An Operational Semantics of Value Passing*

Rance Cleaveland
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@csc.ncsu.edu

July 21, 1993

Abstract

This paper develops an operational semantics for concurrent languages with value passing. An operation analogous to substitution in the λ-calculus is given, and an operational equivalence is defined and shown to coincide with Milner's bisimulation equivalence. In contrast with existing process-algebraic approaches to value passing, this semantics does not reduce data exchange to pure synchronization over (potentially infinite) families of ports indexed by data.

1 Introduction

Process algebras such as CCS [13], CSP [10] and ACP [2] have been proposed as theories that highlight fundamental notions in concurrency in the same way that the λ-calculus [3] does for functional programming. Although the latter essentially focuses on only one programming construct, 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 [4, 5, 6, 7, 11, 12, 16] have been defined that relate concurrent systems on the basis of their externally visible

*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.

behavior, and the theories have been successfully applied to the verification of concurrent systems. However, researchers in process algebra have devoted relatively little attention 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 understanding 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 situation in 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 some kind 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 defines 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, and 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; the semantics should be implementable, and therefore 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 places in the term 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 [14]), and a common mechanism, recursive definition, for building processes will be omitted. Including these is not problematic, but we have chosen to exclude them in the interests of notational simplicity.

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 variables ranged over by $V, V', V_1, \ldots$ and $\mathcal{E}$ be a set of value expressions ranged over by $E, E', E_1, \ldots$; these expressions may contain occurrences of variables V_i. Also let Λ be a (countably infinite) 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 $\Lambda \rightarrow \Lambda$. The syntax of CCS-VP may now be given using the following grammar.

$$P \quad ::= \quad nil \mid \alpha V.P \mid \overline{\alpha}E.P \mid P + P \mid P|P \mid P\backslash L \mid P[f]$$

In the sequel we use $\mathcal{P}$ to denote the set of all 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 $\mathcal{A} = \{\, \alpha E \mid \alpha \in \Lambda \wedge E \in \mathcal{E} \,\} \cup \{\, \overline{\alpha}E \mid \alpha \in \Lambda \wedge E \in \mathcal{E} \,\} \cup \{\tau\}$ 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 $\overline{a}$ 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 $\pi \in \mathcal{A} \rightarrow \Lambda \cup \{\tau\}$ returns the "port component" of an action; so $\pi(\alpha E) = \pi(\overline{\alpha}E) = \alpha$, and $\pi(\tau) = \tau$. Also, if $f \in \Lambda \rightarrow \Lambda$ 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 also assume that $\mathcal{E}$ is 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. We further assume that for any $E \in \mathcal{E}, |\, fv\ (E)| < \infty$ and that there exists a function $\eta \in 2^{\mathcal{V}} \rightarrow \mathcal{V}$ with the property that for any finite $\mathcal{V}' \subseteq \mathcal{V}, \eta(\mathcal{V}') \notin \mathcal{V}'$. We now define substitution for CCS-VP as follows.

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

- $nil[E/V] = nil$.
- $(\alpha V'.P)[E/V] = \begin{cases} \alpha V'.P & \text{if } V \equiv V' \\ \alpha V'.(P[E/V]) & \text{if } V \not\equiv V' \text{ and } V' \notin fv(E) \\ \alpha V''.(P[V''/V'][E/V]) & \text{otherwise} \end{cases}$

  where $V'' = \eta(fv(E) \cup \{V\})$. Notice that the final clause handles the problem of variable capture in the usual fashion by renaming bound variables as necessary.
- $(\overline{\alpha}E'.P)[E/V] = \overline{\alpha}(E'[V := E]).(P[E/V])$.
- $(P_1 + P_2)[E/V] = (P_1[E/V]) + (P_2[E/V])$.

$$
\begin{array}{lcl}
\alpha V.P \stackrel{\alpha E}{\rightarrow} P[E/V] & \text{for any } E \in \mathcal{E} & \\
\overline{\alpha} E.P \stackrel{\overline{\alpha} E}{\rightarrow} P & & \\
P \stackrel{a}{\rightarrow} P' & \Rightarrow & P + Q \stackrel{a}{\rightarrow} P' \\
Q \stackrel{a}{\rightarrow} Q' & \Rightarrow & P + Q \stackrel{a}{\rightarrow} Q' \\
P \stackrel{a}{\rightarrow} P' & \Rightarrow & P|Q \stackrel{a}{\rightarrow} P'|Q \\
Q \stackrel{a}{\rightarrow} Q' & \Rightarrow & P|Q \stackrel{a}{\rightarrow} P|Q' \\
P \stackrel{a}{\rightarrow} P', Q \stackrel{\overline{a}}{\rightarrow} Q' & \Rightarrow & P|Q \stackrel{\tau}{\rightarrow} P'|Q' \\
P \stackrel{a}{\rightarrow} P', \pi(a) \notin L & \Rightarrow & P \backslash L \stackrel{a}{\rightarrow} P' \backslash L \\
P \stackrel{a}{\rightarrow} P' & \Rightarrow & P[f] \stackrel{apply(f,a)}{\rightarrow} P'[f]
\end{array}
$$

Figure 1: The Traditional Semantics of CCS-VP.

- $(P_1|P_2)[E/V] = (P_1[E/V])|(P_2[E/V])$.
- $P\backslash L[E/V] = (P[E/V])\backslash L$.
- $P[f][E/V] = (P[E/V])[f]$.

The traditional semantics of CCS-VP is given as a relation $\rightarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$, where, $P \stackrel{a}{\rightarrow} P'$ 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. 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 the process whose initial action was chosen. $P_1|P_2$ 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 the same port, 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. Finally, in $P[f]$ f is used to rename the labels that P interacts on.

Having defined the operational semantics of CCS-VP processes, we now present the definition of *bisimulation equivalence* [13], which is designed to related 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 $P \stackrel{a}{\rightarrow} P'$ then there is a Q' such that $Q \stackrel{a}{\rightarrow} Q'$ and $P'\ \mathcal{R} Q'$.*
2. *If $Q \stackrel{a}{\rightarrow} Q'$ then there is a P' such that $P \stackrel{a}{\rightarrow} P'$ 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 given 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 in a certain sense 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 a new syntactic category of *input-enabled* terms and an operation analogous to substitution that "channels" an input into one of these terms. Using the new semantics, we define a notion of behavioral equivalence that, as the next section shows, corresponds to bisimulation equivalence.

The set $\mathcal{I}$ of input-enabled terms is given by the following grammar.

$$I ::= V.P \;\mid\; I\rceil P \;\mid\; P\lceil I \;\mid\; I\backslash L \;\mid\; I[f]$$

Here $P \in \mathcal{P}$ is as defined in Section 3. As was the case there, $L \subseteq \Lambda$ and $f \in \Lambda \to \Lambda$. We use $I, I', I_1, \ldots$ to range over $\mathcal{I}$ and $K, K', K_1, \ldots$ to range over $\mathcal{P} \cup \mathcal{I}$.

Intuitively, an input-enabled term results when a process has indicated that it is ready to receive an input on a particular channel and is awaiting the value that it will substitute for the appropriate variable occurrences. In $V.P$, for example, all occurrences V in the process P are to be replaced by the value that eventually arrives. $I\rceil P$ results when process Q in $Q|P$ signals a desire for an input; the $\rceil$ indicates that when the value arrives, it should be routed into the left component. Operator $\lceil$ plays the symmetric role to $\rceil$. Finally, $\backslash L$ and $[f]$ are as in Section 3.

In order formally to specify the new semantics, we first introduce the new set of actions we consider. The set $\mathcal{A}_{in} = \Lambda$ and 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 $\mathcal{A}'$. The functions $\pi' \in \mathcal{A}' \to \Lambda \cup \{\tau\}$ and $apply' \in (\Lambda \to \Lambda) \times \mathcal{A}' \to \mathcal{A}'$ are now

defined by:

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

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

We now define the *routing* operation that "delivers" a value to the appropriate places in an input-enabled term.

Definition 4.1 *Let $I \in \mathcal{I}$ and $E \in \mathcal{E}$. Then $I \triangleleft E$ is defined inductively as follows.*

- $(V.P) \triangleleft E = P[E/V]$.
- $(I \rceil P) \triangleleft E = (I \triangleleft E) | P$.
- $(P \lceil I) \triangleleft E = P | (I \triangleleft E)$.
- $(I \backslash L) \triangleleft E = (I \triangleleft E) \backslash L$.
- $(I[f]) \triangleleft E = (I \triangleleft E)[f]$.

It is straightforward to establish the following.

Lemma 4.2 *Let $I \in \mathcal{I}$ and $E \in \mathcal{E}$. Then $(I \triangleleft E) \in \mathcal{P}$.*

The operational semantics is given in terms of a relation $\mapsto \subseteq (\mathcal{P} \times \mathcal{A}' \times (\mathcal{P} \cup \mathcal{I}))$. The formal definition appears in Figure 2; the rough intuition underlying the rules is the following. The operations $+, \backslash L$, and $[f]$ behave essentially as they do the old semantics, as does $\overline{\alpha}E.P$ (i.e. prefixing by an output action). A process of the form $\alpha V.P$ signals its desire for an input on α by executing the action α and then evolving to the *input-enabled* term $V.P$, where it awaits a value to be routed to it from the environment. The rules for $|$ are slightly more complicated than before, since terms must be decorated with appropriate routing information. Thus, if P requests an input on α and evolves to input-enabled term I, then $P|Q$ must evolve to the input-enabled term $I \rceil Q$ to indicate that its left subterm is awaiting a value. If P wishes to perform an output without synchronizing with P, then $P|Q$ behaves as in the old semantics. Finally, if P signals a desire for an input on α and Q is willing to perform an output on the same port, then the two may synchronize, with a τ resulting and the appropriate value expression being routed to P. The dual situation holds for Q.

As an illustration of semantics, consider the following term.

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

$$
\begin{array}{lcl}
\alpha V.P \stackrel{\alpha}{\mapsto} V.P & & \\
\overline{\alpha}E.P \stackrel{\overline{\alpha}E}{\mapsto} P & & \\
P \stackrel{a}{\mapsto} K & \Rightarrow & P + Q \stackrel{a}{\mapsto} K \\
Q \stackrel{a}{\mapsto} K & \Rightarrow & P + Q \stackrel{a}{\mapsto} K \\
P \stackrel{\alpha}{\mapsto} I & \Rightarrow & P|Q \stackrel{\alpha}{\mapsto} I\rceil Q \\
P \stackrel{a}{\mapsto} P', a \in \mathcal{A}_o & \Rightarrow & P|Q \stackrel{a}{\mapsto} P'|Q \\
Q \stackrel{\alpha}{\mapsto} I & \Rightarrow & P|Q \stackrel{\alpha}{\mapsto} P\lceil I \\
Q \stackrel{a}{\mapsto} Q', a \in \mathcal{A}_o & \Rightarrow & P|Q \stackrel{a}{\mapsto} P|Q' \\
P \stackrel{\alpha}{\mapsto} I, Q \stackrel{\overline{\alpha}E}{\mapsto} Q' & \Rightarrow & P|Q \stackrel{\tau}{\mapsto} (I \triangleleft E)|Q' \\
P \stackrel{\overline{\alpha}E}{\mapsto} P', Q \stackrel{\alpha}{\mapsto} I & \Rightarrow & P|Q \stackrel{\tau}{\mapsto} P'|(I \triangleleft E) \\
P \stackrel{a}{\mapsto} K, \pi'(a) \notin L & \Rightarrow & P\backslash L \stackrel{a}{\mapsto} K\backslash L \\
P \stackrel{a}{\mapsto} K & \Rightarrow & P[f] \stackrel{apply'(f,a)}{\mapsto} K[f]
\end{array}
$$

Figure 2: A New Semantics for Value Passing.

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

$$
\begin{array}{lcl}
P & \stackrel{\alpha}{\mapsto} & (X.\overline{\beta}X.nil\rceil\beta Y.nil)\rceil\overline{\alpha}E.nil \\
P & \stackrel{\beta}{\mapsto} & (\alpha X.\overline{\beta}X.nil\lceil Y.nil)\rceil\overline{\alpha}E.nil \\
P & \stackrel{\overline{\alpha}E}{\mapsto} & (\alpha X.\overline{\beta}X.nil|\beta Y.nil)|nil \\
P & \stackrel{\tau}{\mapsto} & (X.\overline{\beta}X.nil\rceil\beta Y.nil) \triangleleft E|nil \\
 & & \equiv ((X.\overline{\beta}X.nil) \triangleleft E|\beta Y.nil)nil \\
 & & \equiv (\overline{\beta}E.nil|\beta Y.nil)|nil
\end{array}
$$

The last of these transitions illustrates how the $\lceil$ and $\rceil$ operators to guide incoming values to the appropriate subterm, where they are then substituted for the indicated variable. Notice also that in the transfer of data in this 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 implementable stems from the following result.

Theorem 4.3 *Let $P \in \mathcal{P}$, and let $\mathcal{T} = \{ \langle a, K \rangle \mid P \stackrel{a}{\mapsto} K \}$. 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 an 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

As transitions are not defined for elements of $\mathcal{I}$, we cannot directly apply the definition of $\sim$ given in Section 3. Instead, we define the following.

Definition 4.4 *A relation $\mathcal{R} \subseteq (\mathcal{P} \times \mathcal{P}) \cup (\mathcal{I} \times \mathcal{I})$ is an input-sensitive bisimulation if, whenever $K_1 \mathcal{R} K_2$ then the following hold.*

1. *If $\langle K_1, K_2 \rangle \in \mathcal{P} \times \mathcal{P}$ then for all $a \in \mathcal{A}'$:*
   (a) *$K_1 \stackrel{a}{\mapsto} K_1'$ implies there is a K_2' such that $K_2 \stackrel{a}{\mapsto} K_2'$ and $K_1' \mathcal{R} K_2'$.*
   (b) *$K_2 \stackrel{a}{\mapsto} K_2'$ implies there is a K_1' such that $K_1 \stackrel{a}{\mapsto} K_1'$ and $K_1' \mathcal{R} K_2'$.*
2. *If $\langle K_1, K_2 \rangle \in \mathcal{I} \times \mathcal{I}$ then for all $E \in \mathcal{E}, (K_1 \triangleleft E) \mathcal{R} (K_2 \triangleleft E)$.*

$K_1 \simeq K_2$ if there is an input-sensitive bisimulation $\mathcal{R}$ such that $K_1 \mathcal{R} K_2$.

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, when restricted to $\mathcal{P} \times \mathcal{P}$, $\simeq$ relates exactly the same processes as $\sim$. The importance of this result lies in the fact that it in some since 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 $P \stackrel{\alpha E}{\rightarrow} P'$ if and only if there exists an $I \in \mathcal{I}$ such that $P \stackrel{\alpha}{\mapsto} I$ and $P' \equiv I \triangleleft E$.*

Proof. Proceeds by induction on the derivations of $P \stackrel{\alpha E}{\rightarrow} P'$ and $P \stackrel{\alpha}{\mapsto} I$. □

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

Proof. Proceeds by induction on the derivations of $P \stackrel{a}{\rightarrow} P'$ and $P \stackrel{a}{\mapsto} P'$. □

Using these lemmas, we can now prove the following theorem.

Theorem 5.3 *Let $P_1, P_2 \in \mathcal{P}$. Then $P_1 \sim P_2$ if and only if $P_1 \simeq P_2$.*

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. □

6 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. A notion of operational equivalence is 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.

There are a number of directions in which this work can be extended. In this paper we only considered binary synchronous communication in the style of CCS. One could also alter the framework slightly to support synchronous multi-party communication; one way to do so would involve the introduction of an additional "routing" operator for input-enabled terms that would indicate that a data value should be routed to both subterms. It would also be interesting to examine the semantics in this paper in light of the criteria proposed in [15] for an operational semantics to be effective. This could lead to a formalization of the informal argument of implementability given here. Another direction for future research would involve extending the syntax of process terms to permit *explicit routings and substitutions* in the same way that the λ-calculus has been extended with explicit substitutions in, e.g., [1]. In the λ-calculus, explicit substitutions permit a finer study of the cost of reducing one term to another. In addition to enabling a more detailed study of "communication costs" à la the λ-calculus, explicit routings could also be used as a basis for giving a reduction-like semantics for asynchronous communication in which senders need not wait for receivers. To see how this might be done, consider the process

$$T \equiv (\alpha X.P | \beta Y.Q) | \overline{\alpha} E.R.$$

With terms of the form $I \triangleleft E$ allowed as processes, one could imagine defining a semantics in which T could have the following transition.

$$T \stackrel{\tau}{\rightarrow} ((\alpha X.P] \beta Y.Q) \triangleleft E) | R$$

Thus, before the $\triangleleft E$ operation is fully resolved, R could engage in another transition. A number of details would need to be worked out, but the approach appears promising.

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 [11, 13, 14]. One approach to avoiding the attendant infinite number of transitions that this entails can be found in [8, 9]. 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

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

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.

Acknowledgements. I would like to thank Bard Bloom and S. Purushothaman 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. Baetan and W. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1990.

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

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

[5] 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.

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

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

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

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

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

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

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

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

[14] 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.

[15] F. Vaandrager. Expressiveness results for process algebras. Technical Report CS-R9301, University of Amsterdam, Programming Research Group, 1993.

[16] 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.