An Object Calculus for Asynchronous Communication*

Kohei Honda and Mario Tokoro†

Department of Computer Science,
Keio University,
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223,
Japan

Abstract

This paper presents a formal system based on the notion of objects and asynchronous communication. Built on Milner's work on π-calculus, the communication primitive of the formal system is purely asynchronous, which makes it unique among various concurrency formalisms. Computationally this results in a consistent reduction of Milner's calculus, while retaining the same expressive power. Seen semantically asynchronous communication induces a surprisingly different framework where bisimulation is strictly more general than its synchronous counterpart. This paper shows basic construction of the formal system along with several illustrative examples.

1 Introduction

The formal system introduced in this paper is intended to accomplish two purposes. First, it provides a simple and rigorous formalism which encapsulates essential features of concurrent object-orientation [26, 25]. Being successful as a programming methodology for dynamic concurrent computing, its theoretical contents are far from well-understood,[1] leaving theorists and practitioners without a core theory on which they can reason and develop further ideas. Second, it offers a possible foundation for concurrency theory which is quite different from the usual one in the sense that the formalism is purely based on asynchronous communication, both computationally and semantically. The usual observation-based equivalence theory for processes is recaptured as asynchronous bisimulation for objects where asynchronous experiments induces a somewhat more general semantic framework.

*Appeared in *Proc. The Fifth European Conference on Object-Oriented Programming*, LNCS, July 1991,Springer-Verlag. Revised in August 1991.

†Also with Sony Computer Science Laboratory Inc. 3-14-13 Higashi-Gotanda, Shinagawa-ku, Tokyo, 141, Japan

[1]Though recently several important works appeared in this context including [6, 12, 17].

The formalism is based on a series of studies on a port passing process calculus now called π-calculus, initiated by Nielsen and Engberg [18], reformulated by Milner and his colleagues [15], and developed in a crucial way by Milner [17]. Especially [17] has been essential in our construction due to its separation of *structural rules* from *transition rules*, and in its distinction between *computational transition* and *semantic transition*. One interesting thing is that the capability to generate and pass communication ports turns out to be essential not only for object-orientation (which is obvious) but also for control of causality chains in the face of pure asynchrony. This reminds us of the studies on the actor model of computation by Hewitt and his colleagues [8, 7, 5, 1]. Also readers may refer to the authors' work on conceptual framework for open distributed computing environments [22, 23] to understand their general orientation in a different context.

This paper only provides basic concepts and definitions for the formal system along with several illustrative examples, leaving the full presentation of our theoretical results to the coming exposition. Section 2 defines the basic syntax and other constructs of the formal system. Section 3 introduces reduction relation which defines the basic computational mechanism in combination with structural rules. Section 4 provides important primitive constructs for our system and shows that they can be used to encode a fragment of π-calculus [17] which is a superset of our formalism. Section 5 reviews the general semantic framework of asynchronous bisimulation, giving basic definitions and examples. Object-orientation in our semantic framework is also discussed. Finally Section 6 concludes the paper.

2 Syntax and Bindings

This section first briefly summarizes the basic idea of our formalism, then provides definitions for syntax and bindings.

Basic Framework

In the formalism presented hereafter, the notions of objects and communication are captured in the following way.

- **An object** is a collection of *receptors* and *pending messages*. A receptor has a *handle* (an input port) and a *carrier* (a formal parameter) at its *head*, and consumes a message to receive the value at its carrier which carries the value to its *body*. Then this body generates zero or more receptors and zero or more messages. The original receptor just disappears. Notationally, it is expressed as $ax.P$ (a is a handle, x is a carrier, and P is a body). All receptors within an object may operate concurrently and asynchronously.

- **A message** is a simple data structure which carries a piece of information to its target. It should have its *target* and *value*, each being supposed to be a *port name* (i.e. no value is considered except names of ports themselves). The notation for a message is $\leftarrow av$ (a is a target, v is a value). Some of generated messages may go out (becoming output messages), some may be consumed by receptors (causing internal configuration change), and some may be just pending within the object.

The computational implication of this framework is that a message will be consumed by a receptor if and only if its target is the same as the handle of the receptor, that is, $\leftarrow av$ will be consumed by $ax.P$, but not by $bx.P$. Existence of multiple receptors in an object implies an object may have multiple input ports, possibly with duplicate names. The port names are only values to be considered here, sent by messages and consumed by receptors. A configuration will generate new port names which extends the domain of computation.

Syntax

Syntactically our formal system *reduces*, not extends, constructs in process calculi[13, 14, 9, 16], to incorporate asynchronous communication. The key idea is to express asynchronous messages as output processes without subsequent behaviour. That is, $\bar{a}v.P$ (a process which outputs v through port a) is reduced to $\leftarrow av$ (a message to the target a with a value v). In Section 4 we will see that this reduction does not result in loss of expressive power. Below are syntactic definitions of port names, messages, receptors (including recursively defined ones), and general configurations called term expressions. Term variables are necessary for recursively defined receptors. In the right-hand side of each definition, we give formal designation for these syntactic constructs.

Definition 1 The sets of port names $\mathbf{N}$, of sequences of port names $\widetilde{\mathbf{N}}$, of term variables $\mathbf{V}$, and of term expressions $\mathbf{C}$, are given by the following abstract syntax.

$$
\begin{array}{rcll}
\mathbf{N} & = & x \mid \mathbf{N}' & \text{(port names)} \\
\tilde{\mathbf{N}} & = & \varepsilon \mid \mathbf{N}\widetilde{\mathbf{N}} & \text{(sequences of names)} \\
\mathbf{V} & = & X \mid \mathbf{V}' & \text{(term variables)} \\
\mathbf{C} & = & \leftarrow \mathbf{NN} & \text{(a message)} \\
 & \mid & \mathbf{NN.C} & \text{(a receptor)} \\
 & \mid & \{\mathbf{V}(\widetilde{\mathbf{N}}) = \mathbf{NN.C}\}(\widetilde{\mathbf{N}}) & \text{(a recursively defined receptor)} \\
 & \mid & \mathbf{V}(\widetilde{\mathbf{N}}) & \text{(a term variable with parameters))} \\
 & \mid & |\mathbf{N}|\mathbf{C} & \text{(scope restriction)} \\
 & \mid & \mathbf{C}, \mathbf{C} & \text{(concurrent composition)} \\
 & \mid & \Lambda & \text{(the null term)} \quad \blacksquare
\end{array}
$$

There are several important conventions we will obey hereafter.

Conventions 1 *Conventions on notation and designation.*

(i) Non-capital alphabets $(a, b, c, ..)$ range over $\mathbf{N}$, the set of port names. We will often call port names as simply *names*. We assume that different alphabets denote different port names unless specified otherwise.

(ii) $\tilde{a}$, $\tilde{b}$, $\tilde{c}$ ... range over $\widetilde{\mathbf{N}}$.

(iii) X, Y, Z, ... range over $\mathbf{V}$.

(iv) P, Q, R, ... (sometimes A, B, C, ...) range over $\mathbf{C}$, which are sometimes called *configurations*. Specifically, M, M', ... ranges over the subsort of $\mathbf{C}$ which are of the form $\leftarrow \mathbf{NN}$.

(v) $\mathcal{I}$, $\mathcal{J}$, $\mathcal{K}$ etc. denote incomplete expressions.

(vi) We will assume that the constructor "," is the weakest in association, others being of the same precedence.

(vii) In $\leftarrow av$, we call a a *target*, v a *value*. In $ax.P$, we call the portion ax a *head*, P a *body*. The body expression is *guarded* by the head part. In the head, a is called a *handle* (or more descriptively *input port*), x a *carrier*. In $\{V(\tilde{x}) = yz.C\}(\tilde{v})$ and $X(\tilde{a})$, $\tilde{x}, \tilde{a}$ and $\tilde{v}$ are called *parameter parts*, and their preceding sections *main parts*. Then we say $\tilde{x}$ is a *parameter* of V etc. ■

Following these conventions, we will explain some of constructors and their intuitive meaning as follows.

Examples 1 *Meaning of constructors.*

(i) $\leftarrow av$. A message with a target a and a value v. (Note that both are port names.)

(ii) $ax.(\leftarrow ax)$. A receptor with a handle a which, when it consumes a message, creates the same one and dies. Note that the first occurrence of x binds the second x.

(iii) $\{X(x) = xy.(\leftarrow xy,\ X(x))\}(a)$. A receptor with a handle a which, when it consumes a message, creates the same one and regenerates itself. Note that the first occurrence of x is *instantiated* to a at the end, and that the x binds the later occurrences of x.

(iv) $ax.P,\ \leftarrow av,\ by.Q$. Two receptors and one message. The left receptor may consume the message.

(v) $ax.(|v| \leftarrow cv),\ \leftarrow av$. The first two occurrences of v and the third one denote different port names, because the first one is declared as private (restricted) and the third one is not within the same scope. *Restricted names are meant to have different values from those which are outside of the scope, even syntactically they are the same.* ■

Free and Bound Names, and Substitution

The following gives definitions for bound and free names, and substitutions.

Definition 2 *Free and Bound names, substitution.*

(i) In $ax.P$, a is free, x is bound, and free occurrences of x in P are bound by the carrier x. In $\leftarrow av$, both a and v are free. In $|v|P$, v is bound, and free occurrences of v in P are bound by $|v|$. In $\{X(\tilde{x}) = C\}(\tilde{a})$, names in $\tilde{a}$ are free, and names in $\tilde{x}$ binds their occurrences in C.

(ii) Similarly in $\{X(\tilde{x}) = C\}(\tilde{a})$, we define free occurrences of X in C and say those occurrences are bound by X at the top. *Hereafter we will only deal with the cases where no term variables occur free at the top configuration. We also assume that the length (as a sequence) of the parameter of a bound term variable should correspond to the parameter of the term variable which binds it.*

(iii) $\mathcal{N}(P)$ is a set of names in P. $\mathcal{FN}(P)$ (resp. $\mathcal{BN}(P)$) is a set of free (resp. bound) names in P. $\mathcal{HN}(P)$ denotes a set of names used for handles of receptors in the subexpressions of P.

(iv) We assume that, in the expression $\{X(\tilde{x})= C\}(\tilde{a})$, $\mathcal{FN}(C) \subset \{\tilde{x}\}$, and also any pair of names in $\tilde{x}$ are pairwise distinct.

(v) $P[v/x]$ denotes the result of (inductively) substituting the free occurrences of x in P for v, following the standard convention for name collision (cf. [2]).

(vi) We inductively define α-convertibility among terms starting with $ax.P$ is α-convertible to $ay.(P[y/x])$ if y is not free in P, similarly $|x|P$ is α-convertible to $|y|(P[y/x])$ with the same condition. We will assume $[x/y]$ is stronger than any other constructor (i.e. $|x|P[v/x] \stackrel{\text{def}}{=} |x|(P[v/x])$ etc.). ■

We will give some examples of substitutions.

Examples 2 *Examples of substitutions.*

(i) $(\leftarrow xv)[a/x] \equiv \leftarrow av$.

(ii) $(|v| \leftarrow xv)[v/x] \equiv |w| \leftarrow vw$. Here we first perform an α-conversion, then do the substitution. Remember restricted port names denote values *different* from those outside of the scope. ■

Now we are ready to define syntactic equivalence relation called *structural equivalence* and *reduction* (computation) rule for our formal system.

3 Structural Equivalence and Reduction

Structural Equivalence

In port passing calculi, transition rules become quite complicated because of intricate interaction between the port passing concept and scoping rules. It was found by Milner in [17], however, the introduction of congruence relation for syntactic terms, *modulo* which transition rules are defined, results in a surprisingly compact and tractable formulation. The idea is to incorporate within the structural rules tacit yet basic semantics of various constructors, freeing transition rules from expressing those static features laboriously[2]. Thus we can concentrate on truly significant aspects of computational and semantic properties of the target system.

Below is our formulation of such structural rules, which is generally based on that of [17], yet somewhat weakened to make computational aspects explicit. Notable facts are (a) the equation cannot be applied to guarded expressions (i.e. the body of receptors), so that $\equiv$ is not a congruence relation, and (b) the relation induced by (ii), (iii), and (vi)–(ix) is finite for a given term.

Definition 3 *Structural equivalence*, denoted by $\equiv$, is the smallest equivalence relation over terms defined by:

[2]Inspired by Chemical Abstract Machine [3]. It can also be likened to the separation of structural rules in Natural Deduction or to the treatment of α-conversion in [2].

(i) $P \equiv Q$ if P is α-convertible to Q
(ii) $(P, Q), R \equiv P, (Q, R)$
(iii) $|x||y|P \equiv |y||x|P$
(iv) $P, \Lambda \equiv P$
(v) $|x|\Lambda \equiv \Lambda$
(vi) $|x|P, Q \equiv |x|(P, Q) \qquad (x \notin \mathcal{FN}(Q))$
(vii) $P, Q \equiv Q, P$
(viii) $\{X(\tilde{x}) = P\}(\tilde{a}) \equiv P[\tilde{a}/\tilde{x}][\{X(\tilde{x}) = P\}/X]$.
(ix) $P \equiv Q \implies (P, R \equiv Q, R \ \wedge\ |x|P \equiv |x|Q)$

where, in (viii), $[\{X(\tilde{x}) = P\}/X]$ denotes syntactic substitution of term variables with a recursive structure[3]. ■

Note that by rule (ii), we can soundly write $P,\ Q,\ R$ (i.e. without parentheses). Similarly we will write $|xyz|P$ etc. by rule (iii). Several examples of application of structural equivalence will be helpful in its understanding.

Examples 3 *Examples for structural equivalence.* Please note that all of the multiple equations below can be one step from transitivity.

(i) With the definition of $\mathcal{I} \stackrel{\text{def}}{=} \{X(x) = xy.(\leftarrow xy, X(x))\}$ (this already appears in (iii) of Examples 1), we have, by rule (viii),

$$\mathcal{I}(a) \equiv ay.(\leftarrow ay,\ \mathcal{I}(a))$$

(ii) A message can freely *move around* (i.e. change its place in concurrent composition) due to rule (vii).

$$ax.P,\ Q,\ \leftarrow av \quad \equiv \quad ax.P,\ \leftarrow av,\ Q \quad \equiv \quad \leftarrow av,\ ax.P,\ Q\ .$$

(iii) The below abstractly states that a restricted name functions as a globally distinct name.

$$ax.(\leftarrow xv),\ |v| \leftarrow av \quad \equiv \quad ax.(\leftarrow xv),\ |z| \leftarrow az \quad \equiv \quad |z|(ax.(\leftarrow xv),\ \leftarrow az) \ ■$$

Reduction

Below we define reduction (computation) of terms. The intuitive idea is a communication event occurs when a message and a receptor with a target and a corresponding handle somehow meet together, and this is the only way for computation in the configuration to proceed (as far as we do not consider its interaction with outside).

[3] Because of the condition stated in Definition 2 (iv), the problem of name collision never occurs.

Definition 4 *Reduction of terms*, denoted by $\longrightarrow$, is the smallest relation between terms inferred by:

$$
\begin{array}{ll}
\text{COM}: & \leftarrow av,\ ax.P \ \longrightarrow\ P[v/x] \\[1ex]
\text{PAR}: & \dfrac{P_1 \longrightarrow P_1'}{P_1,\ P_2 \longrightarrow P_1',\ P_2} \\[2ex]
\text{RES}: & \dfrac{P \longrightarrow P'}{|x|P \longrightarrow |x|P'} \\[2ex]
\text{STRUCT}: & \dfrac{P_1' \equiv P_1,\ P_1 \longrightarrow P_2,\ P_2 \equiv P_2'}{P_1' \longrightarrow P_2'}
\end{array}
\qquad \blacksquare
$$

The reduction rules, together with structural rules, state basic mechanism of computation in our formal system. We will give some descriptive examples of reduction of terms.

Examples 4 *Examples of Reductions.*

(i) A simple reduction.

$$
\begin{array}{rcl}
ax.(\leftarrow cx),\ cy.(\Lambda),\ \leftarrow av & \equiv & \leftarrow av,\ ax.(\leftarrow cx),\ cy.(\Lambda) \\
& \longrightarrow & \leftarrow cv,\ cy.(\Lambda) \\
& \longrightarrow & \Lambda\ .
\end{array}
$$

(ii) With $\mathcal{I}$ as defined in (iii) of Examples 3,

$$
\begin{array}{rcl}
\leftarrow av,\ \mathcal{I}(a) & \equiv & \leftarrow av,\ ax.(\leftarrow ax, \mathcal{I}(a)) \\
& \longrightarrow & \leftarrow av,\ \mathcal{I}(a) \\
& \longrightarrow & \leftarrow av,\ \mathcal{I}(a) \\
& \longrightarrow & .....
\end{array}
$$

and so on. $\mathcal{I}(a)$ *functions as if it were nothing.*

(iii) By (vi) of Examples 3, we have:

$$
\begin{array}{rcl}
|v| \leftarrow av,\ ax.(\leftarrow xv) & \equiv & |z|(\leftarrow az,\ ax.(\leftarrow xv)) \\
& \longrightarrow & |z| \leftarrow zv
\end{array}
$$

This shows how scope opening, together with α-conversion, induces computation in the face of restriction and name collision. $\blacksquare$

In regard of functionality of the structural equivalence in its relationship to reduction relation, though the structural equivalence is somewhat weak in comparison with structural congruence in [17], it can nonetheless induce the same reduction relation. That is, if we denote the stronger equivalence by $\dot{\equiv}$ (to be formulated as $\equiv$ in [17]) and corresponding reduction rule by $\dot{\longrightarrow}$, then we have $(\dot{\longrightarrow}^*) = (\longrightarrow^* \dot{\equiv})$ where $\longrightarrow^*$ etc. means reflexive and transitive closure of the relation. We can even omit rules (iv) and (v) to get the same result. Detailed study of formulation of structural rules in combination with reduction and other transition rules is required.

4 Expressing Causality

This section introduces two important concepts for constructing *causal chains* in our purely asynchronous formal system. They are sequentialization and selection. Along the way various primitive constructs for general computation are expressed in our formal system.

Sequentialization

Our formal system is characterized with its lack of sequential constructors except when inevitable (i.e. in value passing and resulting term generation). But as we see below, this can be realized by a chain of communication events, sequentialization of value passing in this case.

Definition 5 (a) *Notations for sequentialization.* Suppose E is a term expression. We define *sequential connectives* $\lhd$ (of type $\mathbf{N}^2 \times \widetilde{\mathbf{N}}$) and $\rhd$ (of type $\mathbf{N}^2 \times \widetilde{\mathbf{N}} \times \mathbf{C}$) as follows.

$$\begin{cases} ax \lhd \varepsilon & \stackrel{\text{def}}{=} & \Lambda \\ ax \lhd v\tilde{w} & \stackrel{\text{def}}{=} & ax.(\leftarrow xv,\ ax \lhd \tilde{w}) \end{cases}$$

$$\begin{cases} \leftarrow ax \rhd \varepsilon.E & \stackrel{\text{def}}{=} & E \\ \leftarrow ax \rhd y\tilde{w}.E & \stackrel{\text{def}}{=} & \leftarrow ax,\ xy. \leftarrow ax \rhd \tilde{w}.E \end{cases} \quad \blacksquare$$

Based on these connectives, we define the following expressions.

Definition 5 (b) *Notations for communication of a series of names.* We define $\leftarrow a : \tilde{v}$ and $a : \tilde{x}.E$ as follows. We suppose that r, c are not free in E and $\tilde{v}$, respectively.

$$\begin{aligned} \leftarrow a : \tilde{v} & \stackrel{\text{def}}{=} |c|(\leftarrow ac,\ cx \lhd \tilde{v}) \\ a : \tilde{x}.E & \stackrel{\text{def}}{=} az.|r|(\leftarrow zr \rhd \tilde{x}.E) \quad \blacksquare \end{aligned}$$

Examples 5 *Sequentialization of communication.* With the condition that $r, c \notin \mathcal{FN}(P) \cup \{v_1, v_2\}$,

$$\begin{aligned} \leftarrow a : v_1v_2,\ a : y_1y_2.P & \equiv |c|(\leftarrow ac,\ \ cx \lhd v_1v_2),\ \ az.|r|(\leftarrow zr \rhd y_1y_2.P) \\ & \longrightarrow |cr|(cx \lhd v_1v_2,\ \ \leftarrow cr \rhd y_1y_2.P) \\ & \longrightarrow |cr|(\leftarrow rv_1,\ cx \lhd v_2,\ \ ry_1. \leftarrow cr \rhd y_2.P) \\ & \longrightarrow |cr|(cx \lhd v_2,\ \ \leftarrow cr \rhd y_2.(P[v_1/y_1])) \\ & \longrightarrow |cr|(\leftarrow rv_2,\ \ ry_2.(E[v_1/y_1])) \\ & \longrightarrow |cr|P[v_1/y_1][v_2/y_2] \\ & \equiv P[v_1/y_1][v_2/y_2] \quad \blacksquare \end{aligned}$$

Thus two values v_1 and v_2 are passed respectively to y_1 and y_2, preserving their order. Because communication is taking place solely using private ports, no interference from the third party is possible after the first reduction. In a sense, c and r are functioning as *private communication channels* between P and Q. For any $\tilde{v}$ and $\tilde{y}$ with the same length[4], it is easy to verify the below.

$$\leftarrow a:\tilde{v},\ a:\tilde{y}.P \ \longrightarrow^* \ P[\tilde{v}/\tilde{y}] \quad .$$

Another example uses these sequentialization features nontrivially, showing the mapping of our formal system to its superset calculus presented in [17].

Examples 6 *Encoding for the extended calculus.* We replace expressions $\leftarrow av$ and $\{X(\tilde{x})= C\}(\tilde{a})$ with $\bar{a}v.P$ and $!P$ respectively, and assume a structural rule

$$!P \ \equiv \ P,\ !P$$

and a reduction rule

$$\bar{a}v.P,\ a(x).Q \ \longrightarrow \ P,\ Q[v/x] \ .$$

Then a mapping from the expressions in the extended system to the reduced system, written as $[\![\cdot]\!]$, is given as follows.

$$\begin{array}{rcl}
[\![\bar{a}v.P]\!] & = & |c|(\leftarrow a:vc,\ c:\varepsilon.[\![P]\!]) \\
[\![ax.P]\!] & = & a:xy.(\leftarrow y:\varepsilon,\ [\![P]\!]) \\
[\![!P]\!] & = & |c|(\leftarrow c:\varepsilon,\ \{X(x)= x:\varepsilon.([\![P]\!],\ \leftarrow c:\varepsilon,\ X(x))\}(c)) \qquad (c \notin \mathcal{FN}(P)) \\
[\![|x|P]\!] & = & |x|[\![P]\!] \\
[\![P,Q]\!] & = & [\![P]\!],[\![Q]\!] \\
[\![\Lambda]\!] & = & \Lambda\ . \qquad \blacksquare
\end{array}$$

The key idea of the coding is to let the *receiver* of a message send the activation message as a reply to the sender, so that the subsequent behaviour of the sender (which is coded as another receptor, $c:\varepsilon.[\![P]\!]$) can become active. We do not verify the correctness of this mapping in this paper, which can be done by saying that if there is a reduction in the world of superset expressions then the corresponding reduction does exist in our coding, and that if a term in our coding reduces to something then it has some further reduction which corresponds to some reduction in the domain of the superset.

Selections

A more advanced way of constructing a causal chain can be achieved through the use of *selections*. This is especially important for us because the formal system has no summation. We only deal with binary selection but it can be extended with ease.

[4]This constraint is not essential since a little change in Definition 5 (b) results in capability of coping with cases where two lengths can be different, by using new port generation.

Definition 6 *Notations for selection.* Suppose E, E_1, E_2 are term expressions and $i = 1$ or $i = 2$. We define *connectives for selection* of type $\mathbf{N}^4 \times \mathbf{C}$ and of type $\mathbf{N}^3 \times \mathbf{C}^2$ as follows.

$$\begin{array}{rcl} x : y_1 y_2 \lhd_i v : E & \stackrel{\text{def}}{=} & x : y_1 y_2 . (\leftarrow y_i : v,\ E) \\ \leftarrow x : y_1 y_2 \rhd_1 (z_1 . E_1) \rhd_2 (z_2 . E_2) & \stackrel{\text{def}}{=} & \leftarrow x : y_1 y_2,\ \ y_1 : z_1 . E_1,\ \ y_2 : z_2 . E_2 \quad \blacksquare \end{array}$$

The idea is for the first one to selectively send a message ($\leftarrow y_i v$) and generate a term (E), and the second one to send the options ($\leftarrow x : y_1 y_2$) and wait for activation ($y_1 : z_1 . E_1,\ y_2 : z_2 . E_2$).

To safely use these connectives, we again rely on new port name generation. The encoding for natural numbers and the successor function are given below.

Examples 7 *Natural numbers and the successor function.*

$$\begin{array}{rcl} 0(n) & \stackrel{\text{def}}{=} & \{X(x) = (x : y_1 y_2 \lhd_1 x : X(x))\}(n) \\ N'(n) & \stackrel{\text{def}}{=} & |p|(\{X(xz) = (x : y_1 y_2 \lhd_2 z : X(xz))\}(np),\ N(p)) \\ \mathcal{S}(s) & \stackrel{\text{def}}{=} & s : nc.|z|(\{X(xn) = (x : y_1 y_2 \lhd_2 n : X(xp)\}(n), \leftarrow cz)) \quad \blacksquare \end{array}$$

Note that a natural number is expressed as an *object* which knows its predecessor (p in above), in contrast to the expression of a natural number as a *function* in λ-calculus. Thus even "0" is defined recursively, which is necessary because its handle will be passed around among its "users". c in the successor stands for a *customer* [1], the target of the reply. The predecessor and judgment of zero should *decode* these data structures (when a formal parameter is not necessary, we will omit it).

Examples 8 *The predecessor and judge-if-zero functions.*

$$\begin{array}{rcl} \mathcal{P}(s) & \stackrel{\text{def}}{=} & s : nc.|y_1 y_2|(\leftarrow n : y_1 y_2 \rhd_1 (p.(\leftarrow c : p)) \rhd_2 (p.(\leftarrow c : p))) \\ \mathcal{J}(jtf) & \stackrel{\text{def}}{=} & j : nc.|y_1 y_2|(\leftarrow n : y_1 y_2 \rhd_1 (\leftarrow c : t) \rhd_2 (\leftarrow c : f)),\ \ 0(t),\ \ 0'(f) \quad \blacksquare \end{array}$$

Here *true* and *false* are expressed as 0 and 1 respectively.

The next example shows more advanced branching structures.

Examples 9 *If and Parallel Or.*

$$\begin{array}{rcl} \mathcal{C}(i) & \stackrel{\text{def}}{=} & i : bp_1 p_2 . |y_1 y_2|(\leftarrow b : y_1 y_2 \rhd_1 (\leftarrow p_1 : \varepsilon) \rhd_2 (\leftarrow p_2 : \varepsilon)) \\ \mathcal{O}(o) & \stackrel{\text{def}}{=} & o : b_1 b_2 c . |s_1 ... s_6| \\ & & (\leftarrow b_1 : s_3 s_4 \rhd_1 (\leftarrow s_1 : b_1) \rhd_2 (\leftarrow s_2 : b_1), \\ & & \leftarrow b_2 : s_5 s_6 \rhd_1 (\leftarrow s_1 : b_2) \rhd_2 (\leftarrow s_2 : b_2), \\ & & s_1 : x . (\leftarrow c : x,\ s_1 : \varepsilon . \Lambda), \\ & & s_2 : x . s_2 : x .\ \leftarrow c : x) \quad \blacksquare \end{array}$$

The combination of conditional expressions can easily construct "and", sequential "or", and "not", so we omit them here. The "parallel or" above uses the method similar to the one by Nierstrasz in [20], using a synchronizer to invoke only one action out of multiple candidates. This method is directly usable to realize the parallel case construct. Primitives for selection can also be used for *method invocation* in usual object-orientation. It can be proved that we can construct any computable functions on natural numbers by combination of the constructs we have encoded and the use of recursively defined receptors.

Finally we show a very simple stateful entity called a *cell*. It is primitive yet indeed possesses typical properties of concurrent objects as we know. Its first option is "read", and the other option is "write". It contains some port name as its state. It gets o as its option (representing 0 or 1), and then decodes it to take an action accordingly. w is used as a value to write, but when the option is "read", w is just neglected. Note that how it regenerates itself, with or without change of its state according to the option.

Examples 10 *A cell.*

$$\mathcal{L}(lv) \stackrel{\mathrm{def}}{=} \{X(xy) = x : owc.(|y_1 y_2| \leftarrow o : y_1 y_2 \rhd_1 (\leftarrow c : y,\ X(xy)) \rhd_2 (\ X(xw)))\}(lv)$$

This small concurrent object concludes this section, and we proceed to see a bit of the semantic framework of our formalism.

5 Semantics

This section gives several basic definitions for our semantic framework based on asynchronous interaction, and discusses its notable theoretical properties informally.

Asynchronous Interaction

Our semantic framework is based on the notion of observation by *asynchronous experiments*. This means that an experimenter just sends asynchronous messages to the concerned system, and (possibly continuing to send further messages) wait for output messages from the configuration. Thus it does not matter whether or not a message the experimenter sends is actually consumed by some receptors in the configuration. This notion of *asynchronous interaction* can be given its formal representation as a labeled transition system.

The below shows a set of labels we will use for our labeled transition system.

Definition 7 *Labels.* The sets of labels for interaction $\mathbf{L}$ and of their series $\widetilde{\mathbf{L}}$ are given by the following abstract syntax.

$$\mathbf{L} \ = \ \tau \mid \downarrow\mathbf{NN} \mid \uparrow\mathbf{NN} \mid \uparrow\mathbf{N}|\mathbf{N}| \qquad \blacksquare$$

The above labels have the following intuitive meanings.

(1) τ denotes the internal computation (unseen from the outside), that is, the same thing as *reduction* (Definition 4).

(2) $\downarrow av$ means that the configuration asynchronously gets a message $\leftarrow av$ from outside. Seen differently, this rule tells us that the experimenter sends a message to the configuration.

(3) $\uparrow av$ means the configuration asynchronously emits a message or the experimenter receives such a message.

(4) $\uparrow a|v|$ means sending a value of a name restricted inside the configuration, corresponding to scope opening in structural rules (Definition 3 (vi)). For an experimenter, this means that he acquires a piece of new information which he has not had until then.

Conventions 2 *Notation.*

(i) We will let $l, l', \ldots$ range over $\mathbf{L}$,

(ii) We denote $\mathcal{FN}(l)$ to be a set of port names in l except in the case $l = \uparrow a|v|$, then $\mathcal{FN}(l) = \{a\}$. Similarly $\mathcal{BN}(l) = \phi$ except $\mathcal{BN}(\uparrow a|v|) = \{v\}$. $\mathcal{N}(l)$ is the union of these two. ■

Based on these definitions and conventions, we define the *interaction relation* as follows. It is a triple of (P, l, P'), which is written as $P \xrightarrow{l} P'$.

Definition 8 *Interaction of terms*, denoted by $\xrightarrow{l}$, is the smallest relation inferred by:

$$\text{IN}: \quad \Lambda \xrightarrow{\downarrow av} \leftarrow av$$

$$\text{OUT}: \quad \leftarrow av \xrightarrow{\uparrow av} \Lambda$$

$$\text{COM}: \quad \leftarrow av,\ ax.P \xrightarrow{\tau} P[v/x]$$

$$\text{PAR}: \quad \frac{P_1 \xrightarrow{l} P_1'}{P_1,\ P_2 \xrightarrow{l} P_1',\ P_2} \qquad (\mathcal{BN}(l) \notin \mathcal{FN}(P_2))$$

$$\text{RES}: \quad \frac{P \xrightarrow{l} P'}{|x|P \xrightarrow{l} |x|P'} \qquad (x \notin \mathcal{N}(l))$$

$$\text{OPEN}: \quad \frac{P \xrightarrow{\uparrow ax} P'}{|x|P \xrightarrow{\uparrow a|x|} P'} \qquad (a \neq x)$$

$$\text{STRUCT}: \quad \frac{P_1' \equiv P_1,\ P_1 \xrightarrow{l} P_2,\ P_2 \equiv P_2'}{P_1' \xrightarrow{l} P_2'} \qquad ■$$

Intuitively, these rules define behaviour of a configuration in terms of its interaction with the outside as asynchronous exchange of messages between them. In this regard the essential rule which is directly related with asynchronous character of the semantics, is the first IN rule. Indeed this is the only rule which differentiates this semantic definition from Milner's one in [17], yet which results in surprisingly different semantic properties. For the purpose of comparison, we stipulate the synchronous counterpart of our semantics, which is a reformulated version of Milner's one.

Definition 9 *Synchronous interaction of terms*, denoted by $\xrightarrow{l}_s$, is the smallest relation inferred by the same rules as Definition 8, with $\xrightarrow{l}$ replaced by $\xrightarrow{l}_s$ except IN rule which is reformulated as

$$\mathrm{IN}_s: \quad ax.P \xrightarrow{\downarrow av}_s P[v/x] \qquad \blacksquare$$

A few remarks on Definition 8 are due here.

(1) The rule clearly shows that $\xrightarrow{\tau} = \longrightarrow$.

(2) Note the symmetry between IN and OUT rules in Definition 8. This is destroyed by introduction of IN_s rule. Also note that the corresponding forms of IN and OUT in π-calculus also enjoy a symmetry of their own [17]. This implies the naturalness of synchronous semantics for π-calculus and asynchronous one for our system.

(3) One interesting aspect of interaction rules lies in OPEN rule, which denotes that if one configuration emits a private label to outside, it is regarded as free (i.e. public) from then on. This reminds us of Agha's notion of "adding receptionists by communication to outside" in the context of the actor model [1].

(4) It may seem rather extraordinary that because of IN rule in Definition 8, *any* message can come into the configuration, regardless of the forms of inner receptors. But this is perfectly consistent with our intuitive notion of asynchronous experiments. As the experimenter is not synchronously interacting with the configuration (which means he should own corresponding input/output port names), he may send any message as he likes. Moreover it does not result in difficulties in proving various semantic properties as far as we know.

Asynchronous Bisimulation

As we noted already, from the experimenter's point of view, IN rule states that the experimenter sends some message to the concerned configuration and OUT rule states he receives some message from the configuration. This recaptures Milner's notion of experiments (cf. [13]) in the setting of asynchronous communication. Below we define (weak) bisimulation, or observation equivalence, as a semantic representation of this new notion of experiments. While simulation preorder should be regarded as somewhat more fundamental than the equivalence, within this elementary exposition we confine ourselves to bisimulation.

Definition 10 *Asynchronous bisimulation.* Let us define $\xRightarrow{\hat{l}}$ as $\xrightarrow{\tau}^* \xrightarrow{l} \xrightarrow{\tau}^*$ if $l \neq \tau$ and if else as $\xrightarrow{\tau}^*$. Then P_1 and Q_1 are asynchronously bisimilar, denoted by $P_1 \approx_a Q_1$ if and only if $(P_1, Q_1) \in \mathcal{R}$ where for any $(P, Q) \in \mathcal{R}$ we have

(i) Whenever $P \xrightarrow{l} P'$, for some Q', $Q \xRightarrow{\hat{l}} Q'$ and $(P', Q') \in \mathcal{R}$.

(ii) $\mathcal{R}$ is symmetric. ■

For comparison again, we define its synchronous counterpart.

Definition 11 *Synchronous bisimulation.* Let us define $\stackrel{\hat{l}}{\Longrightarrow}_s$ as $\stackrel{\tau}{\longrightarrow}_s{}^* \stackrel{l}{\longrightarrow}_s \stackrel{\tau}{\longrightarrow}_s{}^*$ if $l \neq \tau$ and if else as $\stackrel{\tau}{\longrightarrow}_s{}^*$. Then P_1 and Q_1 are synchronously bisimilar, denoted by $P_1 \approx_s Q_1$ if and only if $(P_1, Q_1) \in \mathcal{R}$ where for any $(P, Q) \in \mathcal{R}$ we have

(i) Whenever $P \stackrel{l}{\longrightarrow}_s P'$, for some Q', $Q \stackrel{\hat{l}}{\Longrightarrow}_s Q'$ and $(P', Q') \in \mathcal{R}$.

(ii) $\mathcal{R}$ is symmetric. ■

Note that Definition 11 is simpler than the corresponding one in [15], which needs an additional condition for equivalence after substitution of names. This may come from the formulation of their IN rule as shown below.

$$ax.P \stackrel{\downarrow ax}{\longrightarrow}_s P$$

The rule means that the received name should *not* be the same as any free names in P. We do not discuss this point further except pointing out that the following reduction (not interaction) is allowed both in our formal system and (in the corresponding form) in π-calculus. We hope that this will provide an argument for our formulation of IN rule.

$$\leftarrow av,\ ax.(\leftarrow xc,\ vy.P) \quad \longrightarrow \quad \leftarrow vc,\ vy.P \quad .$$

A few examples will be helpful to understand how asynchronous bisimulation works.

Examples 11 *Asynchronous bisimulation (1).*

(i) *Replication.* Let us assume a new notation (cf. Examples 6).

$$!P \stackrel{\mathrm{def}}{=} |c|(\leftarrow c{:}\varepsilon,\ \{X(x) = x{:}\varepsilon.(P,\ \leftarrow c{:}\varepsilon,\ X(x))\}(c)) \qquad (c \notin \mathcal{FN}(P)) \ .$$

Then the following holds.

$$!P \quad \approx_a \quad P,\, !P \quad .$$

To verify, take a relation $(\ (!P,\ R),\ \ (P,\ !P,\ R)\ )$ where R can be an arbitrary term expressions. This is an example where both $\approx_a$ and $\approx_s$ hold.

(ii) *The successor function.* Using notation in Examples 7,

$$|sz|(0(z),\ \mathcal{S}(s),\ \leftarrow s : zc) \quad \approx_a \quad |x|(1(x), \leftarrow cx)$$

(to check, just compute). Again we see both $\approx_a$ and $\approx_s$ hold.

(iii) *Permutation in input.* P and Q are given as follows.

$$P \stackrel{\text{def}}{=} ax.(by.R) \qquad Q \stackrel{\text{def}}{=} by.(ax.R)$$

Then we have both $P \not\approx_s Q$ and $P \not\approx_a Q$. The former obviously holds and the latter can be differentiated by

$$P \stackrel{\downarrow av}{\longrightarrow} \leftarrow av,\ ax.(by.R) \stackrel{\tau}{\longrightarrow} by.R[v/x]$$

but

$$Q \stackrel{\downarrow av}{\longrightarrow} \leftarrow av,\ by.(ax.R) \stackrel{\varepsilon}{\Longrightarrow} \leftarrow av,\ by.(ax.R) \stackrel{\uparrow av}{\longrightarrow} by.ax.R\ .$$

Please note that the only difference comes from the message which comes in and just goes out, while it is possible for it to get consumed in one transition[5]. ■

None of the above examples show any difference between two bisimulations. As the order of sending messages generally cannot matter in asynchronous communication, the next example may seem rather promising.

$$P \stackrel{\text{def}}{=} |l|(\leftarrow lz,\ lz.(\leftarrow av,\ |m|(\leftarrow mz,\ mz.\leftarrow bw)))$$

and

$$Q \stackrel{\text{def}}{=} |l|(\leftarrow lz,\ lz.(\leftarrow bw,\ |m|(\leftarrow mz,\ mz.\leftarrow av)))$$

Here we have $P \approx_a Q$ as expected, providing an interesting comparison with the expressions in π-calculus, $\bar{a}.\bar{b}.\Lambda$ and $\bar{b}.\bar{a}.\Lambda$. However the example does not distinguish $\approx_a$ and $\approx_s$, because $P \approx_s Q$ holds. Is there any case where one can differentiate between these two equivalence theories? The next example shows that such a case does exist.

Examples 12 *Asynchronous bisimulation (2).* Let us remember the expression $\mathcal{I}$ in Section 3 (Example 3 (iii) and Example 4 (ii)). For this special agent, the following holds for any a .

$$\mathcal{I}(a) \approx_a \Lambda$$

To verify, make a relation $\mathcal{R} = (\ (\mathcal{I}(a), P),\ P\ )$, where P is zero or more messages without bound names.

(i) Firstly, if

$$\mathcal{I}(a), P \stackrel{\downarrow av}{\longrightarrow} \mathcal{I}(a), P, \leftarrow av\ .$$

then clearly

$$P \stackrel{\downarrow av}{\longrightarrow} P, \leftarrow av\ .$$

where $(\ (\mathcal{I}(a), P, \leftarrow av),\ (P, \leftarrow av)\ ) \in \mathcal{R}$. We can similarly verify in the case

$$\mathcal{I}(a), P \stackrel{\uparrow av}{\longrightarrow} \mathcal{I}(a), P'\ .$$

[5]This shows that transition relation as formulated in Definition 8 lacks the notion of *locality*.

(ii) Next if

$$\mathcal{I}(a), P \xrightarrow{\tau} Q \ .$$

then the only possibility is there is some P' such that

$$P \equiv P', \leftarrow av$$

but then

$$\mathcal{I}(a), \leftarrow av, P' \xrightarrow{\tau} \mathcal{I}(a), \leftarrow av, P' \equiv \mathcal{I}(a), P$$

As obviously $P \stackrel{\varepsilon}{\Longrightarrow} P$, this case holds.

As the symmetric case holds trivially, $\mathcal{R} \cup \mathcal{R}^{-1}$ is bisimulation, and just by taking $P \stackrel{\text{def}}{=} \Lambda$, the argument holds. ■

This example is notable in two respects.

(1) Because we have $\approx_s \subset \approx_a$ (the proof is not so difficult), the above example shows that this inclusion is strict.

(2) Another fact is that $\approx_a$ is a congruence relation (as well as $\approx_s$) in our system (both proofs are rather long). Thus the example shows that the term $\mathcal{I}(a)$ or any term which is bisimilar to it can be added or deleted from a configuration arbitrarily without changing its meaning. Based on this fact, there is a method to construct $\approx_s$ from $\approx_a$ by adding appropriate $\mathcal{I}(x)$'s to configurations. This suggests the exact range of difference between $\approx_a$ and $\approx_s$.

The difference between $\approx_a$ and $\approx_s$ is important in that it suggests asynchronous interaction (the relation "$\xrightarrow{l}$") is more *abstract* than synchronous one ("$\xrightarrow{l}_s$") in the sense that it does not care the order of consecutive inputs or consecutive outputs. Hence we will deal with *collections* (to be exact, multisets) of messages rather than their sequences. This gives rise to an elegant mathematical treatment of asynchronous interaction semantics, and the property can be directly reflected in our equivalence theory if we add a certain locality notion. Then we have $ax.by.P \approx'_a by.ax.P$ (cf. Example 11 (iii)). We leave the further details to the subsequent exposition to be published elsewhere in the near future.

6 Conclusion

We have seen so far that a formal system based on the notion of pure asynchronous communication can be constructed with full expressive power and important semantic properties. The investigation of the concurrency formalism based on asynchronous communication has just begun, and there are many problems to be solved. Other than the study on asynchronous interaction semantics and its relationship with objects notion, two important points should be pointed out.

(1) We should study whether the construction (or reduction) we performed in this exposition can be applied to CCS or other process calculi formalisms. Especially we should study what results one will obtain for higher-order process calculi which passes processes [21, 4].

(2) The most important possibility of our formal construction in the pragmatic context may exist in sound formulation of the notion of *types* for concurrent object-based computing. There is an interesting work in this direction by Nierstrasz [19]. We hope that the study of asynchronous semantics will provide us with suggestions for typed programming for concurrent objects.

Finally the authors would like to thank Carl Hewitt, who stayed in Keio University from Autumn 1989 to Summer 1990, for beneficial discussions with them, to Professor Joseph Goguen for his suggestions, to Vasco Vasconcelous for discussions and comments on the paper, to Chisato Numaoka for discussions on concurrency, to Kaoru Yoshida for her stimulating e-mails, and to all the labo members for their kind assistance and cheers.

References

[1] Agha, G.: *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] Barendreght, H. : *The Lambda Calculus: Its Syntax and Semantics.* North Holland, 1984.

[3] Berry, G. and Boudol, G. : The Chemical Abstract Machine. In *Proc. 17 the Annual Symposium on Principles of Programming Languages*, 1990.

[4] Boudol, G. : Towards a Lambda-Calculus for Concurrent and Communicating Systems. In *Proc. TAPSOFT 1989*, LNCS 351, Springer-Verlag, 1984.

[5] Clinger, W. : *Foundations of Actor Semantics.* AI-TR-633, MIT Artificial Intelligence Laboratory.

[6] Goguen, J., *Sheaf semantics for concurrent interacting objects.* To appear in Proc. REX School on Foundations of Object-Oriented Programming, Noorwijkerhout, The Netherlands, May 28-June1, 1990.

[7] Hewitt, C. : *Viewing Control Structures as Patterns of Passing Messages.* Artificial Intelligence, 1977.

[8] Hewitt, C., Bishop, P., and Steiger, R. : A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proc. of the 3rd International Joint Conference on Artificial Intelligence,* August 1973.

[9] Hoare, C.A.R. : *Communicatin Sequential Processes.* Prentice Hall, 1985.

[10] Honda, K. and Tokoro, M., An Object Calculus for Asynchronous Communication, *Proc. of European Conference on Object-Oriented Programming*, LNCS, Springer-Verlag, July 1991.

[11] Honda, K. and Tokoro, M., On Asynchronous Communication Semantics, *Object-based Concurrent Computing*, ed. M. Tokoro, LNCS, Springer-Verlag, 1992.

[12] Meseguer J., *Conditional Rewriting Logic as a Unified Model of Concurrency.* SRI-CSL-91-05, Computer Science Laboratory, SRI International, 1991. Also to appear in *Theoretical Computer Science.*

[13] Milner, R. : *Calculus of Communicating Systems.* LNCS 92, Springer-Verlag, 1980.

[14] Milner, R. : Calculi for Synchrony and Asynchrony. *Theoretical Computer Science 25*, 1983.

[15] Milner, R., Parrow, J.G. and Walker, D.J. : *A Calculus of Mobile Processes. Part I and II.* ECS-LFCS-89-85/86, Edinburgh University, 1989

[16] Milner, R. : *Communication and Concurrency.* Prentice Hall, 1989.

[17] Milner, R. : Functions as Processes. In *Automata, Language and Programming*, LNCS 443, 1990. The extended version under the same title as Rapports de Recherche No.1154, INRIA-Sophia Antipolis, February 1990.)

[18] Nielson and Engberg : *A Calculus of Communicating Systems with Label Passing.* Research Report DAIMI PB-208, Computer Science Department, University of Aarhus, 1986.

[19] Nierstrasz, O. : *Towards a Type Theory for Active Objects.* in [25].

[20] Nierstrasz, O., *A Guide to Specifying Concurrent Behaviour with Abacus.* in [25].

[21] Thomsen, B. : A calculus of higher order communicating systems. In *Proc. 16 the Annual Symposium on Principles of Programming Languages*, 1989.

[22] Tokoro, M. : Computational Field Model: Toward a New Computing Model/Methodology for Open Distributed Environment. In *Proc. of The 2nd IEEE Workshop on Future Trends in Distributed Computing Systems*, Cairo, 1990.

[23] Tokoro, M. and Honda, K. : Computational Field Model for Open Distributed Environment. To appear in Yonezawa, A., McColl, W., and Ito, T., (ed.), *Concurrency: Theory, Language, Architecture*, LNCS, Springer Verlag, 1991.

[24] Tokoro, M. : Towards Computing Systems in '2000. LNCS, Springer Verlag, 1991. *(the title to be filled later)*

[25] Tsichritzis, D. (ed.) : *Object Management.* Centre Universitaire D'informatique, Universite de Geneve, July 1990.

[26] Yonezawa, A., and Tokoro, M., (ed.) : *Object-Oriented Concurrent Programming.* MIT Press, 1986.