP* may be viewed as an applicative language. There is no store nor assignment statement and no values may be transmitted between processes; they only communicate by synchronising on signals. Nevertheless this area of research has been very fruitful. Semantic models have been developed and are well understood and well-behaved syntactic constructs have been isolated.

The programming language *Occam* may be seen as a direct descendant of the original programming language *CSP*. It contains most of its basic features. For example, it is imperative, with each process having its own private memory. But communication is now via *channels* although the basic communication principle is the same. However, there are many restrictions on the syntactic form of constructs in the language. Some of these were imposed in order to obtain efficient implementations, others because of conceptual difficulties. In this paper we would like to suggest that, at least from a theoretical point of view, these restrictions are unnecessary. We show that it is possible to define clean extensions to *Occam*, i.e. imperative parallel programming languages, based on private memories for processes and handshake communication, which have fully-abstract denotational models and associated complete proof systems. The key to our approach is to take advantage of the existing work on the more abstract process algebras, in particular by importing into the world of imperative programs the syntactic constructs which have proven to be mathematically well-behaved in the applicative world of process algebras. For example, we will have an *external choice* mechanism similar to the existing ALT construct of *Occam* but in addition we will have an *internal choice* mechanism, which has proven to be of great use in more theoretical developments. This phenomenon already appears in *Occam* : in

$$\begin{array}{l} ALT \\ \quad SKIP \rightarrow P \\ \quad SKIP \rightarrow Q \end{array}$$

an internal nondeterministic choice is made between P and Q. So we only emphasise its importance by making it a basic element of the syntax. We will also abolish all restrictions on the use of channels in the parallel construct, PAR, and will also allow arbitrary processes to be used in the ALT construct. In one significant aspect we will be less general than *Occam*. We will not use the sequential composition construct SEQ. Instead we will use the more primitive notion of *action prefixing* from *CCS*. Here the actions will consist of the usual input and output actions, $c?x$ and $c!e$, together with the new action assignment, $x := e$. This does not impinge significantly on the usefulness of the language but, as we will see, it enables us to simplify considerably the semantic model required to interpret it.

The language is described in detail in Section 2.1 where it is compared with *Occam* and, in particular, the version of *Occam* used in [HR 88] and [Ros 87].

1.2. Denotational Models

A standard model for *TCSP* is the *failure-sets model* described, for example in [BHR 84]. It consists of suitable collections of pairs of the form (s, X) where s is a trace of actions which the process can perform and X is a finite set of actions which it can subsequently refuse. A further component of the model also contains information about possible internal divergences.

This model has been adapted in [Ros 87] in order to interpret a subset of *Occam*. The adaptation is two-fold: the first to handle the communication of values and the second to handle stores. Values are accommodated by using actions of the form $c.v$ where c is a channel name and v is a value. Intuitively this action represents the passage of the value v along the channel c. Stores are incorporated by extending that part of the model which records possible internal divergences. Whereas previously it consisted simply of the traces of a process which could lead to divergence, it also now includes those traces which lead to successful termination and the resulting store.

There are two major problems associated with this extended model, both connected with the treatment of value-passing. The first is that the allowed set of values must be finite for otherwise the semantic operators would no longer be continuous. Although in practice a given program will only ever use a finite set of values, it is conceptually very restricting to limit the possible values usable in a programming language to be finite. The second arises from the form of the actions, $c.v$. Processes *input* values from channels or *output* values to channels and these are different actions. In [Ros 87] they are modelled by the same action, $c.v$ which merely records the passage of v along c. This is possible because the model is only used to interpret a very restricted form of parallelism, where every channel has exactly two users, one using it only for input and the other output. This approach would be inadequate for more general languages which do not impose such restrictions. In this paper we use a different model which may be used for languages where the set of communicated values may be infinite and where there are no restrictions on the use of communication channels.

In [He 88] a model, called Acceptance Trees, very similar to *failure-sets*, is described and used to model abstract languages such as *TCSP* and *CCS*. In order to interpret value-passing in [HI 89] it is extended to a model called AT^v, which is obtained as an instantiation of the more general mathematical model called Natural Interpretation, introduced in [HP 80]. It does not suffer from the disadvantages outlined above. In particular it allows values to range over arbitrary countable sets and it can be used to interpret general forms of the parallel construct because actions of the form $c?v$ and $c!v$ are used, representing input and output respectively. Nevertheless it is an algebraic cpo obtained as the initial solution to a domain equation and supports the usual equational reasoning associated with the more abstract languages such as *TCSP* and *CCS*. In order to be able to use it to model imperative parallel programming languages, it is therefore sufficient to adapt it in order to model the use of stores. However, we claim that for our particular variation of *Occam* this is unnecessary. The reason for this is two-fold.

First we take seriously the idea that each individual process has its own

private store and the only access to this store is by communication with the process. From this point of view the two processes

$$(x := 1).c?x.P$$

and

$$(x := 2).c?x.P$$

are behaviourally identical; although they have different effects on their private memory, this difference is not discernible to any external observer or, indeed, any larger system which uses them as subprocesses. Of course it would be quite different if, after updating its private memory, there is a subsequent possibility of communicating the effect of this update. So, for example, the processes

$$(x := 1).c!x.STOP$$

and

$$(x := 2).c!x.STOP$$

will be distinguished in the model. But in general there is no need to record the sequence of store transformations carried out by a process as it receives and sends communications. This is also true of the model in [Ros 87] but there the terminal states of processes are recorded. This is unnecessary for us because we avoid the use of the sequential composition operator SEQ and in its place make more use of action-prefixing. Thus the processes

$$(x := 1).STOP$$

and

$$(x := 2).STOP$$

are behaviourally equivalent and indeed they are equivalent to the trivial process $STOP$. These would not be reasonable identifications if the language contained SEQ for a difference between them could be discovered by a larger system: the programs

$$\begin{array}{ll} SEQ & \text{SEQ} \\ \quad (x := 1).STOP & \quad (x := 2).STOP \\ \quad c!x & \quad c!x \end{array}$$

are certainly behaviourally different. By avoiding the use of SEQ we lose very little descriptive power but we are able to use the model AT^v unaltered.

We describe the general model Natural Interpretations for the language in §2.2 and show how it is used to interpret our version of *Occam*. In §2.3 the model AT^v is described briefly although the reader is referred to [HI 89] for a complete description.

In §2.4 we define the syntactically and semantically finite approximations of a term and show that the interpretation of a general term is decided by the interpretations of its finite approximations.

1.3. Proof Systems

An immediate advantage of using the existing model AT^v is that we may continue to use all of the well-known equational laws which it supports. Indeed, the

syntactic constructs we borrow from process algebras were specifically designed so as to enjoy simple equational laws. This is in contrast to [HR 88] where a new collection of laws is required, some of them being far from intuitive. Of course we will require additional laws from the new constructs of the language, such as the assignment statement but these are all very simple and intuitive.

In §2.5 the resulting proof system is discussed and shown to be sound and complete for *closed* processes.

1.4. Full-Abstraction

Rather than simply present a reasonable denotational model for our language, we also wish to justify it behaviourally. We do so by proving it to be *fully-abstract* with respect to some suitable notion of behaviour, i.e. we show that the model identifies and only identifies those processes which we deem to be behaviourally equivalent. The notion of behaviour we use is a standard one, based on testing, [He 88]. This is a two-level view of processes. The first gives an operational semantics of the language in terms of *labelled transition systems*, which describes the individual computation steps or actions a process may perform. The second describes how a test is applied to a process. Here we take a test to be another process whose memory or store is distinct from that of the process under observation. So the only knowledge of the store of the observed process available is what may be discovered via communication over channels.

The major difficulty of the paper is to invent a suitable operational semantics for the language. In fact this is the main contribution of the paper. We design a reasonable operational semantics and behavioural equivalence for the language which describes our intuition of the behaviour of processes and agrees with an existing denotational interpretation [Miln 88]. There are three natural types of computation steps a process may perform: input, output and internal move. In the world of process algebras the validity of the denotational model, and many of the equational laws, depends on the fact that internal moves do not affect the syntactic structure of processes. Thus in the operational semantics of [HI 89], [Ol 85], we have

$$\begin{array}{l} ALT \\ \quad P \\ \quad Q \end{array} \quad \rightarrowtail \quad \begin{array}{l} ALT \\ \quad P' \\ \quad Q \end{array} \quad \text{whenever} \quad P \rightarrowtail P'$$

where $\rightarrowtail$ represents an internal move. However this cannot be applied directly to our language as we would obtain non-intuitive computations, and a notion of behaviour different from that encapsulated in the denotational model. For example, let s be a store where x contains 0. Then using the above property of internal moves we would have

$$\left(\begin{array}{l} ALT \\ \quad (x := 1).STOP \\ \quad c!x.STOP \end{array} \quad , s \right) \rightarrowtail \left(\begin{array}{l} ALT \\ \quad STOP \\ \quad c!x.STOP \end{array} \quad , s[1/x] \right)$$

and this latter process could then output 1 on the channel c. However, this is unexpected behaviour of the original process

$$\begin{array}{l} ALT \\ \quad (x := 1).STOP \\ \quad c!x.STOP \end{array}$$

which is supposed to be a choice between two behaviours, one which outputs the value of x, which is 0, and the other which updates the store. Our solution to this problem is given in Section 3.1 where we present the operational semantics. This is followed in Section 3.2 by a definition of testing and the associated preorder. Finally, in Section 3.3, we prove that our model is fully-abstract.

2. The Language and its Model

In this section we introduce our language and describe its denotational semantics. The section is divided into five subsections: In the first one we give the syntax of the language and some example programs. In §2.2 we describe a general mathematical model for the language in terms of natural interpretations and then in §2.3 we give a brief introduction to one such model, the so-called *Strong Acceptance Trees* [HI 89], which is a modification of the model Strong Acceptance Trees introduced in [He 85] and [He 88]. In §2.4 we define syntactically and semantically finite approximations of programs and show that the denotational interpretation is completely decided by these. In the last subsection we define a proof system and prove its soundness and completeness with respect to the model.

2.1. Syntax

Our language, $VPLA$, (a Value-Passing Language with Assignment) is an extension of the applicative concurrent language, VPL, introduced in [HI 89]. We have added the imperative construct assignment but only as a form of prefixing. As we want to compare our language with the existing concurrent programming language *Occam* we omit renaming.

The language is therefore a slight modification of *Occam* although we use a different more abstract syntax. The main differences are that beside the usual external choice or alternation operator ALT, which in our setting is called $+$, we have an internal choice operator $\oplus$. Further we have the restriction in our language that sequential composition is not allowed in general but only as a prefixing of the input/output actions or of an assignment statement. The operator $\backslash c$ plays the role of a local declarations of channels in *Occam* and Ω can be used to model the *error* handling. Further the $RecP.\_\_$ can be used to simulate the construct $While\ e\ p$ from *Occam* and the various datatypes can easily be incorporated into our language.

In the definition of the language we assume a predefined syntactic category of expressions, Exp, ranged over by e. This includes a set of variable symbols, Var, ranged over by x, and a countable set of value symbols, Val, ranged over by v. The assumption that the set Val is countable is very important in our semantical description of the language. This implies that Val has the form $\{v_1, v_2, \cdots\}$ and if we define $V_n = \{v_1, \cdots, v_n\}$ then $Val = \bigcup_n V_n$ where $V_n \subseteq Val$ obviously are finite. We use this notation of V_n throughout the paper.

To obtain a language closer to *Occam* we could, for example, assume that Exp includes the usual datatypes allowed in *Occam*. We will also assume a syntactic category of boolean expressions, $BExp$, ranged over by be, including the set of boolean variables, $BVar$, ranged over by bx, and with the only constants T and F. Further we assume the set of free variables of a data expression, $FVar(e)$,

$$t \quad ::= \quad op(t_1,\ldots,t_k), op \in \Sigma_k \mid P \mid pre.t \mid recP.t \mid be \rightarrow t,t$$

$$pre \quad ::= \quad c!e \mid c?x \mid x := e$$

Fig. 1. Syntax

and a boolean expression, $FVar(be)$, to be predefined. We let $Chan$ denote a predefined set of *channel names*, ranged over by c.

The allowed operators in the syntax are $STOP$, and Ω of arity 0, $\backslash c$ of arity 1 and $\oplus$, $+$ and $|$ of arity 2. We use Σ to denote this collection of operators and Σ_k those of arity k. We also need a predefined set of process names, PN, ranged over by upper-case letters such as P, Q, etc. The set of terms is then defined by the BNF-definition given in Figure 1.

We finish this section by giving some programming examples in $VPLA$:

Example 2.1.1. Consider the process

$$in?x.RecQ.c?y.x = 0 \rightarrow \; out!y.STOP, \; (x := x-1).Q.$$

It has three channels, in, out and c. From the channel in it reads a number, n, and then outputs on out the n-th value input from c. (A simplification of Example 2.1.1 in [HI 89], where communication is used to simulate assignment.)

Example 2.1.2. The process

$$x := 1.recQ.(x := x+1).in?y.x \, div \, 2 = 0 \rightarrow \; left!y.Q, \; right!y.Q.$$

uses three channels, in, $left$ and $right$. It inputs numbers from in and outputs them alternatively on the channels left and right.

2.2. A General Denotational Model

In this subsection we will give the general structure of denotational models for the language $VPLA$.

A *Natural Interpretation* (for the language $VPLA$), consists of a triple

$$\langle D, out_D, in_D \rangle$$

where

i) D is a $\Sigma - cpo$

ii) $out_D : (Chan \times Val \times D) \longrightarrow D$ is a total function continuous in its third argument.

iii) $in_D : (Chan \times (Val \longrightarrow D)) \longrightarrow D$ is a total function continuous in its second argument, where $Val \longrightarrow D$ inherits the natural pointwise ordering from D.

Given such a natural interpretation, D, we can define a semantic interpretation of *VPLA* following the usual approach of denotational semantics. We let Env_D be the set of D-environments, i.e. mappings from PN to D, ranged over by ρ and St the set of stores, mappings from Var into Val, ranged over by σ. We assume evaluation functions $[\![\ \ ]\!] : Exp \longrightarrow (St \longrightarrow Val)$ and $[\![\ \ ]\!] : BExp \longrightarrow (St \longrightarrow \{T, F\})$. Then the semantics of the language *VPLA* is given as a function:

$$D[\![\ ]\!] : VPLA \longrightarrow (Env_D \longrightarrow (St \longrightarrow D))$$

and is defined by structural induction on *VPLA*:

i) $D[\![P]\!]\rho\sigma = \rho(P)$

ii) $D[\![op(\underline{t})]\!]\rho\sigma = op_D(D[\![\underline{t}]\!]\rho\sigma)$

iii) $D[\![recP.t]\!]\rho\sigma = Y\lambda d.D[\![t]\!]\rho[d/P]\sigma$

iv) $D[\![be \rightarrow t, u]\!]\rho\sigma = \begin{array}{ll} D[\![t]\!]\rho\sigma & \text{if } [\![be]\!]\sigma = T \\ D[\![u]\!]\rho\sigma & \text{if } [\![be]\!]\sigma = F \end{array}$

v) $D[\![c!e.t]\!]\rho\sigma = out_D(c, [\![e]\!]\sigma, D[\![t]\!]\rho\sigma)$

vi) $D[\![c?x.t]\!]\rho\sigma = in_D(c, \lambda v.D[\![t]\!]\rho\sigma[v/x])$

vii) $D[\![x := e.t]\!]\rho\sigma = D[\![t]\!]\rho\sigma[[\![e]\!]\sigma/x]$

where Y is the least-fixpoint operator for continuous functions over D.

2.3. Acceptance Trees

In this subsection we will give a brief description of the mathematical model called *Strong Acceptance Trees*, or just *Acceptance Trees*, introduced in [He 85] and explained in more detail in [He 88]. Then we will explain the modified version, which models the value-passing calculus VPL in [HI 89]. The definition of the pure version assumes a set of "pure" actions, Act which processes can perform. Further we need the notion of saturated sets. Thus a finite set, $\mathcal{A} \subseteq \mathcal{P}_{fin}(Act)$, where $\mathcal{P}_{fin}(Act)$ is the family of finite subsets of Act, is said to be saturated if it satisfies the following conditions:

1. $\bigcup \mathcal{A} \in \mathcal{A}$
2. $A, B \in \mathcal{A}$ and $A \subseteq C \subseteq B$ implies $C \in \mathcal{A}$.

The set of all saturated subsets of $\mathcal{P}_{fin}(Act)$, all saturated sets over Act, is denoted by $sat(Act)$. A saturated set over Act is called an acceptance set. The saturated closure, $c(\mathcal{A})$, of a finite set, $\mathcal{A} \subseteq \mathcal{P}_{fin}(Act)$ is defined as the least set which satisfies

1. $\mathcal{A} \subseteq c(\mathcal{A})$
2. $\bigcup \mathcal{A} \in c(\mathcal{A})$

3. $A, B \in c(\mathcal{A})$ and $A \subseteq C \subseteq B$ implies $C \in c(\mathcal{A})$.

It follows easily from the definition, that $c(\mathcal{A})$ is the least saturated set, which includes $\mathcal{A}$.

Finite acceptance trees are rooted finite branching trees of finite depth. Each node is either open or closed. Each closed node is labelled by an acceptance set, $\mathcal{A}$, where each set, A, in $\mathcal{A}$ models a state the process can reach internally or without performing any visible action. The actions in A are those which the process can perform when in that state. The arcs leading from a node are labelled by the actions which occur in the acceptance set labelling the node. Thus for every $a \in \bigcup \mathcal{A}$ there is exactly one outgoing branch labelled by a. In that sense the tree is deterministic but the nondeterminism inherent in processes is modelled by the acceptance sets. A branch labelled by an action, a, leads to a subtree, which models the behaviour of the process after having performed the action a.

Only the leaves of the tree can be open and in that case they are unlabelled. The open nodes model divergent or unspecified processes or states.

If we denote the open nodes by $\perp$, we can define the set of finite acceptance trees, fAT, formally as the least set which satisfies

1. $\perp \in fAT$
2. $\mathcal{A} \in sat(Act)$ and $f : \bigcup \mathcal{A} \longrightarrow fAT$ implies $(\mathcal{A}, f) \in fAT$.

We let T, U, etc. range over fAT and use the notation $[a_1 \longrightarrow T_1, \ldots, a_n \longrightarrow T_n]$ for the function which maps the element a_i to T_i, $i = 1, 2, \ldots, n$. Note that the leaves are either labelled by $\perp$ or by $(\{\emptyset\}, [\,])$, where the latter models the empty process. The trees are ordered as follows:

1. $\perp \leq T$ for all $T \in fAT$.
2. $(\mathcal{A}, f) \leq (\mathcal{B}, g)$ if

   i) $\mathcal{B} \subseteq \mathcal{A}$
   ii)$f \leq g$
   where $f \leq g$ means
   a) $domain(g) \subseteq domain(f)$
   b) for all $a \in domain(g)$ $f(a) \leq g(a)$.

It can be seen very easily, that $\leq$ is a partial order. It represents the intuition that one process is an "improvement" of another if it is more deterministic, i.e. has fewer internal states.

To model infinite or recursively defined processes we need a notion of infinite or recursively defined acceptance trees. Thus we aim for an extension of $(fAT, \leq)$ which is a cpo. This can be obtained by looking at the construction of trees from already existing trees as a functor in the category of cpos, $\underline{\underline{CPO}}$, and defining the set of acceptance trees as the least fixpoint to the functor. Let us assume we have a cpo, D, and a general set of events or actions, Ev. Then we define the tree constructor, H, by

$$H(Ev, D) = \{(\mathcal{A}, f) | \mathcal{A} \in sat(Ev), f : \bigcup \mathcal{A} \longrightarrow D\}$$

By adding the bottom element and using the order $\leq$ defined above we get a cpo $(H(Ev, D))_\perp$. Following the procedure in [Plo 81] it is straightforward to turn H into a continuous functor, which we again call H. We can now define AT to be the least solution to the domain equation

1. $D = (H(Ev, D))_\perp$
2. $Ev = Act$.

It is simple to show that H preserves ω-algebraicness and thus from the general theory in [Plo 81] we get, that the solution, AT, is a ω-algebraic cpo with fAT as the set of finite elements. The elements of AT may be viewed as infinite versions of the finite trees in fAT. They are still finite branching but may have infinite depth.

To model the value-passing calculus we have to carry out some modifications. First we choose a different set of events. Thus we let

$$Ev = Chan? \cup Chan!$$

where $Chan?$ and $Chan!$ mean $\{c?|c \in Chan\}$ and $\{c!|c \in Chan\}$ respectively. Further instead of having elements from the cpo D as residuals in the trees we now have as a residual to a branch labelled by an input event, $c?$, a function which takes a value and returns an element in the cpo D. A residual to a branch labelled by an output event of the form $c!$ will be a finite lists of pairs of values and elements in D, or equivalently a finite partial function from the set of values, Val, into D. To capture this distinction between the input and output events we introduce the notation $f : A_1 \uplus A_2 \longrightarrow B_1 \uplus B_2$ where $\uplus$ is the disjoint union and whenever $a \in A_i$ then $f(a) \in B_i, i = 1, 2$. Now our tree constructor can be defined as

$$F(Ev, D) = H(Ev, G(D))_\perp$$

where

$$G(D) \quad = \quad (Val \longrightarrow D) \uplus Fin(Val \rightharpoonup D)$$

and $Fin(Val \rightharpoonup D)$ is the set of finite partial functions from Val to D and H is defined as before. Again $F(Ev, \_)$ maps a cpo into a cpo and thus can be turned into a continuous functor, again called F. Thus *The (Strong) Acceptance Trees with Value-Passing*, AT^v, can be defined as the least solution to the domain equation

$$\begin{array}{rcl} D & = & F(Ev, D) \\ Ev & = & Chan? \uplus Chan!. \end{array}$$

The two sets AT^v and $F(Ev, AT^v)$ are the "same" in the sense that they are isomorphic and to simplify the notation, we do not distinguish between an element in one of the sets and its isomorphic image in the other. Thus we say that a typical element in AT^v has either the form $\perp$ or the form $(\mathcal{A}, f)$ for some $\mathcal{A}$ and f. Again F preserves ω-algebraicity and therefore AT^v is an ω-algebraic cpo.

The finite elements of AT^v, fAT^v, are those who have "finite" functions as residuals, whereby we mean that the functions have finite domains and return finite elements as results. A formal definition of fAT^v is given below, where we use $Fin(Val \longrightarrow fAT^v)$ to denote the set of total functions from Val to fAT^v where only a finite number of values are mapped to something different from $\perp$. We also introduce the notation $G_{fin}(Val, D)$ for $Fin(Val \longrightarrow D) \uplus Fin(Val \rightharpoonup D)$.

Definition 2.3.1. fAT^v is the least set, which satisfies:

1. $\perp \in fAT^v$
2. if $\mathcal{A} \in sat(Ev)$ and $f : \bigcup \mathcal{A} \longrightarrow G_{fin}(Val, fAT^v)$ then $(\mathcal{A}, f) \in fAT^v$.

We also need a definition of AT^v_n, the *Acceptance Trees of depth and functional size n*. We let $Fin_n(Val \longrightarrow D) = \{f \in Fin(Val \longrightarrow D) | f(v) = \perp \text{ whenever } v \notin V_n\}$ and $G^n_{fin} = Fin_n(Val \longrightarrow D) \uplus Fin(Val \rightharpoonup D)$.

Definition 2.3.2. We define $AT^v_{(n)}$ by induction on n by

1. $\{\perp\} \in AT^v_{(n)}$ for all n
2. if $\mathcal{A} \in sat(Ev)$ and $f : \bigcup \mathcal{A} \longrightarrow G^n_{fin}(Val, AT_{(n)})$ then $(\mathcal{A}, f) \in AT^v_{(n+1)}$.

We have the following result:

Lemma 2.3.1.

1. $AT^v_0 \subseteq AT^v_1 \subseteq \cdots \subseteq fAT^v$
2. $fAT^v = \bigcup_n AT^v_n$.

Proof. The first part follows from an easy induction on n. To prove *2.* we first note that $\bigcup_n AT^v_n \subseteq fAT^v$ follows from *1.* Thus we only have to prove that for any $T \in fAT^v$ there is an n such that $T \in AT^v_n$ which is straightforward. □

To prove that fAT^v is exactly the set of finite elements in AT^v is the subject of the next theorem.

Theorem 2.3.1. fAT^v is the set of finite elements of AT^v.

Proof. We first prove that the elements of fAT^v are finite and then that these are the only finite elements of AT^v. The proof for the first part proceeds by induction on the definition of the elements in fAT^v. We have the following cases:

1. $\perp$ is obviously finite.
2. Assume $T = (\mathcal{A}, f) \in fAT^v$ where $\mathcal{A}$ is an acceptance set and $f \in \bigcup \mathcal{A} \longrightarrow G_{fin}(Val, fAT^v)$. Now assume that

   $$T \leq \bigsqcup_m T_m = T'$$

   where $T_1 \leq T_2 \leq \cdots \leq T'$. Let $T_m = (\mathcal{A}_m, f_m), m = 1, 2, \ldots$ and $T' = (\mathcal{A}', f')$. By the definition of the preorder

   $$\mathcal{A}' \subseteq \mathcal{A}_{m+1} \subseteq \mathcal{A}_m \subseteq \mathcal{A}, \text{ for } m = 1, 2, \ldots$$

   and

   $$f \leq f_m \leq f_{m+1} \leq f' \text{ for } m = 1, 2, \ldots$$

   As $\mathcal{A}$ is finite then $\mathcal{A} = \mathcal{A}_M$ for some M and thus

   $$f'(e) = \bigsqcup_{m>M} f_m(e), \ e \in \bigcup \mathcal{A}.$$

   Further, as $f(e) \leq f'(e)$ for all $e \in \bigcup \mathcal{A}$

   $$(f(e))(v) \leq \bigsqcup_{m>M} (f_m(e))(v), \ e \in \bigcup \mathcal{A}, v \in domain(f(e))$$

By induction $(f(e))(v) \leq (f_{m(e,v)}(e))(v)$ for some $m(e,v)$ for all $e \in \bigcup \mathcal{A}$ and $v \in domain(f(e))$. As $domain(f(e))$ and $\bigcup \mathcal{A}$ are finite, the set $E = \{(e,v) \mid e \in \bigcup \mathcal{A}, v \in domain(f(e))\}$ is finite. We can therefore define

$$N = max\{m(e,v) \mid (e,v) \in E\}$$

and we get easily that $f \leq f_N$ and the result follows.

Next we will prove that fAT^v are the only finite elements in AT^v. For this purpose for any $T \in AT^v$ we define $T^{(n)}$, the projection of T on $AT^v_{(n)}$. This we do in the following way:

1. $T^{(0)} = \bot$

2.a $\bot^{(n+1)} = \bot$

2.b $(\mathcal{A}, f)^{(n+1)} = (\mathcal{A}, f^{(n)})$, $n = 1, 2, \ldots$

where $f^{(n)}(c?)$ is defined by

$$(f^{(n)}(c?))(v) = \begin{cases} (f(c?)(v))^{(n)} & \text{if } v \in V_n \\ \bot & \text{otherwise} \end{cases}$$

and $f^{(n)}(c!)$ is defined by

$$domain(f^{(n)}(c!)) = domain(f(c!)) \cap V_n$$
$$(f^{(n)}(c!))(v) = ((f(c!))(v))^{(n)} \text{ for } v \in domain(f^{(n)}(c!))$$

By an easy induction we get that $T^{(n)} \in AT^v_{(n)}$ and that $T^{(0)} \leq T^{(1)} \leq \cdots T^{(n)} \leq T$ for all n. We will prove that T is the *lub* for the chain. To do so we have to refer to some general results about initial solutions in $\underline{CPO}$ ([Plo 81]).

Now we recall the standard definition of T^n, the n-th approximation of T:

1. $T^0 = \bot$

2.a $\bot^{n+1} = \bot$

2.b $(\mathcal{A}, f)^{n+1} = (\mathcal{A}, f^n)$, $n = 1, 2 \ldots$

where $f^n(c?)$ is defined by

$$f^n(c?) = (f(c?)(v))^n$$

and $f^n(c!)$ is defined by

$$domain(f^n(c!)) = domain(f(c!))$$
$$(f^n(c!))(v) = ((f(c!))(v))^n \text{ for } v \in domain(f^n(c!))$$

Note that the nth approximation, T^n, is in general not a finite element of the model. From the general theory we know that $T = \bigsqcup_n T^n$. Furthermore we can show that $\bigsqcup_n T^{(n)} = \bigsqcup_n T^n$. The "$\leq$" part follows from the obvious fact that $T^{(n)} \leq T^n$ for all n. We get the "$\geq$" part by showing that $T^m \leq \bigsqcup_n T^{(n)}$ for all m by induction on m as follows:

1. $T^0 \leq \bigsqcup_n T^{(n)}$ is obvious
2. Now we want to prove for any T that $T^{m+1} \leq \bigsqcup_n T^{(n)}$ given the statement is true for m. If $T = \bot$ we are done so assume T has the form $(\mathcal{A}, f)$. Then $T^{m+1} = (\mathcal{A}, f^m)$ and $T^{(n)} = (\mathcal{A}, f^{(n-1)})$ for $n = 1, \ldots$. We also can easily prove that $\bigsqcup_n (\mathcal{A}, f^{(n)}) = (\mathcal{A}, \bigsqcup_n f^{(n)})$ where $\bigsqcup_n f^{(n)}$ is defined by

$$((\bigsqcup_n f^{(n)})(e))(v) = \bigsqcup_n (f^{(n)}(e))(v).$$

By induction we get

$$(f^m(e))(v) = ((f(e))(v))^m \leq \bigsqcup_n ((f(e))(v))^{(n)} = \bigsqcup_n (f^{(n)}(e))(v).$$

As e and v are arbitrary this implies $f^m \leq \bigsqcup f^{(n)}$. Thus

$$T^{m+1} = (\mathcal{A}, f^m) \leq (\mathcal{A}, \bigsqcup_n f^{(n)}) = \bigsqcup_n (\mathcal{A}, f^{(n)}) = \bigsqcup_n T^{(n)}$$

as we wanted to prove. From this result we derive that

$$T = \bigsqcup_n T^n = \bigsqcup_n T^{(n)}.$$

The proof now proceeds as follows: Assume T is finite. From the result above we have that $T = \bigsqcup_n T^{(n)}$. By the definition of finite elements $T \leq T^{(N)}$ for some N. As we already know that $T^{(N)} \leq T$ then $T = T^{(N)}$. This proves that $T \in AT^v_{(N)}$ and thereby $T \in fAT^v$ which completes the proof. □

Now what is missing to turn AT^v into a natural interpretation for the language $VPLA$ is to define the functions *in* and *out* and the operators named in Σ and to show the continuity of these. We only give a few of these definitions here but the remainder can be found in the appendix. Regarding the proofs for the continuity we refer to [HI 89].

To make these definitions more compact we will denote AT^v by AT. Further we introduce the following convention : if h is a function from $AT \longrightarrow AT$ we also use h to denote the natural extension $ext(h) : (Val \longrightarrow AT) \longrightarrow (Val \longrightarrow AT)$ defined by

$$ext(h)(f_1, \ldots, f_n)(v) = h(f_1(v), \ldots, f_n(v)).$$

We use the notation $\sum\{T_1, \ldots, T_n\}$, $n \geq 1$ for $T_1 \oplus_{AT} \cdots \oplus_{AT} T_n$ and $\sum\{T_1 \ldots T_n\}$ for $T_1 +_{AT} \cdots +_{AT} T_n$, where the last sum reduces to $STOP_{AT}$ when $n = 0$. This notation will be justified by the associativity of the operators $\oplus_{AT}$ and $+_{AT}$ and the neutrality of $STOP_{AT}$ with respect to $+_{AT}$.

Output:

Define $out_{AT} : Chan \times Val \times AT \longrightarrow AT$ by
$out_{AT}(c, v, t) = \langle \{\{c!\}\}, f \rangle$
where
$domain(f) = \{c!\}$
$f(c!) = \{\langle v, t \rangle\}$.

Intuitively $out_{AT}(c, v, t)$ represents a process which can output the value v on the channel c and then act like the process t.

STOP:

Let $STOP_{AT}$ be the tree $\langle \{\{\}\}, f \rangle$, where

f is the empty function.

This is the terminated process which can perform no actions.

Ω :

Let Ω_{AT} be $\bot_{AT}$

This represents the undefined process.

Internal Nondeterminism:

Define $\oplus_{AT} : AT \times AT \longrightarrow AT$ as

$$Y\lambda I.\lambda t.\lambda u.\ \ if\ t = \bot\ or\ u = \bot\ then\ \bot\ else\ let\ \langle \mathcal{A}, f\rangle = t\ \langle \mathcal{B}, g\rangle = u\ in\ \langle c(\mathcal{A} \cup \mathcal{B}), h\rangle$$

where $c(X)$ is the least saturated set containing X and h is defined by:

$$\begin{aligned} h(c?) &= I(f(c?), g(c?)) \text{ if } c? \in domain(f) \cap domain(g) \\ &= f(c?) \text{ if } c? \in domain(f) \setminus domain(g) \\ &= g(c?) \text{ if } c? \in domain(g) \setminus domain(f) \end{aligned}$$

and

$$\begin{aligned} h(c!) &= f(c!) \text{ if } c! \in domain(f) \setminus domain(g) \\ &= g(c!) \text{ if } c! \in domain(g) \setminus domain(f) \\ &= k \text{ if } c! \in domain(f) \cap domain(g) \end{aligned}$$

where

$$\begin{aligned} k(v) &= I(f(c!)(v), g(c!)(v)) \text{ if } v \in domain(f(c!)) \cap domain(g(c!)) \\ &= f(c!)(v) \text{ if } v \in domain(f(c!)) \setminus domain(g(c!)) \\ &= g(c!)(v) \text{ if } v \in domain(g(c!)) \setminus domain(f(c!)) \end{aligned}$$

This is a recursive definition of a binary operator where I represents the function being defined. This function is strict in both arguments, returning $\bot$ unless the arguments are of the form $\langle \mathcal{A}, f\rangle, \langle \mathcal{B}, g\rangle$ respectively. In this case it returns a tree whose node is labelled by the acceptance set $c(\mathcal{A} \cup \mathcal{B})$ and whose residuals are calculated from those of f and g. For each event $e \in Ev$ the residual after e is obtained from f if e is in the domain of f and not in that of g, it is obtained from g if it is in the domain of g and not that of f while if it is in both domains it is obtained by recursively applying I.

2.4. Syntactically and Semantically Finite Terms

In the previous section we proved that each element of AT^v is determined by a sequence of finite trees which only use a finite number of values. It can be seen very easily that these elements are definable by syntactically finite terms in the language where "syntactically finite", as usual, means that the construction $rec.\_$ does not occur. In fact we can prove that these elements can be defined by syntactically finite terms where the input terms give non-trivial result for only a finite number of input values. We call these defining terms "semantically finite

terms" to distinguish them from the usual syntactically finite terms. We will in the following define formally the semantically finite terms and then show that they denote exactly the finite elements in the model, i.e. fAT^v.

Definition 2.4.1. We define the set of semantically finite terms as the least set, SF, which satisfies:

1. $STOP, \Omega \in SF$
2. $\underline{t} \in SF$ implies $op(\underline{t}) \in SF$
3. $t \in SF, c \in Chan$ and $e \in Exp$ implies $c!e.t \in SF$
4. $t \in SF, c \in Chan$, $x \in Var$ and $V \subseteq Val$ is finite implies $c?x.x \in V \rightarrow t, \Omega \in SF$
5. $t, u \in SF$ and $be \in BExp$ implies $be \rightarrow t, u \in SF$
6. $t \in SF, e \in Exp$ and $x \in Var$ implies $(x := e).t \in SF$

Here we assume that the language for expressions is sufficiently expressive to contain "$x \in V$"; since V is a finite set this is not very onerous. We have the following result:

Lemma 2.4.1.

1. For all $d \in SF$ $[\![d]\!] \in fAT^v$
2. For all $D \in fAT^v$ there is a $d \in SF$ such that $[\![d]\!] = D$.

Proof. An easy structural induction on d in the first case and and induction on the definition of D in the second one. □

In [HI 89] we defined the *syntactically finite approximations* of a term t, t^n, as the n-th unfolding of the recursive definitions and then we showed that the meaning of a term is completely defined by these approximations. As we do not have any restrictions on the input terms the syntactically finite approximations are not in general semantically finite. Therefore, to take advantage of the ω-algebraicness of the model, we introduce the so called semantically finite approximations, $t^{(n)}$, which are defined in a slightly different way in that now the n-th approximation of the input term $c?x.t$ only allows a finite number of inputs. In the following definition and in the rest of the paper we recall that $V_n, n = 1, \ldots$ are defined as in §2.1. Thus $Val = \bigcup_n V_n$, where $V_n \subseteq V_{n+1} \subseteq Val$ are finite.

Definition 2.4.2. (Semantically Finite Approximations) The n-th approximation of a term is defined inductively by following:

i) $t^{(0)} = \Omega$

ii)
1. $P^{(n+1)} = P$
2. $(op(\underline{t}))^{(n+1)} = op(\underline{t}^{(n+1)})$
3. $(c!e.t)^{(n+1)} = c!e.t^{(n+1)}$
4. $(c?x.t)^{(n+1)} = c?x.x \in V_{n+1} \rightarrow t^{(n+1)}, \Omega$
5. $(recP.t)^{(n+1)} = t^{(n+1)}[(recP.t)^{(n)}/P]$
6. $(be \rightarrow t, u)^{(n+1)} = be \rightarrow t^{(n+1)}, u^{(n+1)}$
7. $(x := e.t)^{(n+1)} = x := e.t^{(n+1)}$

Note that the only difference between the definition of the syntactically finite approximations from [HI 89], t^n, and the semantically finite ones, $t^{(n)}$, is in the

case $c?x.t$ as in the first case we have $(c?x.t)^n = c?x.t^n$. We will now show that the interpretation of a term in AT^v is completely defined by the meaning of the semantically finite approximations of the term. This is the content of the following theorem:

Theorem 2.4.1. For all $t \in VPLA, \sigma \in St$ and $\rho \in Env$

$$[\![t]\!]\rho\sigma = \bigsqcup_n [\![t^{(n)}]\!]\rho\sigma$$

Proof. The proof is very similar to the corresponding one for Theorem 4.2.11 in [He 88]. As there we proceed by structural induction. The only case which is different is $t = c?x.u$ and will therefore be given in detail.

By definition of $[\![\_]\!]$ and $t^{(n)}$

$$[\![c?x.u]\!]\rho\sigma = in(c, \lambda v.[\![u]\!]\rho(\sigma[v/x]))$$

and

$$[\![(c?x.u)^{(n)}]\!]\rho\sigma = [\![c?x.x \in V_n \rightarrow u^{(n)}, \Omega]\!]\rho\sigma = in(c, \lambda v.v \in V_n \rightarrow [\![u^{(n)}]\!]\rho(\sigma[v/x]), \perp)$$

Let f be $\lambda v.[\![u]\!]\rho(\sigma[v/x])$ and f_n be $\lambda v.v \in V_n \rightarrow [\![u^{(n)}]\!]\rho(\sigma[v/x]), \perp$. It is sufficient to prove that $f = \bigsqcup f_n$ where $\bigsqcup$ is the *lub* with respect to the usual pointwise order. Therefore take any $v \in Val$ then there is an N such that $v \in V_n$ for all $n \geq N$. Now we have by the structural induction

$$\begin{array}{lcl} f(v) & = & [\![u]\!]\rho\sigma[v/x] = \bigsqcup_n [\![u^{(n)}]\!]\rho\sigma[v/x] \\ & = & \bigsqcup_{n \geq N} [\![u^{(n)}]\!]\rho\sigma[v/x] = \bigsqcup_{n \geq N} f_n(v) \\ & = & \bigsqcup_n f_n(v) = (\bigsqcup_n f_n)(v) \end{array}$$

which completes the proof. □

2.5. The Proof System

In this last subsection we will define a proof system, E, for the language and prove the soundness and completeness for closed terms or processes.

As we have two kinds of variables, process variables and value variables, we also have two kinds of bindings. The construction $recP.t$ binds occurrences of the process name P in the term t and as we are not interested in terms with free process variables we assume that all occurrences of such variables are bound. In the construct $c?x.t$ and $(x := e).t$ the value-variable x is bound in t but e in $c!e.t$ and $(x := e).t$ may again contain free value-variables. Thus the set of free value-variables in a term t $FVar(t)$ is defined by:

$$\begin{array}{l} FVar(STOP) = FVar(\Omega) = \emptyset \\ FVar(op(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} FVar(t_i) \\ FVar(recP.t) = FVar(t) \\ FVar(c!e.t) = FVar(t) \cup FVar(e) \\ FVar(c?x.t) = FVar(t) \setminus \{x\} \\ FVar(x := e.t) = (FVar(t) \setminus \{x\}) \cup FVar(e) \\ FVar(be \rightarrow t, u) = FVar(be) \cup FVar(t) \cup FVar(u). \end{array}$$

A term, t, is said to be a *process* if it is closed, i.e. if $FVar(t) = \emptyset$.

The set of processes is denoted by $Proc$ and ranged over by p, q, etc.. For

$$
\begin{array}{rcl}
X \oplus (Y \oplus Z) & = & (X \oplus Y) \oplus Z \\
X \oplus Y & = & Y \oplus X \\
X \oplus X & = & X \\
X + (Y + Z) & = & (X + Y) + Z \\
X + Y & = & Y + X \\
X + X & = & X \\
X + STOP & = & X \\
pre.X + pre.Y & = & pre.(X \oplus Y) \\
c?x.X + c?x.Y & = & c?x.X \oplus c?x.Y \\
c!e.X + c!e'.Y & = & c!e.X \oplus c!e'.Y \\
X + (Y \oplus Z) & = & (X + Y) \oplus (X + Z) \\
X \oplus (Y + Z) & = & (X \oplus Y) + (X \oplus Z) \\
X \oplus Y & \sqsubseteq & X \\
X + \Omega & \sqsubseteq & \Omega \\
\Omega & \sqsubseteq & X \\
(X \oplus Y) \setminus c & = & X \setminus c \oplus Y \setminus c \\
(X + Y) \setminus c & = & X \setminus c + Y \setminus c \\
(pre.X) \setminus c & = & \begin{cases} pre.(X \setminus c) & \text{if } c \neq chan(pre) \\ STOP & \text{otherwise} \end{cases} \\
STOP \setminus c & = & STOP \\
\Omega \setminus c & = & \Omega \\
(X \oplus Y) \mid Z & = & X \mid Z \oplus Y \mid Z \\
X \mid (Y \oplus Z) & = & X \mid Y \oplus X \mid Z \\
STOP \mid X & = & X \mid STOP \; = \; X \\
X \mid (Y + \Omega) & = & (X + \Omega) \mid Y \; = \; \Omega
\end{array}
$$

Fig. 2. Equations

processes, the interpretation in the model is independent of the store and the environment and thus we can may write $[\![p]\!]$ instead of $[\![p]\!]\sigma\rho$.

The proof system is equationally based and is given in Figure 5 and the equations are given in Figure 2, 3 and 4. In the interleaving law in Figure 4 the predicate $comms(X, Y)$ is defined to be true if X and Y can communicate and false otherwise. So it is true only if there is a pair a_i, b_j of complementary actions, one of the form $c?x$ and the other $c!e$. In Rule V of Figure 5 we use η to

$$
\begin{array}{rcl}
(x := e).STOP & = & STOP \\
(x := e).\Omega & = & \Omega \\
(x := e).X \oplus Y & = & (x := e).X \oplus (x := e).Y \\
(x := e).X + Y & = & (x := e).X + (x := e).Y \\
(x := e).X \mid Y & = & (x := e).X \mid (x := e).Y \\
(x := e).c!e'.X & = & c!e'[e/x].(x := e).X \\
(x := e).c?y.X & = & c?y.(x := e).X,\ x \neq y, y \text{ not free in } e \\
(x := e).be \rightarrow X, Y & = & be[e/x] \rightarrow (x := e).X,\ (x := e).Y
\end{array}
$$

Fig. 3. Equations for Assignment

range over arbitrary substitutions of terms for variables. As usual we define $\sqsubseteq_E$ as the least relation which satisfies the rules and equations in Figures 2-5, and $t =_E u$ means $t \sqsubseteq_E u$ and $u \sqsubseteq_E t$.

The system is basically the same as the one for the applicative language VPL in [HI 89]. We only have added equations, Figure 3, to reason about assignment and one new inference rule, the second part of rule VII in Figure 5, to assure substitutivity for assignment. In Figure 3 the assignment prefixing does not affect the meaning of the processes $STOP$ and Ω. This reflects our ideas that stores and therefore bindings of variables to values can only be investigated by communication. Thus changing the value binding of a variable does not affect the process if it is not able to output the value of that variable. Further assignment distributes over the operators + and | which reflects our ideas of each subcomponent of the system having their private store only accessible by others via communication.

The new rules allow us to remove the assignments from any finite term and prove it equal to an assignment free finite term, i.e. a term in VPL.

Lemma 2.5.1. For all finite $d \in VPLA$ there is a finite $d' \in VPL$ such that $d = d'$.

Proof. Follows easily by structural induction on d and a repeated use of the equations in Figure 3 and Rule IX. □

Another significant difference from the proof system in [HI 89] is the definition of the finite approximations $t^{(n)}$ in the ω-rule (rule VI) as they are now only allowed to use a finite number of values. In [HI 89] we defined the n-th approximation of the term $c?x.u$ by $(c?x.u)^n = c?x.u^n$. Then we proved the completeness of the proof system by means of normal forms and head normal forms. We could have used the same procedure for the completeness proof in

Let X, Y denote $\sum\{a_i.X_i, i \in I\}, \sum\{b_j.Y_j, j \in J\}$ where the same data variables do not occur in X and Y. Then

$$X \mid Y = \begin{cases} ext(X,Y) & \text{if } comms(X,Y) = false \\ (ext(X,Y) + int(X,Y)) \oplus int(X,Y) & \text{otherwise} \end{cases}$$

where

$$ext(X,Y) = \sum\{a_i.(X_i \mid Y),\ i \in I\} + \sum\{b_j.(X \mid Y_j),\ j \in J\}$$
$$int(X,Y) = \sum\ \{X_i[v/x] \mid Y_j,\ a_i = c?x,\ b_j = c!v\}$$
$$\oplus \sum\ \{X_i \mid Y_j, [v/y]\ a_i = c!v,\ b_j = c?y\}$$

Fig. 4. Interleaving Law

I	$t \sqsubseteq t$	$\dfrac{t \sqsubseteq u,\ u \sqsubseteq v}{t \sqsubseteq v}$
II	$\dfrac{t_i \sqsubseteq u_i}{op(\underline{t}) \sqsubseteq op(\underline{u})}$ for every $op \in \sum$	
III	$\dfrac{t \sqsubseteq u}{c!e.t \sqsubseteq c!e.u}$	$\dfrac{t[v/x] \sqsubseteq u[v/x] \text{ for every } v \in V}{c?x.t \sqsubseteq c?x.u}$
IV	$\dfrac{t \sqsubseteq u}{recP.t \sqsubseteq recP.u}$	$\dfrac{}{recP.t = t[recP.t/P]}$
V	$\dfrac{t \sqsubseteq u}{t\eta \sqsubseteq u\eta}$	$\dfrac{}{t \sqsubseteq u}$ for every equation $t \sqsubseteq u$
VI	$\dfrac{\forall n.t^{(n)} \sqsubseteq u}{t \sqsubseteq u}$	
VII	$\dfrac{[\![e]\!] = [\![e']\!]}{c!e.t = c!e'.t}$	$\dfrac{[\![e]\!] = [\![e']\!]}{x := e.t = x := e'.t}$
VIII	$\dfrac{[\![be]\!] = T}{be \rightarrow t, u = t}$	$\dfrac{[\![be]\!] = F}{be \rightarrow t, u = u}$
IX	$\dfrac{}{c?x.t = c?y.t[y/x]}$ if y does not occur free in t	

Fig. 5. Proof System

this case; instead we have chosen to define the n-th approximations $t^{(n)}$ in this slightly different way which can be seen in Definition 2.4.2. Unlike in the previous paper $t^{(n)}$ is semantically finite and thus the new ω-rule

$$\frac{\forall n.\, t^{(n)} \sqsubseteq u}{t \sqsubseteq u}$$

reflects the ω-algebraicness in the model. We can therefore use the completeness of the old system for finite terms and then take advantage of the ω-algebraicness of the model to prove the completeness in the general case.

Obviously the original definition of finite approximations, t^n in [HI 89], dominates the new ones, i.e. we can prove that for all $t \in VPLA$, $t^{(n)} \sqsubseteq_E t^n$ without using the ω–rule and therefore the new ω-rule implies the original one

$$\frac{\forall n.\, t^n \sqsubseteq u}{t \sqsubseteq u}$$

The new system is thus stronger than the old one, and a completeness for the old one implies completeness for the new one. We will take advantage of this fact to prove the completeness of the new proof system.

We end this section by stating and proving the soundness and completeness of the system for processes with respect to AT^v. We start with the soundness.

Theorem 2.5.1. (Soundness) For all t, u

$$t \sqsubseteq_E u \text{ implies } [\![t]\!] \leq [\![u]\!].$$

Proof. The soundness is already proved for most of the rules and equations in [HI 89]. The soundness of the ω-rule is the content of Theorem 2.4.1 and the soundness of the remaining ones is obvious. □

For the completeness we start by proving the result for finite processes and then show how the general result follows from this as an easy corollary.

Lemma 2.5.2. For all finite $d_1, d_2 \in Proc$

$$[\![d_1]\!] \leq [\![d_2]\!] \text{ implies } d_1 \sqsubseteq_E d_2.$$

Proof. Suppose $d_1, d_2 \in Proc$ are syntactically finite and $[\![d_1]\!] \leq [\![d_2]\!]$. By Lemma 2.5.1 , $d_1 =_E d_1'$ and $d_2 =_E d_2'$ for some closed finite $d_1', d_2' \in VPL$. By the soundness of the proof system $[\![d_1]\!] = [\![d_1']\!]$ and $[\![d_2]\!] = [\![d_2']\!]$. Now we note that our proof system is an extension of the proof system introduced in §4 in [HI 89]. Thus we can use the completeness of that proof system to deduce that $d_1' \sqsubseteq_E d_2'$ and the result follows from the transitivity of the proof system. This completes the proof. □

Next we will use this partial result to prove the general completeness.

Corollary 2.5.1. (completeness) For all $p, q \in Proc$

$$[\![p]\!] \leq [\![q]\!] \text{ implies } p \sqsubseteq_E q.$$

Proof. We will prove that the general completeness follows from the previous

Lemma and the ω-algebraicity of the model. Thus assume that $[\![p]\!] \leq [\![q]\!]$. From Lemma 2.4.1 we get that $[\![p]\!] = \bigsqcup_n [\![p^{(n)}]\!]$ and $[\![q]\!] = \bigsqcup_n [\![q^{(n)}]\!]$. Therefore $\bigsqcup_n [\![p^{(n)}]\!] \leq \bigsqcup_n [\![q^{(n)}]\!]$ which implies that

$$\forall k.\, [\![p^{(k)}]\!] \leq \bigsqcup_n [\![q^{(n)}]\!]$$

As $[\![p^{(k)}]\!]$ is a finite element of the model we have

$$\forall k \exists m.\, [\![p^{(k)}]\!] \leq [\![q^{(m)}]\!].$$

By the already proved completeness for finite terms

$$\forall k \exists m.\, p^{(k)} \sqsubseteq_E q^{(m)}.$$

As we know that $q^{(m)} \sqsubseteq_E q$ this implies

$$\forall k .p^{(k)} \sqsubseteq_E q$$

and the result follows from the ω-rule. □

3. Operational Semantics and Full Abstractness

In this section we define an operational semantics for our language and introduce the notion of testing. The operational semantics is defined in such a way that it captures our intuition of the behaviour of processes or configurations described in the introduction, i.e. that each subprocess of a system has its own private store only accessible by other processes by means of communication. For a further justification of our different semantic approaches we show that the model AT_s^v is fully abstract with respect to the resulting testing preorder.

3.1. Operational semantics

We are now no longer dealing with a pure applicative language but with a language with assignments which aims to change the bindings of the variables. We follow the standard approach and use a *store*, a total function from the set of variables into the set of values, to keep track of the bindings of the variables to values. Thus our operational semantics should describe how a configuration, a pair of a process and a store, evolves to another configuration by either an internal or an external move. We denote a configuration by $\langle t, \sigma \rangle$ where t is a term and σ is a store. Note that, unlike in [HI 89], the terms may now contain free data variables as the store takes care of the bindings.

Updating the store is not supposed to be a visible action and is therefore modelled as an internal move. In the standard approach each argument of the operators + and | can move internally without affecting the structure. Thus we have

$$p \succ\!\!\rightarrow p' \text{ implies } p \,|\, q \succ\!\!\rightarrow p' | q$$

and

$$p \succ\!\!\rightarrow p' \text{ implies } p + q \succ\!\!\rightarrow p' + q.$$

In following this standard approach a problem arises when we have to deal with these operators. In both cases each of the components can update the store internally and thus affect the variable bindings for the other. Let us have a look at an example. We use the notation $\sigma[x_1/v_1, \ldots, x_n/v_n]$ for the store σ' defined by $\sigma'(x) = v_i$ if $x = x_i, i = 1, \ldots, n$ and $\sigma'(x) = \sigma(x)$ otherwise. Let

$$t = c!x.STOP \mid x := 1.STOP$$
$$\sigma' = \sigma[0/x].$$

How is the store to be updated according to the assignment statement? We might either have

$$\begin{array}{rl} \langle c!x.STOP \mid x := 1.STOP, \sigma[0/x]\rangle & \stackrel{c!0}{\longrightarrow} \\ \langle STOP \mid x := 1.STOP, \sigma[0/x]\rangle & \succ\!\!\rightarrow \\ \langle STOP \mid STOP, \sigma[1/x]\rangle & \end{array}$$

or

$$\begin{array}{rll} \langle c!x.STOP \mid x := 1.STOP, \sigma[0/x]\rangle & \succ\!\!\rightarrow & \\ \langle c!x.STOP \mid STOP, \sigma[1/x]\rangle & \stackrel{c!1}{\longrightarrow} & \langle STOP \mid STOP, \sigma[1/x]\rangle. \end{array}$$

Thus by directly outputting on the channel c by the first component we get

$$\langle t, \sigma[0/x]\rangle \stackrel{c!0}{\Longrightarrow} \langle STOP \mid STOP, \sigma[0/x]\rangle$$

or alternatively we get

$$\langle t, \sigma[0/x]\rangle \stackrel{c!1}{\Longrightarrow} \langle STOP \mid STOP, \sigma[1/x]\rangle$$

by a side-effect when we first apply the assignment statement of the second component and then output the updated value on the channel c of the first component. Here $\stackrel{a}{\Longrightarrow}$ is the weak transition relation derived from $\stackrel{a}{\longrightarrow}$ by abstracting from the internal moves. This proposed possibility of different computations conflicts with our intuitions. We expect each component to have its own independent store which is not accessible by other processes. Thus in our example each of the subprocesses $c?x.STOP$ and $x := 1.STOP$, when refering to x, are refering to x of its own local store. To reflect this in the operational semantics we create a new copy of the store for each of the operator's argument by an internal move and distribute them over the operator, and thus introduce a parallel composition for configurations instead of terms. Now each copy of the store becomes completely local to each of the arguments and can be updated independently of each other. The only way two parallel configurations can access each other stores is via communication. Our example now becomes

$$\begin{array}{rl} \langle c!x.STOP \mid x := 1.STOP, \sigma[0/x]\rangle & \succ\!\!\rightarrow \\ \langle c!x.STOP, \sigma[0/x]\rangle \mid \langle x := 1.STOP, \sigma[0/x]\rangle & \stackrel{c!0}{\longrightarrow} \\ \langle STOP, \sigma[0/x]\rangle \mid \langle x := 1.STOP, \sigma[0/x]\rangle & \succ\!\!\rightarrow \\ \langle STOP, \sigma[0/x]\rangle \mid \langle STOP, \sigma[1/x]\rangle. & \end{array}$$

That is

$$\langle t, \sigma[0/x]\rangle \stackrel{c!0}{\Longrightarrow} \langle STOP, \sigma[0/x]\rangle \mid \langle STOP, \sigma[1/x]\rangle.$$

This is the only possible external move by the first component and is completely independent of the internal moves of the second one.

The same solution can be used to solve the similar problem related to external

choice outlined in the introduction. Furthermore this solution coincides with the definition of the denotational semantics in the sense that there the stores are distributed over the operators and assignment in one component does not have any influence on the semantics of another component.

In the following we will try to formalise these informal ideas of the operational behaviour of processes, or more precisely configurations.

As motivated above, we need to introduce external choice and parallel composition between configurations. Thus more complex configurations are built up from the basic ones, which only consist of a pair of a term and a store. To simplify the rules for the operational semantics we also introduce the operators $\oplus$, $\backslash c$ and prefixing for configurations. This leads to the following definition of configurations:

Definition 3.1.1. The set of basic configurations, BCon, is defined as:

$$BCon = \{\langle t, \sigma\rangle \mid t \in VPLA, \sigma \in St\}.$$

Now we define the set of configurations, Con, as the least set, which satisfies

1) $BCon \subseteq Con$
2) $\langle t, \sigma\rangle \in BCon$ implies $pre.\langle t, \sigma\rangle \in Con$
3) $\alpha \in Con$ implies $op(\alpha) \in Con$
   for all $op \in \Sigma_1$
   $\alpha_1, \alpha_2 \in Con$ implies $op(\alpha_1, \alpha_2) \in Con$
   for all $op \in \Sigma_2$.

We let ϕ, ψ, etc. range over the set $BCon$ and α, β, etc. over the set Con. The operational semantics can now be given in the same way as in [HI 89]. We define an extended labelled transition system $\langle Con, Act, \longrightarrow, \succ\!\!\rightarrow\rangle$ where

- Act is a set of actions
- $\longrightarrow \subseteq Con \times Act \times Con$
- $\succ\!\!\rightarrow \subseteq Con \times Con$

$(\alpha, a, \beta) \in \longrightarrow$ is usually written $\alpha \xrightarrow{a} \beta$ and intuitively it means that the configuration α may perform the action a and thereby be transformed into β; $\alpha \succ\!\!\rightarrow \beta$ may be read as "α may evolve spontaneously to β".

The set Act consists of all input events of the form $c?v$ and all output events of the form $c!v$ where $c \in Chan$ and $v \in Val$. The relations $\xrightarrow{a}$ and $\succ\!\!\rightarrow$ are defined to be the least which satisfy the rules given in Figures 6 and 7. As in the denotational semantics, these rules presuppose an evaluation mechanism for expressions : $[\![e]\!]\sigma$ gives the value in Val of the expression e in the store σ and $[\![be]\!]\sigma$ returns either T or F. The rule for communication, Rule 9 Figure 6, uses a complementation notation for actions, $\overline{c?v}$ is $c!v$ and $\overline{c!v}$ is $c?v$. The rules for channel hiding also use the obvious notation $name(a) \neq c$ to indicate that the action a does not use the channel c.

The rules themselves are quite straightforward. Rules 5 and 6 in Figure 6 create a structured configuration from a basic one by distributing the store over the operators, the one for assignment updates the store as the rule for conditionals chooses one of the components according to the value of the boolean expression. The remainder are directly from [HI 89] but now on configurations instead of terms.

As before the resulting labelled transition system is not in general finite

1. $\langle \Omega, \sigma \rangle \succ\!\!\rightarrow \langle \Omega, \sigma \rangle$

2. $\langle recP.t, \sigma \rangle \succ\!\!\rightarrow \langle t[recP.t/P], \sigma \rangle$

3. $\langle x := e.t, \sigma \rangle \succ\!\!\rightarrow \langle t, \sigma[v/x] \rangle$, where $v = [\![e]\!]\sigma$

4. $[\![be]\!]\sigma = T$ implies $\langle be \rightarrow t, u, \sigma \rangle \succ\!\!\rightarrow \langle t, \sigma \rangle$

   $[\![be]\!]\sigma = F$ implies $\langle be \rightarrow t, u, \sigma \rangle \succ\!\!\rightarrow \langle u, \sigma \rangle$

5. $\langle op(t_1, \ldots, t_k), \sigma \rangle \succ\!\!\rightarrow op(\langle t_1, \sigma \rangle, \ldots, \langle t_k, \sigma \rangle)$, $op \in \Sigma_1 \cup \Sigma_2$

6. $\langle c?x.t, \sigma \rangle \succ\!\!\rightarrow c?x.\langle t, \sigma \rangle$
   $\langle c!e.t, \sigma \rangle \succ\!\!\rightarrow c!v.\langle t, \sigma \rangle$ where $v = [\![e]\!]\sigma$

7. $\alpha \oplus \beta \succ\!\!\rightarrow \alpha$
   $\succ\!\!\rightarrow \beta$

8. $\alpha_i \succ\!\!\rightarrow \alpha_i'$ implies $op(\alpha_1, \ldots, \alpha_i, \ldots, \alpha_k) \succ\!\!\rightarrow op(\alpha_1, \ldots, \alpha_i', \ldots, \alpha_k)$
   $op \in \Sigma_1 \cup \Sigma_2 \setminus \{\oplus\}$

9. $\alpha \xrightarrow{a} \alpha', \beta \xrightarrow{\overline{a}} \beta'$ implies $\alpha \mid \beta \succ\!\!\rightarrow \alpha' \mid \beta'$
   $\beta \mid \alpha \succ\!\!\rightarrow \beta' \mid \alpha'$

Fig. 6. Rules for $\succ\!\!\rightarrow$

branching as any configuration of the form $c?x.\langle t, \sigma \rangle$ can have an infinite number of derivations since for every $v \in Val$ $c?x.\langle t, \sigma \rangle \xrightarrow{c?v} \langle t, \sigma[x/v] \rangle$. However we have the following results similar to that in [HI 89]. First some notation:

$$\begin{aligned} InC(\alpha) &= \{c \mid \exists \alpha', v.\alpha \xrightarrow{c?v} \alpha'\} \\ OutD(\alpha) &= \{\alpha' \mid \exists c, v.\alpha \xrightarrow{c!v} \alpha'\} \\ IntD(\alpha) &= \{\alpha' \mid \alpha \succ\!\!\rightarrow \alpha'\} \\ D(\alpha, a) &= \{\alpha' \mid \alpha \xrightarrow{a} \alpha'\} \end{aligned}$$

Theorem 3.1.1. For every α in Con, $InC(\alpha)$, $OutD(\alpha)$ and $IntD(\alpha)$ are finite.

Proof. By structural induction on α. Note that the case $\langle recP.t, \sigma \rangle$ is trivial since $IntD(\langle recP.t, \sigma \rangle) = \{\langle t[recP.t/P], \sigma \rangle\}$ and $InC(\langle recP.t, \sigma \rangle) = OutD(\langle recP.t, \sigma \rangle) = \emptyset$. The only nontrivial case is when α has the form $\alpha \mid \beta$, but the proof for this case is the same as the corresponding one in [HI 89], Theorem 2.2.1 and is therefore omitted. □

The following lemma follows immediately from the definition of the operational semantics:

1. $c?x.\langle t,\sigma\rangle \xrightarrow{c?v} \langle t,\sigma[v/x]\rangle$ for any $v \in Val$
   $c!v.\langle t,\sigma\rangle \xrightarrow{c![\![e]\!]} \langle t,\sigma\rangle$
2. $\alpha \xrightarrow{a} \alpha'$ implies $\alpha + \beta \xrightarrow{a} \alpha'$
   $\beta + \alpha \xrightarrow{a} \alpha'$
3. $\alpha \xrightarrow{a} \alpha'$ implies $\alpha \mid \beta \xrightarrow{a} \alpha' \mid \beta$
   $\beta \mid \alpha \xrightarrow{a} \beta \mid \alpha'$
4. $\alpha \xrightarrow{a} \alpha'$ implies $\alpha \setminus c \xrightarrow{a} \alpha' \setminus c$
   if $name(a) \neq c$

Fig. 7. Rules for $\xrightarrow{a}$

Lemma 3.1.1. For all $\phi \in BCon$, other than $\langle STOP,\sigma\rangle$,

1. there exists exactly one α where $\phi \rightarrowtail \alpha$
2. $\phi \not\xrightarrow{a}$ for all a.

3.2. Testing

In this section we extend the general theory of testing from [He 88] and [HI 89] to the language *VPLA*. As in [HI 89] we will only look at the *MUST* case. The main difference is that now testing is defined by interaction between configurations instead of terms. Thus a test is defined as a configuration which may use, in addition to the channels in $Chan$, a special channel w for reporting success. We let tests be ranged over by e and say α *must* e, where α is a configuration if in every maximal computation

$$\alpha|e = \alpha_0|e_0 \rightarrowtail \alpha_1|e_1 \cdots \alpha_k|e_k \rightarrowtail \cdots$$

there exists some $n \geq 0$ such that e_n is successful, i.e. $e_n \xrightarrow{w!v}$ for some value v. Note that by this definition the test has no access to the private memory of the configuration other than what can be obtained by communication. Thus the content of the stores of the configuration in the final state of a computation can never be checked.

We define $\alpha \sqsubseteq_M \beta$ if for every test e,

$$\alpha \textit{ must } e \text{ implies } \beta \textit{ must } e.$$

For two terms t, u, we let $t \sqsubseteq_M u$ if for all stores, σ, $\langle t,\sigma\rangle \sqsubseteq_M \langle u,\sigma\rangle$. Further we let $\simeq_M$ denote the equivalence relation over configurations/terms obtained by taking the kernel of $\sqsubseteq_M$ over configurations/terms.

To simplify reasoning about the operational behaviour we give an alternative characterisation of $\sqsubseteq_M$ in terms of the operational semantics of configurations. This is essentially the same as the alternative characterisation in [HI 89]. As there the actions are of the form $c?v$ or $c!v$ and when representing internal states using acceptance sets we need only remember the names of the channels along which an output can be sent or an input received but not the actual values themselves. As in §2.3 these will be called events.

$$Ev = \{c! \mid c \in Chan\} \cup \{c? \mid c \in Chan\}.$$

Now acceptance sets will be finite collections of finite sets of events. Note the difference to the acceptance sets in the definition of the Acceptance Trees. Here the acceptance sets are not necessarily saturated.

To define the alternative characterisation we need some notation, taken directly from [HI 89].

- For $s \in Act^*$ define $\alpha \stackrel{s}{\Longrightarrow} \beta$ by

  i) $\alpha \stackrel{\varepsilon}{\Longrightarrow} \beta$ if $\alpha \rightarrowtail^* \beta$

  ii) $\alpha \stackrel{as'}{\Longrightarrow} \beta$ if for some α', α'', $\alpha \stackrel{\varepsilon}{\Longrightarrow} \alpha'$, $\alpha' \stackrel{a}{\longrightarrow} \alpha''$ and $\alpha'' \stackrel{s'}{\Longrightarrow} \beta$

- $L(\alpha) = \{s \mid \text{for some } \beta,\ \alpha \stackrel{s}{\Longrightarrow} \beta\}$
- Define $\downarrow$, $\downarrow s$, $\uparrow$ and $\uparrow s$ by:

  i) $\alpha \downarrow$ if there is no infinite internal computation

  $$\alpha = \alpha_0 \rightarrowtail \alpha_1 \rightarrowtail \cdots$$

  ii) $\alpha \downarrow \varepsilon$ if $\alpha \downarrow$

  iii) $\alpha \downarrow as'$ if $\alpha \downarrow$ and if $\alpha \stackrel{a}{\Longrightarrow} \alpha'$ then $\alpha' \downarrow s'$

  iv) $\alpha \uparrow$ if $\alpha \downarrow$ is false and $\alpha \uparrow s$ if $\alpha \downarrow s$ is false.

- Define $S(\alpha) \subseteq Ev$ by

  $$S(\alpha) = \{c? \mid \text{ for some } v,\ \alpha \stackrel{c?v}{\Longrightarrow}\} \cup \{c! \mid \text{ for some } v,\ \alpha \stackrel{c!v}{\Longrightarrow}\}$$

- Define $\mathcal{A}(\alpha, s)$, the acceptance set of events of α after s by:

  $$\mathcal{A}(\alpha, s) = \{S(\alpha') \mid \alpha \stackrel{s}{\Longrightarrow} \alpha'\}.$$

Definition 3.2.1. For $\alpha, \beta \in Con$, $\alpha \ll_M \beta$ if for every $s \in Act^*$

$$\alpha \downarrow s \implies \begin{array}{l} \text{a) } \beta \downarrow s \\ \text{b) } \mathcal{A}(\beta, s) \subset\subset \mathcal{A}(\alpha, s) \end{array}$$

where $\mathcal{A} \subset\subset \mathcal{B}$ is defined by

$$\begin{array}{l} \text{for every } A \in \mathcal{A} \text{ there is some} \\ B \in \mathcal{B} \text{ such that } B \subseteq A. \end{array}$$

Note that if $\mathcal{A}$ and $\mathcal{B}$ are saturated then $\mathcal{A} \subset\subset \mathcal{B}$ reduces to $\mathcal{A} \subseteq \mathcal{B}$, i.e. an ordinary set inclusion, under the assumption that $\bigcup \mathcal{A} = \bigcup \mathcal{B}$.

In this definition of $\ll_M$ the configurations play the role of processes in the definition of $\ll_M$ in [HI 89]. Thus we can state an alternative characterization theorem in the same way as in that paper. Although the proof is essentially the same as the corresponding proof in [HI 89] we will give the details of the "if" part

of it to demonstrate how the existing proofs can be reused with configurations in the role of processes.

Theorem 3.2.1. For all $\alpha, \beta \in Con$

$\alpha \ll_M \beta$ if and only if $\alpha \sqsubseteq_M \beta$.

Proof. In the following we use $1.w$ as a shorthand notation for the term

$$(int?x.w!0.STOP \mid int!0.STOP) \setminus int.$$

This represents a process which is not yet in a successful state but can spontaneously evolve to such a state.

For each $s \in Act^*$ and $a \in Act$ define the terms $\mathbf{c}(s)$ and $\mathbf{e}(s,a)$ as follows:

i) $\mathbf{c}(\varepsilon) = 1.w$
$\mathbf{c}(c?v.s) = c!v.\mathbf{c}(s) + 1.w$
$\mathbf{c}(c!v.s) = (c?x.\ x = v\ \rightarrow \mathbf{c}(s), 1.w) + 1.w$

ii) $\mathbf{e}(\varepsilon, c?v) = c!v.STOP + 1.w$
$\mathbf{e}(\varepsilon, c!v) = (c?x.\ x = v\ \rightarrow STOP, 1.w) + 1.w$
$\mathbf{e}((c?v)s, a) = c!v.\mathbf{e}(s,a) + 1.w$
$\mathbf{e}((c!v)s, a) = (c?x.\ x = v\ \rightarrow \mathbf{e}(s,a), 1.w) + 1.w$

Take any store σ. We define the tests $c(s) = \langle \mathbf{c}(s), \sigma \rangle$ and $e(s,a) = \langle \mathbf{e}(s,a), \sigma \rangle$. Note that as the term is closed these tests are independent of the store σ. With these definitions one can show that

α *must* $c(s)$ if and only if $\alpha \downarrow s$

and

if $\alpha \downarrow s$ then α *must* $e(s,a)$ if and only if $sa \notin L(\alpha)$.

As a corollary we have

if $\alpha \sqsubseteq_M \beta$ and $\alpha \downarrow s$ then $\beta \downarrow s$

and

if $\alpha \sqsubseteq_M \beta$ then $\alpha \downarrow s$ and $s \in L(\beta)$ implies $s \in L(\alpha)$.

Now suppose $\alpha \sqsubseteq_M \beta$ we show that $\alpha \ll_M \beta$. Assume $\alpha \downarrow s$. This implies $\beta \downarrow s$. Let $A \in \mathcal{A}(\beta, s)$. From above $\mathcal{A}(\alpha, s)$ is not empty and, since $\alpha \downarrow s$, it is finite, say $\mathcal{A}(\alpha, s) = \{B_1, \ldots, B_n\}$. We have to show, that $B_i \subseteq A$ for some i. Assume that this is not true. This means $B_i \setminus A \neq \emptyset$ for all i and we can choose $b_i \in B_i \setminus A$ for $i = 1, \ldots, n$. Now for $e \in Ev$ and $L \subseteq Ev$ let $\mathbf{b}(e)$ be defined by

$$\begin{aligned} \mathbf{b}(c?) &= c!0.w!0.STOP \\ \mathbf{b}(c!) &= c?x.w!0.STOP \end{aligned}$$

$\mathbf{b}(s, L)$ by

$$\begin{aligned} \mathbf{b}(\varepsilon, L) &= \Sigma\{\mathbf{b}(a) \mid a \in L\} \\ \mathbf{b}((c?v)s, L) &= c!v.\mathbf{b}(s, L) + 1.w \\ \mathbf{b}((c!v)s, L) &= (c?x.x = v \rightarrow \mathbf{b}(s, L),\ w!0.STOP\ ) +\ 1.w \end{aligned}$$

and $b(s, L)$ by

$$b(s, L) = \langle \mathbf{b}(s, L), \sigma \rangle$$

Then $\alpha \underline{must}\, b(s,B)$ where $B = \{b_1, \ldots, b_n\}$, but $\beta \not\underline{must}\, b(s,B)$ because of the unsuccessful computation

$$b(s,B)|\beta \rightarrowtail^* b(\varepsilon,B)|\gamma$$

where $\beta \stackrel{s}{\Longrightarrow} \gamma$ and $S(\gamma) = A$. □

As a corollary to the theorem we have the following:

Corollary 3.2.1. $\sqsubseteq_M$ over configurations is preserved by all the operations in Σ.

Proof. Similar to the proof for the corresponding result in [HI 89]. □

3.3. Full Abstractness

This last subsection is devoted to the proof of the full abstractness for configurations of the denotational model AT^v with respect to the testing preorder $\sqsubseteq_M$. First we have to extend the definition of the interpretation of terms to that of configurations. We write $[\![\_]\!]$ instead of $AT^v[\![\_]\!]$.

Definition 3.3.1. The semantics of configurations is given as a function:

$$[\![\ ]\!] : Con \longrightarrow (Env_D \longrightarrow D)$$

defined by:

i) $[\![\langle t, \sigma\rangle]\!]\rho = [\![t]\!]\rho\sigma$

ii) $[\![op(\underline{\alpha})]\!]\rho = op([\![\underline{\alpha}]\!]\rho)$.

Also the definition of the finite approximations extends to configurations in the obvious way, and we can easily deduce that the meaning of a configuration is the limit of the meaning of its finite approximations:

$$[\![\alpha]\!] = \bigsqcup\{[\![\alpha^{(n)}]\!] \mid n \geq 1\}.$$

By full abstractness of the model we mean as usual

$$[\![\alpha]\!] \leq [\![\beta]\!] \quad \Leftrightarrow \quad \alpha \sqsubseteq_M \beta$$

for all configurations α, β.

The proof follows very much the same lines as the corresponding one for the applicative case in [HI 89]. In the following we will outline the proof of full abstractness in [HI 89] and show how our new theory can be fitted into this existing proof which thus can be more or less reused. Recall that the denotational model is the same and that the configurations in the new settings play the role of processes in the previous one.

In [HI 89] we defined a transition relation and a divergence predicate in AT^v, and from that deduced an alternative characterization for the preorder $\leq$, called $\ll_M$. As we use the same model we can use the same definition. Thus we define the transition relation by

$$(\mathcal{A}, f) \stackrel{c*v}{\longrightarrow} T \text{ if } * \in \{!, ?\}, c* \in \bigcup \mathcal{A} \text{ and } f(c*)(v) = T$$

The divergence predicate, $\uparrow$, is defined by letting $\bot \uparrow$ and extending it in the usual way for $s \in Act^*$. $\downarrow s$ denotes the the negation of $\uparrow s$. The acceptance set of a tree after s, $\mathcal{A}(T,s)$ is defined by:

i) $\mathcal{A}(\bot, \varepsilon) = \emptyset$
$\mathcal{A}((\mathcal{B}, f), \varepsilon) = \mathcal{B}$

ii) $\mathcal{A}(T, as) = \begin{cases} \mathcal{A}(T', s) & \text{if } T \xrightarrow{a} T' \\ \emptyset & \text{otherwise} \end{cases}$

Now we define the preorder $\ll_M$ on AT^v by:

For all $T, U \in AT^v$ let $T \ll_M U$ if for every $s \in Act^*$,

$T \downarrow s \Rightarrow$ $i) U \downarrow s$
$ii) \mathcal{A}(U,s) \subseteq \mathcal{A}(T,s)$

Further we have from [HI 89], that $\ll_M = \leq$ in AT^v. We have also proved that for configurations $\ll_M = \sqsubseteq_M$ and therefore, for the full abstractness, it is sufficient to prove that

$$[\![\alpha]\!] \ll_M [\![\beta]\!] \quad \Leftrightarrow \quad \alpha \ll_M \beta.$$

This follows as an immediate consequence of the following two properties:

For all $\alpha \in Con$ and $s \in Act^*$

i) $[\![\alpha]\!] \downarrow s$ if and only if $\alpha \downarrow s$
ii) if $\alpha \downarrow s$ then $\mathcal{A}([\![\alpha]\!], s) = c(\mathcal{A}(\alpha, s))$

We have therefore reduced the proof of the full abstractness to the proof of of $i)$ and $ii)$. To prove this we again rely on the similarities to [HI 89].

To prove i) above in the mentioned paper we introduced the notion of weak sum forms and proved that each finite process can be rewritten via the proof system to such a weak sum form. Similarly we used the notion of head normal forms and a normalization theorem for convergent processes to prove ii). To reuse this proof we have to define the notion of weak sum forms and head normal forms such that they fit into the original proof. But first we define a proof system over configurations. We will use this proof system to transform configurations to weak normal forms and head normal forms.

The equations are now of two types. The first type consists of the equations already introduced for processes in §2.5, Figure 2 and 4. The variables are now supposed to be instantiated by configurations instead of processes. Also $STOP$ and Ω are instantiated by $\langle STOP, \sigma \rangle$ for arbitrary σ. The second type (Figure 8) gives the connection between the semantics of basic configurations (of the form $\langle t, \sigma \rangle$) and the more complex ones. Here the variables are instantiated by terms. The inference rules are given in Figure 9 and are the same as those introduced in §2.5 except that the rules concerning recursion are omitted. The soundness of the whole system is with respect to configurations in the first two sets of equations and the inference rules but with respect to terms in the third set of equations. We call this new system A and the corresponding preorder $\sqsubseteq_A$.

Theorem 3.3.1. (Soundness) For all $\alpha \in Con$:

$$\alpha \sqsubseteq_A \beta \quad \text{implies} \quad [\![\alpha]\!] \leq [\![\beta]\!] \quad \text{and} \quad \alpha \sqsubseteq_M \beta$$

From Lemma 3.1.1 and the definition of the proof system we get:

$$
\begin{array}{rcl}
\langle (x := e); X, \sigma \rangle & = & \langle X, \sigma[v/x] \rangle \text{ where } v = [\![e]\!]\sigma \\
\langle be \rightarrow X_1, X_2, \sigma \rangle & = & \begin{cases} \langle X_1, \sigma \rangle & \text{if } [\![be]\!]\sigma = true \\ \langle X_2, \sigma \rangle & \text{if } [\![be]\!]\sigma = false \end{cases} \\
\langle op(\underline{X}), \sigma \rangle & = & op(\langle \underline{X}, \sigma \rangle) \\
\langle c?x.t, \sigma \rangle & = & c?x.\langle t, \sigma \rangle \\
\langle c!e.t, \sigma \rangle & = & c!v.\langle t, \sigma \rangle \text{ where } v = [\![e]\!]\sigma
\end{array}
$$

Fig. 8. Equations for Basic Configurations

Lemma 3.3.1. For all $\phi \in BCon$ there exists at most one α such that $\phi \rightarrowtail \alpha$. For this α, $\alpha =_A \phi$.

Proof. The first result is already stated in lemma 3.1.1. The second one follows easily from this and the equations in Figure 8 □

Now we define weak sum forms for configurations. As now the meaning of the configurations $\langle STOP, \sigma \rangle$ and $\langle \Omega, \sigma \rangle$ is completely independent of the store σ, both in the operational and the denotational semantics, we use the notation $STOP$ respectively Ω for these configurations.

Definition 3.3.2. (Weak Sum Forms) The set of weak sum forms, wSF, is the least set which satisfies

1. $STOP, \Omega \in wSF$
2. If $\phi_{ij} \in BCon$ and pre_{ij} are input/output prefixings, $i = 1, \ldots, i_j, j = 1, \ldots, N$, then

$$\sum_{j=1}^{N} \sum_{i-1}^{i_j} pre_{ij}.\phi_{ij} \in wSF.$$

We have the following:

Theorem 3.3.2. If δ is finite then $\delta =_A wsf(\delta)$ for some $wsf(\delta) \in wSF$.

Proof. Using the equation

$$\langle x := e.X.\sigma \rangle = \langle X, \sigma[[\![e]\!]\sigma/x] \rangle$$

we can remove assignment from the terms which appear in any finite configuration. Then we proceed in the same way as in the corresponding proof in [HI 89]. □

An easy consequence of this theorem is:

$$\text{I}\quad \alpha \sqsubseteq \alpha \qquad\qquad \frac{\alpha \sqsubseteq \beta,\ \beta \sqsubseteq \gamma}{\alpha \sqsubseteq \gamma}$$

$$\text{II}\quad \frac{\alpha_i \sqsubseteq \beta_i}{op(\underline{\alpha}) \sqsubseteq op(\underline{\beta})} \quad \text{for every } op \in \sum$$

$$\text{III}\quad \frac{\alpha \sqsubseteq \beta}{c!e.\alpha \sqsubseteq c!e.\beta} \qquad \frac{\alpha[v/x] \sqsubseteq \beta[v/x] \text{ for every } v \in V}{c?x.\alpha \sqsubseteq c?x.\beta}$$

$$\text{IV}\quad \frac{\alpha \sqsubseteq \beta}{\alpha\rho \sqsubseteq \beta\rho} \qquad \frac{}{\alpha \sqsubseteq \beta} \quad \text{for every equation } \alpha \sqsubseteq \beta$$

$$\text{V}\quad \frac{[\![e]\!] = [\![e']\!]}{c!e.\alpha = c!e'.\alpha}$$

$$\text{VI}\quad \frac{}{c?x.\alpha = c?y.\alpha[y/x]} \quad \text{if } y \text{ does not occur free in } \alpha$$

Fig. 9. Proof System

Corollary 3.3.1. For all finite δ

$$\delta \uparrow \ \Leftrightarrow\ \ \delta =_A \Omega.$$

Proof. By theorem 3.3.2 we may assume, that δ is in wSF. The result follows immediately from the structure of the weak sum forms and the strictness equations in Figure 2. □

Finally we introduce head normal forms for configurations. The definition is basically the same as the one given in [HI 89].

Definition 3.3.3. (Head Normal Forms) HNF is the least set which satisfies

1. $STOP \in HNF$
2. Let $\mathcal{A} \in sat(EV)$ and f a partial function, which associates with every $e \in \bigcup\mathcal{A}$ of the form $c!$ a finite nonempty set, $f(c)$, of pairs of values and basic configurations. Then any configuration of the form

$$\sum\!\!\!\!\!\!\bigcirc\ \{\sum\{\alpha_{e,f} | e \in A\} | A \in \mathcal{A}\}$$

is in HNF, where

a) if e is $c?$ then $\alpha_{e,f}$ is a configuration of the form $c?x.\langle t, \sigma\rangle$.

b) if e is $c!$ then $\alpha_{e,f}$ is the configuration

$$\sum\{c!v.\phi_v | (v,\phi_v) \in f(c)\}$$

We have the following normalization theorem for convergent configurations:

Theorem 3.3.3. (Normalization) If $\alpha \downarrow$ then $\alpha =_A h(\alpha)$ for some $h(\alpha) \in HNF$.

Proof. Similar to the proof for Proposition 4.2.1 in [HI 89]. □

For the sake of completeness we state the above mentioned properties as a proposition:

Proposition 3.3.1. For all $\alpha \in Con$ and $s \in Act^*$

i)$[\![\alpha]\!] \downarrow s$ if and only if $\alpha \downarrow s$
ii) if $\alpha \downarrow s$ then $\mathcal{A}([\![\alpha]\!], s) = c(\mathcal{A}(\alpha, s))$

Proof. Now when we have defined weak sum forms and head normal forms for configurations and proved the corresponding normalization theorems, the proof can proceed in exactly the same way as the corresponding proof in [HI 89] □

As motivated at the beginning of this section our main theorem, the full abstractness theorem, follows as a corollary to Proposition 3.3.1.

Theorem 3.3.4. (Full Abstractness) For all configurations α, β

$[\![\alpha]\!] \leq [\![\beta]\!]$ if and only if $\alpha \sqsubseteq_M \beta$

Proof. Follows immediately from Proposition 3.3.1 □

We will end this section by summarising the results and compare them to the results in the previous section.

In §2 we have a completeness result for the proof system with respect to closed terms whereas in this section we have proved full abstractness of the denotational model with respect to configurations. These two results are now combined in the following theorem:

Theorem 3.3.5. For all processes p, q
$p \sqsubseteq_A q$ if and only if $[\![p]\!] \leq [\![q]\!]$ if and only if $p \sqsubseteq_M q$

Proof. The first statement is just the content of Theorem 2.5.1. For the second one we recall that by definition $p \sqsubseteq_M q$ if and only if $\langle p, \sigma\rangle \sqsubseteq_M \langle q, \sigma\rangle$ for all σ. Theorem 3.3.4 implies that this is true if and only if $[\![\langle p, \sigma\rangle]\!] \leq [\![\langle q, \sigma\rangle]\!]$ for all σ. But as p and q are closed, this is equivalent to $[\![p]\!] \leq [\![q]\!]$. This completes the proof. □

4. Conclusion

In this paper we have represented a semantic theory for a process algebra which supports value-passing and is equipped with the imperative assignment construct. This is an direct extension of [HI 89], which handles an applicative version of the language, i.e. the same language but without assignment and stores.

As in the mentioned paper, the theory is approached from three different angles. Thus we define a denotational model for the language as a version of the model *Acceptance Trees*([He 88],[HI 89]). Further we give an axiomatization of the model and finally we define an operational behaviour in terms of testing. We show the equivalence of all three approaches.

Following the standard approach the semantic interpretation is given with respect to a store whereby we mean a total function which keeps track of the bindings of the variables to values. Our approach is based on the idea, which already occurs in [Hoa 78], that each subcomponent of a system has it own private store, only accessible for other processes by communication. Thus the possible side effects are localised and can only take place within the subcomponents of the system. This implies that a configuration, a pair of a term and a corresponding store, behave very much like processes in the applicative case with the consequence that we can reuse most of the theory already developed for this case ([HI 89]). Thus the denotational model is exactly the same and most of the equations are still valid; we only have to add a few very intuitive ones, dealing with assignment, to obtain a sound and complete proof system.

Our language can be considered as an extension of the programming language *Occam* which in turn can be considered as an descendant of the original programming language *CSP*. In one respect we are more limited than *Occam* in our language as we do not allow sequential composition in general but only as action prefixing but we believe that this does not affect the usefulness of the language too much.

Some theoretical work has been done on *CSP* and *Occam*. An operational semantics for *CSP* is given in [Plo 82], a denotational model for *Occam* in [Ros 87] and an axiomatic description of *Occam* in [HR 88]. Our theory is more complete in the sense that we give all three approaches and show that they all coincide. The operational semantics is very similar to that in [Plo 82]. As pointed out in the introduction our denotational semantics is quite different to that in [Ros 87] and moreover we do not need to introduce any restrictions on the use of channels and variables. Further our axiomatization is much simpler and more intuitive than in the previous case.

As mentioned earlier we restrict the sequential composition to a simple action prefixing. It could be the subject to future work to extend our language with the more general form for sequential composition and termination. The main problem we meet in this connection is how to handle the termination of a parallel composition of two processes.

Further, like in [HI 89], the proof system introduced in this paper is mainly of theoretical interest but could be extended along the lines of [He 89] to a more practical proof system.

Appendix: Operators in AT^v

Output:

Define $out_{AT} : Chan \times Val \times AT \longrightarrow AT$ by
$out_{AT}(c, v, t) = \langle \{\{c!\}\}, f \rangle$
where
$domain(f) = \{c!\}$
$f(c!) = \{\langle v, t \rangle\}$

Input:

Define $in_{AT} : Chan \times (Val \longrightarrow AT) \longrightarrow AT$ by
$in_{AT}(c, g) = \langle \{\{c?\}\}, f \rangle$
where
$domain(f) = \{c?\}$
$f(c?) = g$

STOP:

Let $STOP_{AT}$ be the tree $\langle \{\{\}\}, f \rangle$, where
f is the empty function

Ω :

Let Ω_{AT} be $\perp_{AT}$

Internal Nondeterminism:

Define $\oplus_{AT} : AT \times AT \longrightarrow AT$ as
$Y\lambda I.\lambda t.\lambda u.$ *if* $t = \perp$ *or* $u = \perp$
then $\perp$
else let
$\langle \mathcal{A}, f \rangle = t$
$\langle \mathcal{B}, g \rangle = u$
in $\langle c(\mathcal{A} \cup \mathcal{B}), h \rangle$

where $c(X)$ is the least saturated set containing X and h is defined by:

$h(c?) = I(f(c?), g(c?))$ if $c? \in domain(f) \cap domain(g)$
$= f(c?)$ if $c? \in domain(f) \setminus domain(g)$
$= g(c?)$ if $c? \in domain(g) \setminus domain(f)$

and

$h(c!) = f(c!)$ if $c! \in domain(f) \setminus domain(g)$
$= g(c!)$ if $c! \in domain(g) \setminus domain(f)$
$= k$ if $c! \in domain(f) \cap domain(g)$

where

$k(v) = I(f(c!)(v), g(c!)(v))$ if $v \in domain(f(c!)) \cap domain(g(c!))$
$= f(c!)(v)$ if $v \in domain(f(c!)) \setminus domain(g(c!))$
$= g(c!)(v)$ if $v \in domain(g(c!)) \setminus domain(f(c!))$

External Nondeterminism:

Define $+_{AT} : AT \times AT \longrightarrow AT$ by
$t +_{AT} u =$ *if* $t = \perp$ *or* $u = \perp$
then $\perp$

$$\begin{aligned}&\qquad else\ let\\&\qquad\qquad \langle \mathcal{A}, f\rangle = t\\&\qquad\qquad \langle \mathcal{B}, g\rangle = u\\&\qquad in\ \langle \mathcal{A}v\mathcal{B}, h\rangle\end{aligned}$$

where

$$\mathcal{A}v\mathcal{B} = \{A \cup B | A \in \mathcal{A}, B \in \mathcal{B}\}$$

and h is defined by

$$\begin{aligned}h(a) &= f(a) \text{ if } a \in domain(f) \setminus domain(g)\\ &= g(a) \text{ if } a \in domain(g) \setminus domain(f)\\ h(c?) &= f(c?) \oplus_{AT} g(c?) \text{ if } c? \in domain(f) \cap domain(g)\end{aligned}$$

and for $c! \in domain(f) \cap domain(g)$, $h(c!)$ is defined by

$$\begin{aligned}h(c!)(v) &= f(c!)(v) \oplus_{AT} g(c!)(v)\\ &\qquad \text{if } v \in domain(f(c!)) \cap domain(g(c!))\\ &= f(c!)(v) \text{ if } v \in domain(f(c!)) \setminus domain(g(c!))\\ &= g(c!)(v) \text{ if } v \in domain(g(c!)) \setminus domain(f(c!))\end{aligned}$$

Restriction:

For each $c \in Chan$ let $\setminus_{AT}\ c : AT \longrightarrow AT$ denote

$$\begin{aligned}&Y\lambda R.\lambda t.\ if\ t = \bot\ then\ \bot\\&\qquad else\ let\ \langle \mathcal{A}, f\rangle = t\\&\qquad\quad in\ \langle \mathcal{B}, g\rangle\end{aligned}$$

where

$$\mathcal{B} = \{A \setminus \{c?, c!\} \mid A \in \mathcal{A}\}$$

and

$$g(e) = R(f(e)) \text{ for } e \in Ev(\mathcal{B})$$

Parallel:

Define $|_{AT}\ : AT \times AT \longrightarrow AT$ as

$$\begin{aligned}&Y\lambda F.\lambda t.\lambda u.if\ t = \bot\ or\ u = \bot\\&\qquad then\ \bot\\&\qquad else\ let\\&\qquad\qquad \langle \mathcal{A}, f\rangle = t\\&\qquad\qquad \langle \mathcal{B}, g\rangle = u\\&\qquad\quad in\ \textstyle\sum \{t_{AB} | A \in \mathcal{A}, B \in \mathcal{B}\}\end{aligned}$$

where

$$\begin{aligned}t_{AB} = &if\ INT(A,B) = \emptyset\\ &then\ sumext(A,B)\\ &else\ (sumext(A,B) + sumint(A,B)) \oplus sumint(A,B)\end{aligned}$$

where

$$\begin{aligned}sumext(A,B) &= \textstyle\sum EXT(A,B)\\ sumint(A,B) &= \textstyle\sum INT(A,B)\end{aligned}$$

where $INT(A,B)$, $EXT(A,B)$ are defined by

$$\begin{aligned}INT(A,B) = &\{F(f(c?)(v), g(c!)(v)) \mid c? \in A, c! \in B\\ &\quad and\ v \in domain(g(c!))\}\\ &\cup\{F(f(c!)(v), g(c?)(v)) \mid c! \in A, c? \in B\\ &\quad and\ v \in domain(f(c!))\}\\ EXT(A,B) = &\{in_{AT}(c, \lambda v.F(f(c?)(v), u) \mid c? \in A\}\end{aligned}$$

$$\cup\{in_{AT}(c, \lambda v.F(t, g(c?)(v)) \mid c? \in B\}$$
$$\cup\{out_{AT}(c, v, F(f(c!)(v), u) \mid c! \in A,$$
$$\quad v \in domain(f(c!))\}$$
$$\cup\{out_{AT}(c, v, F(t, g(c!)(v)) \mid c! \in B,$$
$$\quad v \in domain(g(c!))\}$$

References

[BHR 84] Brookes, S.D., Hoare, C.A.R. and Roscoe, A.W. "A Theory of Communicating Sequential Processes", *JACM* 31(7), 560-599 1984.

[Bri 86] Brinksma, E. "A Tutorial on LOTOS." *Proceedings of IFIP Workshop on Protocol Specification, Testing and Verification V*, M. Diaz, ed., pp. 73-84. North-Holland, Amsterdam, 1986.

[DNH 84] DeNicola, R. and M. Hennessy. "Testing Equivalences for Processes." *Theoretical Computer Science*, 24, 1984, pp. 83-113.

[FLP 84] Francez, N., Lehman, D. and Pnueli, A. "A Linear History of Semantics for Languages with Distributed Processing, *TCS, 32, 25-46*, 1984.

[Gue 81] Guessarian, I., "Algebraic Semantics", *Springer-Verlag Lecture Notes in Computer Science*, vol.99, 1981.

[HP 80] Hennessy, M. and Plotkin, G., "A Term Model for CCS", *Springer-Verlag Lecture Notes in Computer Science*, vol.88, 1980.

[He 85] Hennessy, M. "Acceptance Trees." *Journal of the ACM*, v. 32, n. 4, October 1985, pp. 896-928.

[He 88] Hennessy, M. *Algebraic Theory of Processes.* MIT Press, Cambridge, 1988.

[HI 89] Hennessy, M., Ingólfsdóttir, A. A Theory of Communicating Processes With Value-Passing, University of Sussex Technical Report No 3/89, 1989. To appear in Information and Computation.

[He 89] Hennessy, M., A Proof System for Communicating Processes With Value - Passing, *Formal Aspects of Computing*, v. 3,1991, pp. 346-366. No 5/89, 1989.

[Hoa 78] Hoare, C.A.R. "Communicating Sequential Processes", *Comm. ACM*, 21(8), 666-677 1978.

[Hoa 85] Hoare, C.A.R. *Communicating Sequential Processes.* Prentice-Hall International, London, 1985.

[HR 88] Hoare, C.A.R. and Roscoe, A.W., "The Laws of Occam", TCS 60, pp 177-229, 1988.

[In 84] Inmos Ltd., *The Occam Programming Manual*, Prentice-Hall, London, 1984.

[Miln 88] Milne, R., "Concurrency Models and Axioms", RAISE/STC/REM/6/V2, STC Technology Ltd., 1988.

[Mil 80] Milner, R. *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92. Springer-Verlag, Berlin, 1980.

[Mil 89] Milner, R., *Calculus for Communication and Concurrency*, Prentice-Hall, London 1989.

[Ol 85] Olderog, E.R., Process Theory: Semantics, Specifications and Verifications, in J.W. deBakker, W.P. deRover, G. Rozenberg (Eds), Current Trends in Concurrency, Lecture Notes in Computer Science 224, Springer-Verlag, 1986, pp 442-509.

[Plo 81] Plotkin, G., "Lecture Notes in Domain Theory", University of Edinburgh, 1981.

[Plo 82] Plotkin, G., "An Operational Semantics for CSP", *Proc. of IFIP WG 2.2, Working Conference on Formal Description of Programming Concepts 11*, 1989.

[Ros 87] Roscoe, A.W., "Denotational Semantics for Occam", PRG Monograph, Oxford University, 1988.

[Smt 86] Schmidt, D., *Denotational Semantics*, Allen and Bacon, 1986.

[SP 82] Smyth, M. and Plotkin, G., "The Category-Theoretic Solution of Recursive Domain Equations", *SIAM Journal on Computing*, vol.11, No.4, 1982.