An Operational Framework for Value-Passing Processes*

Rance Cleaveland
Daniel Yankelevich
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
USA
tel: +1 919 515 7862
fax: +1 919 515 7896
e-mail: {rance, dany}@csc.ncsu.edu

September 13, 1994

Abstract

This paper develops a semantic framework for concurrent languages with value passing. An operation analogous to substitution in the λ-calculus is given, and a semantics is given for a value-passing version of Milner's Calculus of Communicating Systems (CCS). An operational equivalence is then defined and shown to coincide with Milner's (early) bisimulation equivalence. We also show how semantics may given for languages with asynchronous communication primitives. In contrast with existing approaches to value passing, this semantics does not reduce data exchange to pure synchronization over (potentially infinite) families of ports indexed by data, and it avoids variable renamings that are not local to processes engaged in communication.

1 Introduction

Process algebras such as CCS [22], CSP [18] and ACP [4] have been proposed as theories that highlight fundamental notions in concurrency in the same way that the λ-calculus [6] does

*Research supported by NSF/DARPA grant CCR-9014775, NSF grant CCR-9120995, ONR Young Investigator Award N00014-92-J-1582, and NSF Young Investigator Award CCR-9257963. Paper appeared in *21st ACM Symposium on Principles of Programming Languages*, pages 326–338, Portland, Oregon, January 1994.

for functional programming. Although the latter essentially includes only two programming constructs, λ-abstraction and function application, it has proven very useful as a framework for studying numerous issues, both implementational as well as theoretical, in functional programming languages. In the search for similarly foundational operations in concurrency, process algebra researchers have typically focused on modeling synchronization between, and the simultaneous execution of, processes. Appropriate equivalences and preorders [7, 10, 13, 15, 20, 21, 29] have been defined that relate concurrent systems on the basis of their externally visible behavior, and the theories have been successfully applied to the verification of concurrent systems. However, relatively little attention has been devoted to the operational aspects of the exchange of data between processes. This attention to "flow of values" is central in the λ-calculus, where notions of β-reduction and substitution form the basis for defining function application. When value passing has been modeled in process algebra, it has typically been done via a reduction to simple synchronization, with data values in essence encoded in the "name" of the ports that synchronizing processes use. While adequate at an abstract level, this approach obscures details associated with ensuring that the correct transmission of data from one process to another. Consequently, it is difficult to use these semantics to guide the development of implementations—specifically, interpreters—of value-passing languages.

The goal of this paper is to formulate an implementable semantics for process algebras with value passing. To this end, we present a semantics that accounts for operational concerns associated with the exchange of data between processes while maintaining a degree of abstraction from specific implementational details. The central difficulty to overcome is that, in contrast to the λ-calculus, where the syntactic structure of a reducible expression (redex) determines the subexpressions involved in a step of β-reduction, the emitters and consumers of values in concurrent programs may be arbitrarily separated, syntactically. Consequently, exchanging a value requires, at some level, the use of *routing information* to determine where the datum is to be delivered.

The remainder of the paper is structured as follows. The next section describes more precisely the intuitive motivation for the work presented in this paper. The section following then outlines a version of Milner's CCS with value-passing, defines its operational semantics according to the standard approach, and introduces an associated behavioral equivalence, bisimulation equivalence. Section 4 then remarks on a shortcoming of the traditional semantics, gives a semantics for value-passing designed to remedy this difficulty, and presents a new notion of semantic equivalence. The section following establishes that the semantic equivalences coincide, while Section 6 shows how the semantics may be altered to model asynchronous communication. The final section presents our conclusions and directions for future research.

2 Reduction and the Flow of Values

The aim of this paper is to give a semantics for concurrent processes that pass values; as we wish the semantics to be implementable, it should account for issues associated with the transmission of data, while remaining abstract with respect to more concrete implementational details. In this section we outline aspects peculiar to concurrent languages that complicate the development of such a semantics. We begin by first reviewing relevant intuitions from the λ-calculus, some familiarity with which will be assumed on the part of the reader.

The λ-calculus is built on the notion of β-reduction, which in turn is based on the operation of substitution. β-reduction can be thought of as specifying how an argument to a function is to be distributed inside the body of the function; in the reduction step

$$(\lambda x.M)N \rightarrow M[N/x],$$

the $\rightarrow$ relation in essence states that the expression N should be "sent" to all (free) occurrences of x in M. It should be noted that this "flow" of data is local to the redex $(\lambda x.M)N$ in the sense that the places in where datum N is to be "sent" can only be within M; any context that might surround the term cannot alter the possible destinations of N. This affords a syntactic characterization of redexes—a term can be a redex if and only if it is of the form $M\ N$ (i.e. M and N are syntactically adjacent) and M has the form $\lambda x.M'$. Different operational accounts of the λ-calculus exploit this definition of redex to define evaluation relations that have direct implementations as interpreters.

In concurrent languages, by way of contrast, the locality of data flow present in the λ-calculus is, by design, absent, since these languages are meant to model systems consisting of entities that exchange data without necessarily being adjacent to one another. For example, consider the following term in a putative process algebra.

$$\cdots \mid\mid \alpha X.P \mid\mid \cdots \mid\mid \overline{\alpha} E.Q \mid\mid \cdots$$

This system consists of some number of subsystems running in parallel. Process $\overline{\alpha}E.Q$ wishes to output expression E on port α and then behave like Q, while $\alpha X.P$ is awaiting an input on port α which it will then "assign" to X throughout P. What is notable here is the nonlocal flow of information; $\overline{\alpha}E.Q$ and $\alpha X.P$ in some sense form a "redex" even though they are not syntactically adjacent and may in fact be separated arbitrarily far apart in the term. Additionally, other processes (subterms) may also wish to communicate using α, and these may interfere with the exchange between the two processes above. Both of these features complicate the development of an "implementable" semantics for languages supporting concurrency and value passing; the work reported in this paper aims at circumventing these difficulties.

3 CCS and Value Passing

This section presents the syntax and traditional semantics for a language with value passing. In the interests of notational simplicity, values and processes will be distinct (in contrast with the language found in [27]).

The language—CCS with Value Passing (CCS-VP)—is parameterized with respect to several sets of syntactic entities. Let $\mathcal{V}$ be a (countably infinite) set of value variables ranged over by $V, V', V_1, \ldots$ and $\mathcal{E}$ be a set of value expressions ranged over by $E, E', E_1, \ldots$. Also let $\mathcal{P}r$ be a set of predicate expressions ranged over by $\pi, \pi', \ldots$. Elements of $\mathcal{E}$ and $\mathcal{P}r$ may contain occurrences of variables from $\mathcal{V}$. We assume $\mathcal{X}$ to be a (countably infinite) set of process variables, ranged over by $x, x', x_1, \ldots$, and Λ to be a set of *ports*, or labels, ranged over by $\alpha, \beta, \ldots$. We use $L, L', L_1, \ldots$ to represent finite subsets of Λ, and f to denote a (relabeling) function in $\Lambda \to \Lambda$. For technical convenience we assume that $\mathcal{V} \cap \mathcal{X} = \emptyset$. The syntax of CCS-VP may now be given using the following grammar.

$$\begin{array}{rcl} P & ::= & nil \mid \alpha V.P \mid \overline{\alpha} E.P \mid (\pi).P \mid P + P \\ & \mid & P|P \mid P\backslash L \mid P[f] \mid x \mid rec\, x.\ P \end{array}$$

In $\alpha V.P$ and $rec\, x.\ P$ the operators $\alpha E.$ and $rec\, x.$ bind free occurrences of V and x, respectively, in P in the usual sense. In the sequel we restrict our attention to terms in which all instances of process variables are *guarded*, i.e., occur inside a $\alpha V.$ or $\overline{\alpha} E.$ context. We use $\mathcal{P}$ to denote the set of all such guarded terms in CCS-VP, with $P, Q, P', Q', P_1, Q_1, \ldots$ ranging over $\mathcal{P}$. We sometimes refer to elements of $\mathcal{P}$ as *processes.*

In order to give the standard operational semantics of CCS-VP we first define the following notions. We call expressions of the form αE and $\overline{\alpha} E$, where $\alpha \in \Lambda$ and $E \in \mathcal{E}$, *actions.* We also let $\tau \notin \Lambda$ denote a special internal action and use

$$\begin{array}{rcl} \mathcal{A} & = & \{\, \alpha E \mid \alpha \in \Lambda \wedge E \in \mathcal{E} \,\} \\ & & \cup \{\, \overline{\alpha} E \mid \alpha \in \Lambda \wedge E \in \mathcal{E} \,\} \cup \{\tau\} \end{array}$$

to represent the set of all actions. $a, a', a_1, \ldots$ are used to range over $\mathcal{A}$. Intuitively, a process can perform αE when it is capable of receiving E as an input on α from its environment, while it can perform $\overline{\alpha} E$ when it may output E on α to its environment. We extend $\overline{}$ to actions as follows: if $a = \alpha E$ then $\overline{a} = \overline{\alpha} E$, and if $a = \overline{\alpha} E$ then $\overline{a} = \alpha E$. The function $\varpi \in \mathcal{A} \to \Lambda \cup \{\tau\}$ returns the "port component" of an action; so $\varpi(\alpha E) = \varpi(\overline{\alpha} E) = \alpha$, and $\varpi(\tau) = \tau$. Also, if $f \in \Lambda \to \Lambda$ and $a \in \mathcal{A}$, then $apply(f, a)$ relabels the port component of a by f; formally, this is given by

$$apply(f,a) = \begin{cases} f(\alpha)E & \text{if } a = \alpha E \\ \overline{f(\alpha)}E & \text{if } a = \overline{\alpha} E \\ \tau & \text{otherwise (i.e. } a = \tau) \end{cases}$$

We assume that $\mathcal{E}$ and $\mathcal{P}r$ are equipped with notions of free variable and substitution of expressions for variables; we use $fv(E) \subseteq \mathcal{V}$ to represent the free variables in E and $E_1[V := E_2] \in \mathcal{E}$ to denote the expression E_1 with all occurrences of V replaced by E_2, and similarly for $\pi \in \mathcal{P}r$. We furthermore assume that for any $E, |fv(E)| < \infty$. We also use $fv(P) \subseteq \mathcal{V}$ and $fv_{\mathcal{X}}(P)$ to represent the set of free value and process variables, respectively, in P; note that $fv(P)$ and $fv_{\mathcal{X}}$ are finite for every P. For $\pi \in \mathcal{P}r$, we assume the existence of a method $\models$ for evaluating π; so $\models \pi$ holds if π evaluates to *true*. If $\mathcal{X}$ is an infinite set, we use $\eta_{\mathcal{X}} \in 2^{\mathcal{X}} \to \mathcal{X}$ to denote a function with the property that for every finite $\mathcal{X}' \subseteq \mathcal{X}$, $\eta_{\mathcal{X}}(\mathcal{X}') \notin \mathcal{X}'$. We now define substitution for CCS-VP as follows.

Definition 3.1 *Let $W \in \mathcal{V} \cup \mathcal{X}$ and $R \in \mathcal{E} \cup \mathcal{P}$. Then $P[R/W]$ for $P \in \mathcal{P}$ is defined inductively as follows.*

- $nil[R/W] = nil$

- $x[R/W] = \begin{cases} R & \textit{if } x \equiv W, R \in \mathcal{P} \\ x & \textit{otherwise} \end{cases}$

- $(\alpha V.P)[R/W] =$

  $$\begin{cases} \alpha V.P & \textit{if } W \equiv V \\ \alpha V.(P[R/W]) & \textit{if } W \in \mathcal{X} \\ \alpha V'.(P[V'/V][R/W]) & \textit{otherwise} \end{cases}$$

  where $V' = \eta_{\mathcal{V}}(fv(R))$. Notice that the final clause handles variable capture in the usual fashion by renaming bound variables.

- $(\overline{\alpha} E'.P)[R/W] =$

  $$\begin{cases} \overline{\alpha}(E'[W := R]).(P[R/W]) & \textit{if } W \in \mathcal{V}, R \in \mathcal{E} \\ \overline{\alpha}E'.(P[R/W]) & \textit{otherwise} \end{cases}$$

- $((\pi).P)[R/W] =$

  $$\begin{cases} (\pi[W := R]).(P[R/W]) & \textit{if } W \in \mathcal{V}, R \in \mathcal{E} \\ (\pi).(P[R/W]) & \textit{otherwise} \end{cases}$$

- $(P_1 + P_2)[R/W] = (P_1[R/W]) + (P_2[R/W])$

- $(P_1|P_2)[R/W] = (P_1[R/W]) \mid (P_2[R/W])$

- $(P\backslash L)[R/W] = (P[R/W])\backslash L$

- $(P[f])[R/W] = (P[R/W])[f]$

- $(rec\, x.\ P)[R/W] =$

  $$\begin{cases} rec\, x.\ P & \textit{if } W \equiv x \\ rec\, x.\ (P[R/W]) & \textit{if } W \in \mathcal{V} \\ rec\, x'.\ (P[x'/x][R/W]) & \textit{otherwise} \end{cases}$$

  where $x' = \eta_{\mathcal{X}}(fv_{\mathcal{X}}(P))$. The final clause prevents variable capture.

The traditional semantics of CCS-VP is given in terms of a relation $\rightarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$, where $P \xrightarrow{a} P'$ holds if P is capable of executing action a and evolving to P' as a result. The formal definition of $\rightarrow$ appears in Figure 1. Intuitively, $\alpha V.P$ is a process waiting for an input on channel α; upon receiving an expression it evolves to P with occurrences of V replaced by this value. $\overline{\alpha}E.P$ first outputs E on α and then behaves like P. A process $(\pi).P$ may execute an action from P only if the predicate π is true. Operator $+$ is a choice operator; $P_1 + P_2$ is initially capable of the actions of either P_1 or P_2, with the process thereafter behaving like

$$
\begin{array}{llll}
\alpha V.P \xrightarrow{\alpha E} P[E/V] & \text{for any } E \in \mathcal{E} & & \text{(prefin)} \\
\overline{\alpha} E.P \xrightarrow{\overline{\alpha} E} P & & & \text{(prefout)} \\
P \xrightarrow{a} P', \models \pi & \Rightarrow & (\pi).P \xrightarrow{a} P' & \text{(cond)} \\
P \xrightarrow{a} P' & \Rightarrow & P + Q \xrightarrow{a} P' & \text{(suml)} \\
Q \xrightarrow{a} Q' & \Rightarrow & P + Q \xrightarrow{a} Q' & \text{(sumr)} \\
P \xrightarrow{a} P' & \Rightarrow & P|Q \xrightarrow{a} P'|Q & \text{(parl)} \\
Q \xrightarrow{a} Q' & \Rightarrow & P|Q \xrightarrow{a} P|Q' & \text{(parr)} \\
P \xrightarrow{a} P', Q \xrightarrow{\overline{a}} Q' & \Rightarrow & P|Q \xrightarrow{\tau} P'|Q' & \text{(comm)} \\
P \xrightarrow{a} P', \pi(a) \notin L & \Rightarrow & P \backslash L \xrightarrow{a} P' \backslash L & \text{(res)} \\
P \xrightarrow{a} P' & \Rightarrow & P[f] \xrightarrow{apply(f,a)} P'[f] & \text{(rel)} \\
P[rec\,x.\ P/x] \xrightarrow{a} P' & \Rightarrow & rec\,x.\ P \xrightarrow{a} P' & \text{(rec)}
\end{array}
$$

Figure 1: The Traditional Semantics of CCS-VP.

the process whose initial action was chosen. Process $P_1|P_2$ models the parallel composition of P_1 and P_2 by interleaving the actions available to each; in addition, if one of the processes is capable of outputting a value on a port that the other is capable of receiving on, then the two processes synchronize, with an internal action resulting. In $P \backslash L$ the ports in L become *local*; no interaction can take place between P and its environment using these channels. In $P[f]$ f is used to rename the labels that P interacts on. Finally, $rec\,x.\ P$ represents a recursive process that can be thought of as a "solution" to the equation $x = P$.

Having defined the operational semantics of CCS-VP processes, we now present the definition of *bisimulation equivalence* [22], which is designed to relate processes on the basis of the behavior they exhibit to their environments. The definition is given in terms of bisimulations, which are relations that require related processes to have "matching transitions".

Definition 3.2 *A relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a bisimulation if, whenever $P\ \mathcal{R}\ Q$, then the following holds for all $a \in \mathcal{A}$.*

1. *If $P \xrightarrow{a} P'$ then there is a Q' such that $Q \xrightarrow{a} Q'$ and $P'\ \mathcal{R} Q'$.*
2. *If $Q \xrightarrow{a} Q'$ then there is a P' such that $P \xrightarrow{a} P'$ and $P'\ \mathcal{R} Q'$.*

Bisimulation equivalence, $\sim \subseteq \mathcal{P} \times \mathcal{P}$, is defined by: $P \sim Q$ if and only if there is a bisimulation $\mathcal{R}$ such that $P\ \mathcal{R}\ Q$.

4 An Effective Semantics for Value Passing

While elegant, the semantics of CCS-VP is not directly implementable, in a certain sense. To see this, consider how one might try to implement an interpreter for CCS-VP on the basis of the rules in Figure 1. Given a process term P, such an evaluator would need first to compute the set

$$\{\, \langle a, P' \rangle \mid P \xrightarrow{a} P' \,\}.$$

However, this set will in general be infinite when P is capable of any input action and the set of value expressions is infinite, and thus the interpreter would not, in general, terminate.

The problem with the semantics is that it is too abstract with respect to modeling the exchange of values. In essence, the semantics works by replacing a given input port name by a number of port names indexed by the possible values that may be received on it. For example, suppose that $\mathcal{E} = Nat$, the set of natural numbers, and consider a process of the form $P \equiv \alpha V.P'$. From the operational semantics given in the previous section it is the case that $P \xrightarrow{\alpha i} P'[i/V]$ for any natural number i. The same behavior could be deduced for the following process in a non-value-passing calculus: $\Sigma_{i \in Nat} \alpha_i.P'[i/V]$. This has the effect of obscuring how the transfer of values between P and $Q \equiv \overline{\alpha}E.Q'$ should be achieved in the system $P|Q$.

In this section, we develop an alternative operational semantics for CCS-VP that is computable; it also aims to expose what we feel is a main operational consideration in the implementation of communicating concurrent systems: the routing of data from senders to receivers. The new semantics relies on the introduction of *process addresses* and an operation of *routed substitution* that "channels" a substitution to the process specified by a given address. Intuitively, input actions will now be tagged with the address of the process expecting the value; when a synchronization occurs, the address borne by the input action will guide the resulting substitution to the appropriate destination.

Formally we define the set $\mathcal{S}$ of process addresses using the following grammar.

$$s ::= \bullet \;\mid\; s\rceil \;\mid\; s\lceil$$

We use $s, s', s_1, \ldots$ to range over $\mathcal{S}$. Intuitively, a process address describes the position of a process within a larger system. An address of the form $\bullet$ indicates the current process, while $\rceil$ and $\lceil$ indicate in which component—left or right, respectively—of a parallel composition $P|Q$ the process is located. For example, the address of Q in $(P|Q)|R$ is $\bullet\lceil\rceil$. This syntax has alrady been used in the literature to describe addresses of subprocesses; see for instance [24].

In order to specify the new semantics, we first introduce the new set of actions we consider. The set $\mathcal{A}_{in} = \{\, \alpha V@s \mid \alpha \in \Lambda \wedge V \in \mathcal{V} \wedge s \in \mathcal{S} \,\}$ represents the set of input actions. $\mathcal{A}_o = \{\, \overline{\alpha}E \mid \alpha \in \Lambda \wedge E \in \mathcal{E} \,\} \cup \{\tau\}$ contains the "noninput actions"; then $\mathcal{A}' = \mathcal{A}_{in} \cup \mathcal{A}_o$. We use $a, a', a_1, \ldots$ to range over $\mathcal{A}'$, and $\breve{a}$ to range over $\mathcal{A}_o$. The functions $\varpi' \in \mathcal{A}' \to \Lambda \cup \{\tau\}$ and $apply' \in (\Lambda \to \Lambda) \times \mathcal{A}' \to \mathcal{A}'$ are now defined by:

$$\varpi'(a) = \begin{cases} \alpha & \text{if } a = \alpha V@s \\ \alpha & \text{if } a = \overline{\alpha}E \\ \tau & \text{otherwise} \end{cases}$$

$$apply'(f,a) = \begin{cases} f(\alpha)V@s & \text{if } a = \alpha V@s \\ \overline{f(\alpha)}E & \text{if } a = \overline{\alpha}E \\ \tau & \text{otherwise (i.e. } a = \tau) \end{cases}$$

We now introduce the notion of routed substitution. Intuitively, $[E/V]@s$ indicates that the substitution $[E/V]$ should be performed on the process at address s. For instance, applying $[E/V]@\bullet\lceil\rceil$ to the process $(\bar{\beta}(V+3).nil \,|\, \bar{\alpha}(V*2).nil) \,|\, \alpha.nil$ sends the substitution to the subterm $\bar{\alpha}(V*2).nil$, returning $(\bar{\beta}(V+3).nil \,|\, \bar{\alpha}(E*2).nil) \,|\, \alpha.nil$. The formal definition is as follows.

Definition 4.1 *Let $P, Q \in \mathcal{P}$, $E \in \mathcal{E}$, $V \in \mathcal{V}$, and $s \in \mathcal{S}$. Then $P[E/V]@s$ is defined inductively as follows.*

- *If $s = \bullet$ then $P[E/V]@s = P[E/V]$.*
- *If $s \neq \bullet$ then $P[E/V]@s$ is defined as follows.*
  - $(P|Q)[E/V]@s =$
    $\begin{cases} (P[E/V]@s')|Q & \textit{if } s = s'\rceil \\ P|(Q[E/V]@s') & \textit{if } s = s'\lceil \end{cases}$
  - $(P\backslash L)[E/V]@s = (P[E/V]@s)\backslash L$.
  - $(P[f])[E/V]@s = (P[E/V]@s)[f]$.
  - $nil[E/V]@s = nil$.
  - $(a.P)[E/V]@s = a.P$.
  - $((\pi).P)[E/V]@s = (\pi).P$.
  - $(P+Q)[E/V]@s = P+Q$.
  - $(rec\, x.\ P)[E/V]@s = rec\, x.\ P$.

Note that when $s \neq \bullet$ and P has the form nil, $a.P'$, $P_1 + P_2$ or $rec\, x.\ P'$, the rules for $P[E/V]@s$ indicate that the substitution is dropped. Intuitively, in this situation s represents a "faulty" address; it is fairly straightforward to show that in the rules given in Figure 2, these cases can never arise.

The operational semantics is given in terms of a relation $\longmapsto \subseteq (\mathcal{P} \times \mathcal{A}' \times \mathcal{P})$, the formal definition for which appears in Figure 2. Most of the rules from the old semantics remain the same; only the treatment of prefixing with an input action and $|$ need changing.

The intuition underlying the rules is the following. In this semantics, input actions are labeled with the address of the process to which an incoming value should be directed. The operations $+, \backslash L, [f]$ and $rec\, x.$ behave essentially as they do in the old semantics, as does prefixing with an output action or a predicate. The rule for input-action prefixing assigns the address $\bullet$ to the action labeling the transition. The rules for $|$ are slightly more complicated than before, since actions must be decorated with appropriate routing information. Thus, if P requests an input on αV at address s and evolves to P', then $P|Q$ must add a $\rceil$ marker to

Rules (prefout), (cond), (suml), (sumr), (res), (rel), and (rec) from Figure 1, with $\longmapsto$ replacing $\rightarrow$.

$$
\begin{array}{llll}
\alpha V.P \overset{\alpha V @ \bullet}{\longmapsto} P & & & \text{(prefin')} \\
P \overset{\alpha V @ s}{\longmapsto} P' & \Rightarrow & P|Q \overset{\alpha V @ s \rceil}{\longmapsto} P'|Q & \text{(parli')} \\
P \overset{\breve{a}}{\longmapsto} P' & \Rightarrow & P|Q \overset{\breve{a}}{\longmapsto} P'|Q & \text{(parlo')} \\
Q \overset{\alpha V @ s}{\longmapsto} Q' & \Rightarrow & P|Q \overset{\alpha V @ s \lceil}{\longmapsto} P|Q' & \text{(parri')} \\
Q \overset{\breve{a}}{\longmapsto} Q' & \Rightarrow & P|Q \overset{\breve{a}}{\longmapsto} P|Q' & \text{(parro')} \\
P \overset{\alpha V @ s}{\longmapsto} P', Q \overset{\overline{\alpha} E}{\longmapsto} Q' & \Rightarrow & P|Q \overset{\tau}{\longmapsto} (P'[E/V]@s)|Q' & \text{(comml')} \\
P \overset{\overline{\alpha} E}{\longmapsto} P', Q \overset{\alpha V @ s}{\longmapsto} Q' & \Rightarrow & P|Q \overset{\tau}{\longmapsto} P'|(Q'[E/V]@s) & \text{(commr')}
\end{array}
$$

Figure 2: A New Semantics for Value Passing.

s in evolving to $P'|Q$ to indicate that the substitution should be sent to the left component of the $|$. If P wishes to perform an output without synchronizing with P, then $P|Q$ behaves as in the old semantics, and this is reflected in the form of rules (parlo') and (parro'). Finally, if P signals a desire for an input on αV and Q is willing to perform an output on the same port, then the two may synchronize, with a τ resulting and the appropriate substitution being routed to P. The dual situation holds for Q. The rules implementing this are (comml') and (commr').

As an illustration of semantics, consider the following term.

$$P \equiv (\alpha X.\overline{\beta} X.nil \,|\, \beta Y.nil)|\overline{\alpha} E.nil$$

By applying the above rules, we may deduce the following transitions for P.

$$
\begin{array}{lll}
P & \overset{\alpha X @ \bullet \rceil\rceil}{\longmapsto} & (\overline{\beta} X.nil \,|\, \beta Y.nil)|\overline{\alpha} E.nil \\
P & \overset{\beta Y @ \bullet \lceil\rceil}{\longmapsto} & (\alpha X.\overline{\beta} X.nil | nil)|\overline{\alpha} E.nil \\
P & \overset{\overline{\alpha} E}{\longmapsto} & (\alpha X.\overline{\beta} X.nil \,|\, \beta Y.nil)|nil \\
P & \overset{\tau}{\longmapsto} & (\overline{\beta} X.nil \,|\, \beta Y.nil)[E/X]@\bullet\rceil \mid nil \\
 & & \equiv ((\overline{\beta} X.nil)[E/X]@\bullet \mid \beta Y.nil) \mid nil \\
 & & \equiv (\overline{\beta} E.nil \,|\, \beta Y.nil)|nil
\end{array}
$$

The last of these transitions illustrates how the $\lceil$ and $\rceil$ operators guide incoming substitutions to the appropriate subterm. Notice also that in the transfer of data in the last transition, the routing procedure does not "touch" the process $\beta Y.nil$; no attempt is made to substitute E into it.

4.1 Implementing the Semantics

That the semantics is implementable stems from the following result.

Theorem 4.2 *Let $P \in \mathcal{P}$, and let $\mathcal{T} = \{ \langle a, K \rangle \mid P \xmapsto{a} K \}$. Then $|\mathcal{T}| < \infty$.*

On the basis of this theorem, it is relatively straightforward to implement a routine that computes the transitions available to a term. Given a term as input, the procedure would recursively compute the transitions of its subterms and combine them appropriately, as indicated by the applicable rules, to obtain the transitions of the over-all term. As the theorem guarantees that these sets are finite, the routine is effectively computable.

4.2 A Semantic Equivalence

In $P \xmapsto{\alpha V} Q$, Q may contain free occurrences of V; therefore we cannot directly apply the definition of $\sim$ given in Section 3. Instead, we define the following.

Definition 4.3 *A relation $\mathcal{R} \subseteq (\mathcal{P} \times \mathcal{P})$ is an input-sensitive bisimulation if, whenever $P \mathcal{R} Q$ then the following hold.*

1. *If $P \xmapsto{\breve{a}} P'$, then there is a Q' such that $Q \xmapsto{\breve{a}} Q'$ and $P' \mathcal{R} Q'$, and symmetrically for Q.*
2. *If $P \xmapsto{\alpha V @ s} P'$ then $\forall E \in \mathcal{E}$ there are $Q' \in \mathcal{P}, V' \in \mathcal{V}$ and $s' \in \mathcal{S}$ such that $Q \xmapsto{\alpha V' @ s'} Q'$ and $P'[E/V]@s \; \mathcal{R} \; Q'[E/V']@s'$, and symmetrically for Q.*

$P \simeq Q$ if there is an input-sensitive bisimulation $\mathcal{R}$ such that $P \mathcal{R} Q$.

Notice the parallel between this equivalence and the traditional extensional equivalence on functions that relates two functions iff they give the same results for all possible arguments. It is straightforward to establish that $\simeq$ is itself an input-sensitive bisimulation, and is in fact the largest such relation.

5 The Equivalences Are Equivalent

This section establishes that $\simeq$ relates exactly the same CCS-VP processes as $\sim$. The importance of this result lies in the fact that it licenses the use of either form of operational semantics for CCS-VP. If one is not interested in the specific mechanisms of data exchange, then one may use the traditional, more abstract semantics; if on the other hand one is interested in more implementational considerations, then one may use the new semantics. In either case, the equivalences deduced between processes at one level will hold at the other.

The correspondence between equivalences rests on the following lemmas.

Lemma 5.1 *Let $P \in \mathcal{P}, \alpha \in \Lambda$, and $E \in \mathcal{E}$. Then $P \xrightarrow{\alpha E} P'$ if and only if there exist $Q \in \mathcal{P}, V \in \mathcal{V}$ and $s \in \mathcal{S}$ such that $P \xmapsto{\alpha V @ s} Q$ and $P' \equiv Q[E/V]@s$.*

Proof. Proceeds by induction on the derivations of $P \xrightarrow{\alpha E} P'$ and $P \xmapsto{\alpha V @ s} Q$. $\square$

Lemma 5.2 *Let $P \in Proc$ and $a \in \mathcal{A}_o$. Then $P \xrightarrow{a} P'$ if and only if $P \stackrel{a}{\longmapsto} P'$.*

Proof. Proceeds by induction on the derivations of $P \xrightarrow{a} P'$ and $P \stackrel{a}{\longmapsto} P'$. $\square$
Using these lemmas, we can now prove the following theorem.

Theorem 5.3 *Let $P, Q \in \mathcal{P}$. Then $P \sim Q$ if and only if $P \simeq Q$.*

Proof. Using Lemmas 5.1 and 5.2, one can show that every input-respecting bisimulation is also a bisimulation and that every bisimulation may be extended into an input-respecting bisimulation. The proof then follows. $\square$

Early vs. Late Equivalence

The traditional equivalence presented in this paper is sometimes referred to as *early* bisimulation equivalence. The intuitive explanation for this terminology is that when trying to match an input transition of process P, Q gets to "look ahead" at incoming values in order to determine which of its transitions to use. More formally, consider the form of the second condition in the definition of input-enabled bisimulation.

$$\begin{array}{l}\forall E \in \mathcal{E}\ \exists Q' \in \mathcal{P}, V' \in \mathcal{V}, \text{ and } s' \in \mathcal{S} \text{ such that} \\ Q \stackrel{\alpha V' @ s'}{\longmapsto} Q' \text{ and } P'[E/V]@_s \, \mathcal{R} \, Q'[E/V']@_{s'}\end{array}$$

The universal quantification over values is done before the existential quantification over transitions. Hence, this condition has the form $\forall E\, \exists P.\, C$.

If one alters the definition of equivalence by reversing the univeral and existential quantifications above, then one obtains *late* bisimulation equivalence.

Definition 5.4 *A relation $\mathcal{R} \subseteq (\mathcal{P} \times \mathcal{P})$ is a late input-sensitive bisimulation if, whenever $P \, \mathcal{R} \, Q$ then the following hold.*

1. *If $P \stackrel{\breve{a}}{\longmapsto} P'$, then there is a Q' such that $Q \stackrel{\breve{a}}{\longmapsto} Q'$ and $P' \, \mathcal{R} \, Q'$, and symmetrically for Q.*

2. *If $P \stackrel{\alpha V @ s}{\longmapsto} P'$ then there are $Q' \in \mathcal{P}, V' \in \mathcal{V}$ and $s' \in \mathcal{S}$ such that $Q \stackrel{\alpha V' @ s'}{\longmapsto} Q'$ and $\forall E \in \mathcal{E}\ \ P'[E/V]@_s \, \mathcal{R} \, Q'[E/V']@_{s'}$, and symmetrically for Q.*

$P \simeq_l Q$ if there is a late input-sensitive bisimulation $\mathcal{R}$ such that $P \mathcal{R} Q$.

This equivalence has usually been presented in the literature by giving different the operational rules together with the definition of bisimulation [17, 19]. In our setting, the definition is obtained solely by swapping quantifiers, as is also done in [23] for the Π-calculus. One immediate advantage of this formulation is that the fact that late equivalence implies early equivalence is a consequence of a well-known property of quantifiers in first order logic.

Lemma 5.5 *Let $P, Q \in \mathcal{P}$. Then, $P \simeq_l Q$ implies that $P \simeq Q$.*

Proof. The lemma is implied by the property $\exists P \forall E. C \;\Rightarrow\; \forall E \exists P. C$. $\Box$

The following example shows that in general, early equivalence does *not* imply late equivalence. Assume that $\mathcal{E}$ includes the set of integers as well as the operations $|n|$ (absolute value) and $n!$ (factorial).

Example 5.6 *Assume P and Q are defined as follows.*

$$\begin{array}{rcl} P & = & \alpha V.((V \geq 0).\bar{\beta}(V!).nil + (V < 0).\bar{\gamma}V.nil) + \\ & & \alpha V.((V < 0).\bar{\beta}((-V)!).nil + (V \geq 0).\bar{\gamma}V.nil) \\ Q & = & \alpha V.\bar{\beta}(|V|!).nil + \alpha V.\bar{\gamma}V.nil \end{array}$$

Then $P \simeq Q$ but $P \not\simeq_l Q$.

6 A Semantics for Asynchronous Communication

The semantics given above requires that processes wishing to communicate must do so *synchronously*—senders must block until a receiver is ready. Such a communication scheme is standard in process algebras, as one can model other disciplines using synchronous communication. However, other programming notations provide asynchronous communication primitives in which senders need not block; examples include Linda [14], Linear Objects [2], Alice [28], Gamma [5], concurrent logic languages [26], and concurrent constraint languages [25]. Moreover, other languages, such as CRP [9], provide synchronous primitives but implement them over an asynchronous model; reasoning about these implementations would require a semantics with asynchrony represented explicitly. In this section we show how the semantic framework developed above may be altered to cater for different forms of asynchronous communication, and we study the incorporation of *explicit routed substitutions* in our language as a means of accounting for communication costs.

6.1 Asynchronous CCS with Values

Other semantic approaches to asynchronous message passing typically include buffers for storing messages that have been sent but not received. When a sender wishes to send, it deposits its message into the buffer and continues with its execution. A receiver first examines the buffer, retrieving a value if one exists and blocking otherwise.

In order to accommodate asynchronous communication in CCS-VP, we add a *buffering* operator $\rhd$; intuitively, $\overline{\alpha}E \rhd P$ represents a process that has deposited a message containing value E and destination port α in its "output buffer". Formally, the syntax of ACCS-VP is given as follows.

$$\begin{array}{rcl} P & ::= & nil \mid \alpha V.P \mid \overline{\alpha}E.P \mid P + P \mid P|P \\ & \mid & P\backslash L \mid P[f] \mid x \mid rec\, x.\, P \mid \overline{\alpha}E \rhd P \end{array}$$

As before, we assume that all recursive definitions are guarded. We use $\mathcal{P}_A$, ranged over by $P, Q, \ldots$, to denote the set of syntactically correct terms.

The rules from Figure 2, except (prefout).

$$\overline{\alpha}E.P \stackrel{\tau}{\longmapsto} \overline{\alpha}E \rhd P \qquad \text{(prefout')}$$

$$\overline{\alpha}E \rhd P \stackrel{\overline{\alpha}E}{\longmapsto} P \qquad \text{(buff1)}$$

$$P \stackrel{a}{\longmapsto} P' \quad \Rightarrow \quad \overline{\alpha}E \rhd P \stackrel{a}{\longmapsto} \overline{\alpha}E \rhd P' \qquad \text{(buff2)}$$

Figure 3: Transition Rules for Asynchrony

To define the operational semantics of ACCS-VP we first extend the definitions of substitution and routed substitution in the obvious fashion as follows.

- $(\overline{\alpha}E' \rhd P)[R/W] = \overline{\alpha}E' \rhd (P[E/V])$
- $(\overline{\alpha}E \rhd P)[E/V]@_s = \overline{\alpha}E \rhd (P[E/V]@_s)$

Our asynchronous operational semantics is given in terms of the relation $\longmapsto$ introduced in Figure 3. Most of the operations inherited from CCS-VP behave as they did before, and hence the rules do not change. The core of the new semantics lies in (prefout'); this rule informally says that an output prefixing $\overline{\alpha}E.P$ does not communicate directly with other processes, but instead buffers its message. Notice that inserting a message in the buffer is seen as an internal, unobservable operation, and thus the transition is labeled by a τ. The rules for $\overline{\alpha}E \rhd P$ allow this process either to execute its output action (Rule (buff1)) or perform an action available to P (Rule (buff2)). As an example, consider the following term.

$$P \equiv (\overline{\alpha}E_1.\overline{\beta}E_2.nil\,|\alpha V.nil)\backslash\{\alpha\}$$

By applying the rules, we deduce the following sequences of transitions for P.

$$\begin{array}{lll}
P & \stackrel{\tau}{\longmapsto} & (\overline{\alpha}E_1 \rhd \overline{\beta}E_2.nil\,|\alpha V.nil)\backslash\{\alpha\} \\
 & \stackrel{\tau}{\longmapsto} & (\overline{\alpha}E_1 \rhd \overline{\beta}E_2 \rhd nil\,|\alpha V.nil)\backslash\{\alpha\} \\
 & \stackrel{\overline{\beta}E_2}{\longmapsto} & (\overline{\alpha}E_1 \rhd nil\,|\alpha V.nil)\backslash\{\alpha\} \\
 & \stackrel{\tau}{\longmapsto} & (nil\,|\,nil)\backslash\{\alpha\}
\end{array}$$

$$\begin{array}{lll}
P & \stackrel{\tau}{\longmapsto} & (\overline{\alpha}E_1 \rhd \overline{\beta}E_2.nil\,|\alpha V.nil)\backslash\{\alpha\} \\
 & \stackrel{\tau}{\longmapsto} & (\overline{\beta}E_2.nil\,|\,nil)\backslash\{\alpha\} \\
 & \stackrel{\tau}{\longmapsto} & (\overline{\beta}E_2 \rhd nil\,|\,nil)\backslash\{\alpha\} \\
 & \stackrel{\overline{\beta}E_2}{\longmapsto} & (nil\,|\,nil)\backslash\{\alpha\}
\end{array}$$

Our buffering operator should be compared with the *state* operator of [12]. While both constructs are used to give a semantics to asynchronous communication, they differ in that the state operator has as an argument a data structure that contains the values that have been sent but not yet received. By varying the form of this data structure, one obtains different queuing disciplines for message delivery. As will be shown in the next section, we may obtain some of the same flexibility, with lower syntactic overhead, by varying the operational rules governing $\rhd$.

6.2 Buffering Policies

Schemes for asynchronous communication typically differ in terms of their *delivery scheme* (i.e. is communication point-to-point or broadcast-based?) and *buffering policy* (i.e. are messages delivered in FIFO order?). While the former issue is beyond the scope of this paper, this section shows how the operational semantics given in the previous section may be altered to model different approaches to buffering.

The rules for $\longmapsto$ given in Figure 3 in essence define a communication framework in which delivery is point-to-point (Rules (comml') and (commr')) and the buffering policy defines the communication medium to be a *bag*: each process has an unordered, unbounded buffer of messages. This behavior follows from Rule (buff2): using it, we may conclude, for instance, that process $\overline{\alpha}E \rhd \overline{\beta}E' \rhd P$ is capable of engaging in transitions labeled by $\overline{\beta}E'$ as well as by $\overline{\alpha}E$. This communication mechanism is used in one form or another in many coordination or programming languages (see for instance Linda, Alice, Linear Objects and Gamma). Other notations (e.g. Estelle [11], UNIX sockets), however, provide FIFO or Quasi-FIFO (i.e. buffers that are FIFO for messages on each channel) communication schemes, and to model these in an implementation-independent manner we would like to have the policy be "built" into the operational semantics. It turns out that our semantics can be changed to support Quasi-FIFO or FIFO communication by adding different side conditions to Rule (buff2).

Definition 6.1 *The rules for Quasi-FIFO ACCS-VP are obtained by replacing the rule (buff2) by the rule*

$$P \stackrel{a}{\longmapsto} P',\, a \neq \overline{\alpha}E' \quad \Rightarrow \quad \overline{\alpha}E \rhd P \stackrel{a}{\longmapsto} \overline{\alpha}E \rhd P'$$

This alteration to the semantics ensures that in process $\overline{\alpha}E \rhd \overline{\alpha}E' \rhd P$, the action $\overline{\alpha}E$ must be executed before the $\overline{\alpha}E'$ is possible.

Definition 6.2 *The rules for FIFO ACCS-VP are obtained by replacing the rule (buff2) by the rule*

$$\begin{array}{c} P \stackrel{a}{\longmapsto} P',\ \forall \beta \in \Lambda.\, a \neq \overline{\beta}E' \\ \Rightarrow\ \overline{\alpha}E \rhd P \stackrel{a}{\longmapsto} \overline{\alpha}E \rhd P' \end{array}$$

This change stipulates that in process $\overline{\alpha}E \rhd \overline{\beta}E' \rhd P$, action $\overline{\alpha}E$ occurs before $\overline{\beta}E'$ (but a $\gamma V@s$ action of P can be arbitrarily interleaved with them).

6.3 Explicit Routed Substitutions

In our semantics, routed substitution has played a role analogous to the one played by substitution in the λ-calculus. In computational terms, this operation, like substitution in the λ-calculus, can be expensive to implement, and leaving it as a single operation in a semantics precludes the study of the relative efficiency of different system implementations. A small example serves to illustrate this point. Consider a system of the form $P|(Q|R)$; if we interpret $|$ as a "communication device" for linking together two systems, then we may think of this system as a tree-structured network consisting a five processors: P, Q, R, a "communication processor" connecting Q and R, and another communication processor linking this processor to P. Under this informal interpretation, communication among P, Q and R must flow through the relevant communication processors. Now suppose we have that $P \stackrel{\overline{\alpha}E}{\longmapsto} P'$ while $Q \stackrel{\alpha V @ \bullet}{\longmapsto} Q'$; we want to consider how E will be delivered to Q'. In general, two "hops" would be required: one in which the "root" communication processor channels E from P to the communication processor connecting Q and R, and one in which the second communication processor directs E to Q. On the other hand, $(P|Q)|R$ requires only one such hop to effect the transfer of the same value from P to Q. If this communication happens frequently, then, we might, for reasons of efficiency, prefer the second system configuration to the first, since in a certain sense P and Q are "closer" in the second system than in the first. However, our current semantic account for ACCS-VP does not permit us to distinguish these systems, as (with respect to the equivalence $\simeq$) $|$ may be shown to be associative.

In the case of substitution and the λ-calculus, this shortcoming of the traditional β-reduction semantics has been addressed by including *explicit substitutions* [1] in the syntax of terms. Explicit substitutions may be thought of as syntactic renderings of closures; the operational semantics of a language with explicit substitutions in essence encodes reduction rules corresponding to the different cases in the inductive definition of substitution. Thus, on the basis of the length of a reduction sequence one can obtain information about the cost of reducing one term to another. In this section, we consider a similar modification to ACCS-VP by introducing *explicit routed substitutions* into the language and including operational rules defining τ transitions for terms involving routed substitutions. As will be seen, processes that are "farther apart" in a term will in general require more τ-transitions to communicate than those closer together; this operational semantics may be then used with a semantic preorder that is sensitive to the number of τ actions performed (see, e.g, [3]) to relate systems on the basis of their relative *efficiency*.

Formally, we alter the syntax of ACCS-VP to include explicit routed substitutions as follows. Let s be a process address.

$$\begin{array}{rcl} P & ::= & nil \mid \alpha V.P \mid \overline{\alpha}E.P \mid P+P \mid P|P \\ & \mid & P\backslash L \mid P[f] \mid x \mid rec\, x.\, P \mid \overline{\alpha}E \rhd P \\ & \mid & P\{E/V\}@s \end{array}$$

As before, we consider only guarded terms. The new set of processes is denoted by $\mathcal{P}_e$, which will be ranged over by $P, Q, \ldots$.

To define the semantics of the altered language we first introduce rules for resolving explicit substitutions; these appear in Figure 4. The reader should note the similarity between

Normal Rules

$$
\begin{array}{llll}
P\{E/V\}@\bullet & \stackrel{\tau}{\longmapsto} & P[E/V] & \\
(P|Q)\{E/V\}@s\rceil & \stackrel{\tau}{\longmapsto} & (P\{E/V\}@s)|Q & \\
(P|Q)\{E/V\}@s\lceil & \stackrel{\tau}{\longmapsto} & P|(Q\{E/V\}@s) & \\
(P\backslash L)\{E/V\}@s & \stackrel{\tau}{\longmapsto} & (P\{E/V\}@s)\backslash L & \text{if } s \neq \bullet \\
(P[f])\{E/V\}@s & \stackrel{\tau}{\longmapsto} & (P\{E/V\}@s)[f] & \text{if } s \neq \bullet \\
(\overline{\alpha}E \rhd P)\{E/V\}@s & \stackrel{\tau}{\longmapsto} & \overline{\alpha}E \rhd (P\{E/V\}@s) & \text{if } s \neq \bullet
\end{array}
$$

Exception Rules

$$
\begin{array}{llll}
nil\{E/V\}@s & \stackrel{\tau}{\longmapsto} & nil & \text{if } s \neq \bullet \\
(a.P)\{E/V\}@s & \stackrel{\tau}{\longmapsto} & a.P & \text{if } s \neq \bullet \\
((\pi).P)\{E/V\}@s & \stackrel{\tau}{\longmapsto} & (\pi).P & \text{if } s \neq \bullet \\
(P+Q)\{E/V\}@s & \stackrel{\tau}{\longmapsto} & P+Q & \text{if } s \neq \bullet \\
rec\,x.\ P\{E/V\}@s & \stackrel{\tau}{\longmapsto} & rec\,x.\ P & \text{if } s \neq \bullet
\end{array}
$$

Figure 4: Operational Rules for Routed Substitutions.

the transitions defined for terms with routed substitutions and definition of routed substitution given in Definition 4.1. We also classify the rules as "normal" rules, which drive a substitution to the place where it has to be applied, and "exception" rules, which deal with processes containing substitutions with addresses that would not arise during the execution of a "normal" system. The complete semantics consists of the rules of Figure 4 and the ones of Figure 3 except for (comml') and (commr'), which are replaced by the following.

$$
\begin{array}{l}
P \stackrel{\alpha V @ s}{\longmapsto} P', Q \stackrel{\overline{\alpha}E}{\longmapsto} Q' \ \Rightarrow \\
P|Q \stackrel{\tau}{\longmapsto} (P'\{E/V\}@s)|Q' \quad (comml'')
\end{array}
$$

$$
\begin{array}{l}
P \stackrel{\overline{\alpha}E}{\longmapsto} P', Q \stackrel{\alpha V @ s}{\longmapsto} Q' \ \Rightarrow \\
P|Q \stackrel{\tau}{\longmapsto} P'|(Q\{E/V\}@s) \quad (commr'')
\end{array}
$$

The following property shows that the rules of Figure 4 actually define a function.

Property 6.3 *The set of rules of Figure 4 is* deterministic*: if $P\{E/V\}@s \in \mathcal{P}_e$ then there is at most one $P' \in \mathcal{P}_e$ such that $P \stackrel{\tau}{\longmapsto} P'$.*

We now show that the rules for explicit substitution are consistent with the definition of routed substitution. Let $P \stackrel{\tau}{\longmapsto}^* P'$ if there is a sequence of τ-labeled transitions from P to P', and $P \longmapsto^* P'$ if there is a sequence of any transitions from P to P'.

Theorem 6.4 *Let $P \in \mathcal{P}_A$, $V \in \mathcal{V}$, $E \in \mathcal{E}$ and $s \in \mathcal{S}$. Then $P\{E/V\}@s \stackrel{\tau}{\longmapsto}^* P[E/V]@s$.*

Proof. By structural induction on the definition of $P[E/V]@s$. For each equation in the definition of the form $P[E/V]@s = Q$ we have a rule $P\{E/V\}@s \overset{\tau}{\longmapsto} Q$ in Figure 4 and vice versa. $\square$

The next result establishes that during "normal operation", the Exception rules are never used.

Lemma 6.5 *Let $P \in \mathcal{P}_A$ and $Q \in \mathcal{P}_e$ be such that $P \longmapsto^* Q$. Then there exists $P' \in \mathcal{P}_A$ such that $Q \overset{\tau}{\longmapsto}^* P'$ can be proved without using Exception rules.*

Proof. The proof has two parts. First, by induction on the length of $\longmapsto^*$ it is easy to show that the process addresses in Q always refer to subterms of Q. The desired result follows by structural induction on Q. $\square$

To illustrate this theorem, consider the term $P_1 \in \mathcal{P}_A$ given by

$$P_1 \equiv (\alpha V.nil \,|\, \overline{\beta} E'.nil) \,|\, \overline{\alpha} E.nil.$$

The following sequence of τ-transitions, each deducible using the Normal rules, is the one guaranteed by the theorem.

$$\begin{array}{lcl} P_1 & \overset{\tau}{\longmapsto} & (nil \,|\, \overline{\beta} E'.nil)[E/V]@\bullet\rceil \,|\, nil \\ & \overset{\tau}{\longmapsto} & (nil[E/V]@\bullet \,|\, \overline{\beta} E'.nil) \,|\, nil \\ & \overset{\tau}{\longmapsto} & (nil[E/V] \,|\, \overline{\beta} E'.nil) \,|\, nil \equiv (nil \,|\, \overline{\beta} E'.nil) \end{array}$$

By way of contrast, the last step in the sequence of transitions for

$$P_2 \equiv (\alpha V.nil | \overline{\beta} E.nil)[E'/V']@\bullet\lceil\rceil$$

requires the use of an Exception rule.

$$\begin{array}{lcl} P_2 & \overset{\tau}{\longmapsto} & ((\alpha V.nil[E'/V']@\bullet\lceil) | \overline{\beta} E.nil) \\ & \overset{\tau}{\longmapsto} & (\alpha V.nil | \overline{\beta} E.nil) \end{array}$$

Discussion

We close this subsection by remarking on two issues that our treatment of explicit routed substitutions has not addressed. The first concerns accounting for "output costs." Suppose that in a term of the form $((P|Q)|R)|S$, both P and R wish to perform an output $\overline{\alpha} E$ and S is willing to engage in $\alpha V@\bullet$. Then our semantics would conclude that the communication costs for the two message deliveries would be the same: in both cases a sequence of two τ-transitions would be required to complete the transfer from P/R to S, even though in some sense R is "closer" to S than P. One approach to accounting more accurately for output costs would be to associate addresses to output actions as well as input actions. Then the semantics of communication could be changed to introduce τ's to model the "hops" needed for an output action to reach the point at which it "synchronizes" with an input action to yield an explicit substitution.

The second concerns the fact that the semantics of explicit substitutions in essence *suspends* the execution of a process to which a substitution is being applied until (one step of) the substitution is resolved. That is, the operational definition does not include a rule of the form

$$P \stackrel{a}{\longmapsto} P' \Rightarrow P\{E/V\}@s \stackrel{a}{\longmapsto} P'\{E/V\}@s.$$

In general, introducing such a rule into our semantics would lead to problems with free variables; for example, in

$$(\overline{\alpha}V.P|(Q|R))\{E/V\}@\bullet\rceil,$$

the process inside the substitution is capable of a $\stackrel{\overline{\alpha}V}{\longmapsto}$ transition; however, this transition should not be allowed until the substitution of E for V is performed. On the other hand, not having the above rule can limit the degree of concurrency exhibited by a system. Consider a process of the form $(P|(Q|R))\{E/V\}@\bullet\rceil$; process P is awaiting the delivery of E, which will then be substituted for its free occurrences of V. The only transition available to this system involves the resolution of the substitution as follows.

$$(P|(Q|R))\{E/V\}@\bullet\rceil \stackrel{\tau}{\longmapsto} P\{E/V\}@\bullet|(Q|R)$$

Note that even if Q and R wish to communicate, they are unable to do so until the above transition takes place, even though the substitution will not affect the communication they wish to engage in. With a rule such as the one above, Q and R would be allowed to engage in their synchronization without having to wait for the resolution of the substitution. One approach to addressing these issues would involve the introduction of an *explicit suspension* operator into the language. A process would be suspended when it executes an input transition, and a suspended process would be incapable of certain transitions (e.g. output transitions in which the value expression contains a free variable) until a routed substitution is applied to it. With this notion of suspension made explicit, one would then be able to add the above rule to our operational semantics without disturbing the treatment of free variables.

7 Conclusions

In this paper we have presented an alternative semantics for a concurrent language with value passing. Inspired by the operational considerations that are evident in the reduction-style semantics given to the λ-calculus, this semantics attempts more faithfully to reflect abstractly the operations concerns associated with the exchange of values between processes distributed in space. Accordingly, a notion of *routed substitution* is defined and used to model the synchronous transfer of values between processes. A notion of operational equivalence is also defined and shown to coincide with the well-known bisimulation equivalence of CCS. Thus the new semantics maintains consistency with the traditional semantics given to value-passing languages while providing more information about how data is passed between processes. We then show how the semantics may be altered smoothly to support different forms of asynchronous communication and to provide more information regarding the cost associated with transferring data between processes.

There are a number of directions in which this work can be extended. In this paper we only consider binary communication schemes. One could also alter the framework slightly to support multi-party communication; one way to do so would involve the introduction of an additional indicator in the syntax of addresses to signal that a data value should be routed to both subterms in a parallel composition. Developing an axiomatization for the equivalence proposed in Section 4 for CCS-VP and ACCS-VP would also yield an interesting method for establishing that processes are equivalent. In the context of explicit routed substitutions, it would also be worthwhile to examine notations for describing non-tree structured networks. As the parallel composition operator in ACCS-VP is binary, it only permits the description of networks organized as binary trees. Developing a syntax for expressing more complicated graphical structures (e.g. rings, hypercubes) would provide a basis for investigating the relative efficiency of different architectures for different problems.

Finally, it would also be interesting to extend the language in this paper by considering different types of values and permitting data variables to have types ascribed to them. One could alter the reduction semantics for value-passing by introducing a sensitivity to type: input actions could be labeled by the type of the variable they bind, and a reduction would take place only if the type of the output expression matches that of the variable. In this way one could obtain a (first-order) typed reduction system for concurrent processes. Several interesting directions for further inquiry would then present themselves. One would be to investigate the development of a higher-order typed language for processes. Another would be the extraction of "pure" (finite-state) process algebra terms from typed terms that would retain some of the behavioral properties of the original (infinite-state) terms. For example, one could imagine taking a term of the form $\alpha V : Int.P$ and generating a term of the form $\alpha_{Int}.T(P)$, where $T(P)$ is the "type" for P. To the extent that such a procedure would yield a term reflecting at an abstract level the behavior and types of passed values of the original term, useful behavioral information could be automatically computed for concurrent programs.

Related Work

Traditionally, researchers in process algebra have defined the semantics of value-passing by "reducing" it to pure synchronization in the fashion described in Section 3 [20, 22, 27]. One approach to avoiding the attendant infinite number of transitions that this entails can be found in [16, 17]. There the semantics dictates that processes such as $\alpha X.P$ may engage in the single transition $\alpha X.P \stackrel{\alpha X}{\rightarrow} P$; when processes synchronize, then, data transfer is handled via substitution in the usual sense. The approach is similar to ours, although it differs in that unnecessary substitutions can be performed, with an attendant need to rename bound variables on occasion. For example, consider the term $\alpha X.P|\overline{\beta}X.Q$; note that X is free in the right-hand processes. If this system is allowed to engage in the transition

$$\alpha X.P|\overline{\beta}X.Q \stackrel{\alpha X}{\to} P|\overline{\beta}X.Q$$

then X is now free in both the left-hand and right-hand sides. However, since their semantics requires that a value arriving to this term as the result of a synchronization on α be substituted X into both processes, the free occurrence of X in the right-hand side process would incorrectly be replaced. To ensure this does not happen, the X in the left-hand process must be renamed. In contrast, our semantics would not require the renaming of X in here, because the routing operation would ensure that substitution is only performed for occurrences of X on the left.

The Chemical Abstract Machine (ChAM) [8] represents another approach to defining the operational semantics of concurrent systems. The ChAM conceives of systems of processes as "solutions", with individual processes representing "molecules" that may collide to form larger processes with parallel components which may then interact and break apart. By design, this framework abstracts away from any underlying system geometry, and hence it is likely to be unsuitable for the implementation-level, efficiency-based analyses that our framework, via explicit routed substitutions, can support.

Acknowledgements. The first author would like to thank Bard Bloom and S. Purushothaman Iyer for their insightful comments on an earlier version of this paper.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 31–46, San Francisco, January 1990. Computer Society Press.

[2] J.-M. Andreoli and R. Pareschi. Linear objects: logical processes with built-in inheritance. In *Proceedings OOPSLA '91*, pages 1–20. ACM Press, 1991.

[3] S. Arun-Kumar and M. Hennessy. An efficiency preorder for processes. Technical Report 5/90, Computer Science, University of Sussex, Brighton, 1990.

[4] J. Baetan and W. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1990.

[5] J. P. Banatre and D. L. Metayer. Introduction to Gamma. In *Research Directions in High-Level Parallel Programming Languages*, volume 574 of *Lecture Notes in Computer Science*, pages 197–202. Springer-Verlag, 1991.

[6] H. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1984.

[7] J. Bergstra and J. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[8] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[9] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 85–98. Computer Society Press, January 1993.

[10] S. Brookes, C. Hoare, and A. Roscoe. A theory of communicating sequential processes. *Journal of the Association for Computing Machinery*, 31(3):560–599, July 1984.

[11] S. Budkowski and P. Dembinski. An introduction to Estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems*, (14):3–23, 1987.

[12] F. de Boer, J. Klop, and C. Palamidessi. Asynchronous communication in process algebra. In *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 137–147, Santa Cruz, CA, June 1991. Computer Society Press.

[13] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1983.

[14] D. Gelernter. Generative communication in Linda. *ACM Trans. on Programming Languages and Systems*, 7(1):80–112, 1985.

[15] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, Boston, 1988.

[16] M. Hennessy and H. Lin. Proof theories for symbolic bisimulation. Technical report, Sussex University, forthcomming 1992.

[17] M. Hennessy and H. Lin. Symbolic bisimulations. Technical Report 1/92, Sussex University, 1992.

[18] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall, London, 1985.

[19] A. Ingólfsdóttir and B. Thomsen. Semantic models for ccs with values. In *Proc. of Chalmers Workshop on Concurrency*, Baastad, Sweden, May 1991.

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

[21] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[22] R. Milner. *A Calculus of Communicating Systems.* Prentice-Hall, London, 1989.

[23] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Part I + II. *Information and Control*, 100(1):1–77, 1992.

[24] U. Montanari and D. Yankelevich. A parametric approach to localities. In *Proceedings* 19^{th} *ICALP,* Vienna, volume 623 of *Lecture Notes in Computer Science*, pages 617–628. Springer-Verlag, 1992.

[25] V. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, 1989. Also in ACM Distinguished Dissertation Series.

[26] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 3(21):412–510, 1989.

[27] B. Thomsen. A calculus of higher-order communicating systems. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 143–154, Austin, Texas, January 1989. Computer Society Press.

[28] R. Tolksdorf. Alice: Basic model and subtyping agents. Technical Report 1993/7, Graduiertenkolleg kommunikationsbasierte Systeme, TU Berlin, Berlin, Germany., 1993.

[29] D. Walker. Bisimulation and divergence in CCS. In *Third Annual Symposium on Logic in Computer Science*, pages 186–192, Edinburgh, July 1988. Computer Society Press.