Asynchronous Communication Model Based on Linear Logic

Naoki Kobayashi Akinori Yonezawa
Department of Information Science
the University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113 Japan
{koba, yonezawa}@is.s.u-tokyo.ac.jp
TEL +81-3-5800-6913
FAX +81-3-5689-4365

July 1992

Abstract

We propose a new framework called ACL for concurrent computation based on linear logic. ACL is a kind of *linear logic programming* framework, where its operational semantics is described in terms of *proof construction* in linear logic. We also give a model-theoretic semantics as a natural extension of *phase semantics*, a model of linear logic. Our framework well captures concurrent computation based on asynchronous communication. It will, therefore, provide us with a new insight into other models of concurrent computation from a *logical* point of view. We also expect ACL to become a formal framework for verification, reasoning, and transformation of concurrent programs by the use of techniques for traditional logic programming. ACL's attractive features for concurrent programming paradigms are also discussed.

1 Introduction

For future massively parallel processing environments, concurrent programming languages based on *asynchronous communication* would become more and more important. Due to the difficulty of writing and debugging programs in such environments, computers would need to aid programmers for transforming and verifying of concurrent programs, hence the role of formal frameworks for concurrent computation would be significant. Recently, several applications of Girard's linear logic[12] to logic programming were proposed and shown that they correspond to reactive paradigms[4][5][17].

We propose a new framework called ACL (Asynchronous Communication based on Linear logic) for concurrent computation along this line. Computation in ACL is described in terms of *proof construction* in linear logic. We restrict the inference rules and formulas in linear sequent calculus so that the restricted rules have a proof power equivalent to the original rules for the restricted formulas. The resulting computational framework contains rich mechanisms for concurrent computation, such as message-passing style asynchronous communication, identifier creation, and hiding operator. They are all described in a *pure logical* form. We also give a model-theoretic semantics as a natural extension of *phase semantics*, a model of linear logic, by using a popular fixpoint construction. ACL inference rules can be proven to be sound and complete w.r.t. this model-theoretic semantics. Our framework well captures concurrent computation based on asynchronous communication. It

will, therefore, provide us a new insight into other models of concurrent computation from a *logical* point of view. In fact, the actor model[2] and asynchronous CCS[18][19] can be directly translated into our ACL framework. We also expect ACL to become a formal framework for verification, reasoning, and transformation of concurrent programs with techniques used in traditional logic programming. ACL also exhibits attractive features as a concurrent programming paradigm, subsuming the actor computation[2] and providing mechanisms for waiting multiple messages, sharing information, inheritance, etc.

The contributions of this paper are: (1)to give a formal foundation for linear logic programming, (2)to relate not a full but large set of classical linear logic (including connectives $(\otimes, \parr, \oplus, \&, \multimap, (\ )^{\perp}, ?, \forall, \exists)$) to intuitionistic meaning of computation, (3)to give a logical point of view to other frameworks of concurrent programming languages including object-oriented concurrent programming[26] and (4)to introduce new concepts on concurrent programming.

The rest of this paper is organized as follows. Section 2 describes the syntax and operational semantics of the basic fragment of ACL. Section 3 gives a model-theoretic semantics and proves the soundness and completeness theorems. In Section 4, we extend ACL to include mechanisms for value passing, identifier creation, and hiding. Section 4 also gives an extension where processes can be consumed as resources, and also discusses the use of incomplete models as active types of concurrent processes. Section 5 shows the translation from the actor model and asynchronous CCS into ACL. Section 6 summarizes features of ACL as concurrent programming paradigms. Section 7 compares ACL to the previous work. Section 8 concludes this paper.

2 ACL Framework

In this section, we introduce the basic (propositional) fragment of ACL. We give transition rules in a form of restricted inference rules of linear sequent calculus.

2.1 Program Syntax

First, we define the *ACL program clause*.

Definition 1 *A program is a set of clauses, which are defined as follows:*

Clause ::= Head $\circ\!\!-$ Body
Head ::= A_P
Body ::= Statement | Choice
Choice ::= Guarded_Statement| Choice $\oplus$ Guarded_Statement
Guarded_Statement ::= Guard $\otimes$ Statement
Guard ::= $A_m^{\perp}$ | Guard $\otimes$ $A_m^{\perp}$
Statement ::= $\top$ | $\perp$ | A_P | A_m | Body $\parr$ Body
| Body & Body | $?A_m$
A_P ::= $P, Q, R, \ldots$ (process predicates)
A_m ::= $m, n, \ldots$ (message predicates)

Example. A buffer process with one place can be defined in ACL as follows:

$$EmptyBuffer \circ\!\!- put^{\perp} \otimes FullBuffer$$

$$FullBuffer \circ\!\!- get^{\perp} \otimes (reply \parr EmptyBuffer)$$

This definition is quite similar to the following description in CCS[19],

$$EmptyBuffer = put.FullBuffer$$

$$FullBuffer = get.\overline{reply}.EmptyBuffer$$

though there is a significant difference that communication in ACL is *asynchronous* as is described below, whereas it is synchronous in CCS.

2.2 Operational Semantics

Transition rules are given as a restricted form of inference rules in linear sequent calculus. Please note that the rules should be read that the *conclusion* of an inference rule transits to its *premise formula*. For instance, rule (C2) should be read as $\oplus_j(m_j^{\perp} \otimes A_j), m_i, \Gamma \longrightarrow A_i, \Gamma$.

ACL Inference rules are given as follows:

- Structural Rules

  (S1) $\frac{\vdash \Delta}{\vdash \Gamma}(\Delta \text{ is a permutation of } \Gamma) \cdots$(Exchange)

  (S2) $\frac{\vdash B, \Gamma}{\vdash A, \Gamma}(A \text{ is a logically equivalent formula to } B) \cdots$(Logical Equivalence)

- Parallel

  (P1) $\frac{\vdash A, B, \Gamma}{\vdash A \parr B, \Gamma} \cdots$(parallel)

  (P2) $\frac{\vdash A, \Gamma \qquad \vdash B, \Gamma}{\vdash A \,\&\, B, \Gamma} \cdots$(fork)

  $A \parr B$ is a usual parallel composition of A and B, whereas $A \,\&\, B$ is a process which copies the entire environment and executes A and B independently, which is a similar operation to Unix[1] fork.

- Communication Rules

  (C1) $\frac{\vdash m, B, \Gamma}{\vdash m \parr B, \Gamma} \cdots$(message send(equivalent to parallel rule))

  (C2) $\frac{(\vdash m_i, m_i^{\perp}) \qquad \vdash A_i, \Gamma}{\vdash \oplus_j(m_j^{\perp} \otimes A_j), m_i, \Gamma} \cdots$(normal message reception)

  where $\oplus_j F_j$ is an abbreviated form of $F_1 \oplus \ldots \oplus F_k$

  (C3) $\frac{(\vdash m_i, m_i^{\perp}) \qquad \vdash A_i, ?m_i, \Gamma}{\vdash \oplus_j(m_j^{\perp} \otimes A_j), ?m_i, \Gamma} \cdots$(modal message reception)

- Termination Rules

  (T1) $\frac{}{\vdash \top, A} \cdots$(program termination)

  (T2) $\frac{\vdash A}{\vdash \perp, A} \cdots$(suicide)

- Clause Rule

  (Cl1) $\frac{\vdash B, \Gamma}{\vdash A, \Gamma}$(if there is a clause: $A \circ\!\!- B) \cdots$(process unfolding)

[1] Unix is a registered trademark of AT&T

- Context Rule

  (Co1) $\dfrac{\vdash C[B], \Gamma}{\vdash C[A], \Gamma}$(if $\dfrac{\vdash B}{\vdash A}$ is derived from the other rules)

  C[ ], called a *positive context*, is defined as follows:

  $$C[\,] ::= [\,] \mid C[\,] \parr F \mid F \parr C[\,] \mid C[\,] \,\&\, F \mid F \,\&\, C[\,] \mid C[\,] \otimes F \mid F \otimes C[\,] \mid C[\,] \oplus F \mid F \oplus C[\,]$$

  where F is any formula of linear logic.

Rules (C1)-(C3) are rules for communication. $m \parr A$ in rule (C1) represents a *sender process* which sends message m. This operation is asynchronous, because $\vdash m, A$ and $\vdash m \parr A$ are logically equivalent in linear logic. $\oplus_j(m_j^{\perp} \otimes A_j)$ in rule (C2) represents a *receiver process* which waits any one of messages $m_1, \ldots, m_k$ and becomes A_i when receiving m_i. This corresponds to the *input guard* in CSP [7]. $\vdash m_i, m_i^{\perp}$ in rule (C2) is an axiom, hence a proof tree ends at this leaf. Computation, therefore, goes to the direction of up-right. $?m$, which we call a *modal message*, is a message which can be copied unboundly, hence may be used several times by several processes.

$\top$ in the rule (T1) is a terminator which terminates a whole program, whereas $\perp$ in (T2) terminates only itself. Context rule (Co1) may be redundant, but is included to exploit *intra process parallelism*.

The following proposition states that the above inference rules have a proof power equivalent to the original inference rules in linear sequent calculus for the restricted formula.

Proposition 1 (Equivalence between ACL rules and linear sequent calculus) *Let P be a program and A be a body formula. $\vdash A$ is provable by the above inference rules if and only if $\vdash ?P^{\perp}, A$ is provable in linear sequent calculus.*

The proof of this proposition is given in Appendix.

3 Model based on Phase Semantics[2]

In this section, we give a model for the body formulas defined in the previous section by extending the *phase semantics*[12] of linear logic. The resulting model is compositional because of the compositionality of the original phase model.

3.1 Preliminary — Phase Semantics by Girard

Before we give our model for ACL, we review the original definition of phase semantics[12] by Girard. Phase model P is given as a triple $(M, \perp, a_M)$, where M is a commutative monoid, $\perp$, called *orthogonal phases*, is a subset of M, and a_M is an assignment of *fact* for each propositional letter (facts are subsets G of M such that $G^{\perp\perp} = G$).

For each formula F, its model F^* is defined as follows:

$a^* = a_M$ (if a is a propositional letter)

$(F^{\perp})^* = (F^*)^{\perp} \stackrel{\text{def}}{=} \{p \in M \mid \forall q(q \in F^* \rightarrow pq \in \perp)\}$

[2]The readers who are interested in more computational aspects of ACL can skip this section.

$(F \parr G)^* = F^* \parr G^* \stackrel{\text{def}}{=} (F^{*\perp} \times G^{*\perp})^\perp$
$(F \otimes G)^* = (F^* \times G^*)^{\perp\perp}$
$(F \& G)^* = F^* \cap G^*$
$(F \oplus G)^* = (F^* \cup G^*)^{\perp\perp}$
where $F \times G = \{pq \mid p \in F \wedge q \in G\}$

3.2 Model for ACL

A set of program clauses is written in the form of

$$< P, Q, R, \ldots > \circ\!\!- \vec{F}(< P, Q, R, \ldots >),$$

where F is a monotonic function on phase space, which is composed of projection, product, and connectives of linear logic ($\parr, \&, \oplus, \otimes, ?$).

Given a phase model $(M, \top, m_M)$ where m_M is an assignment of facts to message predicates, we assign minimal facts to process predicate so that clauses be valid. To make clauses be valid in phase semantics, it is necessary and sufficient that the following condition holds:

$$\vec{1} \in < P^*, Q^*, R^*, \ldots > \circ\!\!- \vec{F}^*(< P^*, Q^*, R^*, \ldots >)$$

$$\Longleftrightarrow < P^*, Q^*, R^*, \ldots > \supset \vec{F}^*(< P^*, Q^*, R^*, \ldots >)$$

Therefore, we define the model $P^*, Q^*, R^*, \ldots$ of $P, Q, R, \ldots$ by the following equation:

$$< P^*, Q^*, R^*, \ldots > = \bigoplus_{n \in \omega} (\vec{F}^*)^n(\vec{\mathbf{0}^*})$$

We can prove that the ACL inference rules are sound and complete w.r.t. this model.

Proposition 2 (Soundness) *The ACL inference rules are sound w.r.t. the extended phase model in the following sense: Let G be a body formula. If G is provable, then G is valid (i.e., $1 \in G^*$) in all the extended phase models.*

Before we prove this proposition, we introduce some definitions.

Definition 2 (Substitution) *We define substitution on formulas, $\sigma = [X_1/Y_1, \ldots, X_n/Y_n]$ as follows:*

1. *$a\sigma = a$ where a is an atomic predicate.*
2. *$X_i\sigma = Y_i$*
3. *$(FopG)\sigma = F\sigma op G\sigma$ where op is $\parr, \&, \otimes, \oplus$.*
4. *$F^\perp\sigma = (F\sigma)^\perp$*
5. *$(?F)\sigma = ?(F\sigma)$*
6. *$(!F)\sigma = !(F\sigma)$*

Proof of Proposition 2 Suppose that G is provable by ACL inference rules. By lemma 1, we have a proof by inference rules in linear sequent calculus and ACL inference rule (Cl1).

Let $A_1, \ldots, A_n$ be process predicates, and clauses be

$$A_1 \circ\!\!- F_1(A_1, \ldots, A_n)$$
$$\ldots$$
$$A_n \circ\!\!- F_n(A_1, \ldots, A_n)$$

We define $A_i^{(k)}$ by

$$A_i^{(0)} = \mathbf{0}$$
$$A_i^{(k+1)} = F_i(A_1^{(k)}, \ldots, A_n^{(k)})$$

Then

$$A_i^{(0)*} \subseteq A_i^{(1)*} \subseteq A_i^{(2)*} \subseteq \ldots$$

holds, because F_is are monotonic(w.r.t. set inclusion) functions on the phase space. Therefore, it is sufficient to show that

$$1 \in G[A_1/A_1^{(m)}, \ldots, A_n/A_n^{(m)}]^*$$

for some m.

We prove it by induction on the length of the proof.

1. Length=0
   The conclusion G is either $\top \parr \Gamma$ or $A \parr A^{\perp}$. Then G is valid from the soundness of linear sequent calculus (See [12]) for any assignment of facts to $A_1, \ldots, A_n$. Therefore, it holds that
   $$1 \in G[A_1/A_1^{(0)}, \ldots, A_n/A_n^{(0)}]^*$$

2. Length=k+1 ($k \geq 0$)
   Let G' be a premise formula of the last inference. By the induction hypothesis,
   $$1 \in G'[A_1/A_1^{(m)}, \ldots, A_n/A_n^{(m)}]^*$$
   holds for some m. If the last inference rule is unfolding rule on A_i, G contains a formula A_i, and it holds that
   $$G\{A_i/F_i(A_1, \ldots, A_n)\} = G'$$
   where $G\{A/B\}$ is the formula whose one occurrence of A is replaced by B. Hence it holds that
   $$\begin{aligned} 1 \in\ & G[A_1/A_1^{(m)}, \ldots, A_{i-1}/A_{i-1}^{(m)}, A_i/A_i^{(m+1)}, A_{i+1}^{(m)}, \ldots, A_n/A_n^{(m)}]^* \\ \in\ & G[A_1/A_1^{(m+1)}, \ldots, A_n/A_n^{(m+1)}]^* \end{aligned}$$
   In the other cases,
   $$1 \in G' \Rightarrow 1 \in G$$
   holds from the soundness of linear sequent calculus.

□

Proposition 3 (Completeness) *ACL inference rules are complete w.r.t. the extended phase model in the following sense: Let G be a body formula. If G is valid (i.e., $1 \in G^*$) in all the extended phase models, G is provable by ACL inference rules.*

Proof We can assume that G is an atomic formula A_i because otherwise we can add the following clause to the program clauses:

$$A_{n+1} \circ\!\!- G$$

where A_{n+1} is a new process predicate. From the definition,

$$A_i^* = \bigoplus_{k \in \omega} (A_i^{(k)*})$$

Suppose that A_i^* is valid in all the extended phase model. By Lemma 5, it is sufficient to show that A_i is provable by inference rules in linear sequent calculus and ACL clause rules and context rules.

$$1 \in A_i^{(m)*}$$

holds for some m, because

$$A_i^{(0)*} \subseteq A_i^{(1)*} \subseteq A_i^{(2)*} \subseteq \ldots$$

From the completeness of linear sequent calculus[12], $A_i^{(m)}$ is provable in linear sequent calculus.

Let σ be a substitution: $[A_1/F_1(A_1, \ldots, A_n), \ldots, A_n/F_n(A_1, \ldots, A_n)]$, and $B = A_i\sigma^m$. Then

$$B[A_1/\mathbf{0}, \ldots, A_n/\mathbf{0}] = A_i^{(m)}$$

holds. Because there is no inference rule for $\mathbf{0}$, B is also provable by inference rules in linear sequent calculus. By applying clause rules and context rules to B several times, we can deduce A_i from B, hence A_i is provable by inference rules in linear sequent calculus and ACL clause rules and context rules. □

The following proposition states that $\bigoplus_{n\in\omega}(\vec{F}^*)^n(\mathbf{0}^*)$ is the minimal facts which make clauses valid.

Proposition 4 (Fixpoint) $\bigoplus_{n\in\omega}(\vec{F}^*)^n(\mathbf{0}^*)$ *is the least fixpoint of* $\vec{F}^*$.

Proof Suppose that x is a fixpoint of $\vec{F}^*$. Because $\mathbf{0}^*$ is a minimal fact,

$$x \supseteq \vec{\mathbf{0}^*}$$

then,

$$(\vec{F}^*)^n(x) = x \supseteq (\vec{F}^*)^n(\vec{\mathbf{0}^*}) \text{ for all n}$$
$$(\vec{F}^*)^n(x) = x \supseteq \bigoplus_{n\in\omega}(\vec{F}^*)^n(\vec{\mathbf{0}^*})$$

Therefore, it is sufficient to show that $A^\omega = \bigoplus_{n\in\omega}(\vec{F}^*)^n(\vec{\mathbf{0}^*})$ is a fixpoint.

$F(A^\omega) \supseteq A^\omega$ holds from the monotonicity of F.

Let A^m be $\bigoplus_{n\leq m}(\vec{F}^*)^n(\vec{\mathbf{0}^*}) = (\vec{F}^*)^m(\vec{\mathbf{0}^*})$. Then it holds that

$$A^0 \subseteq A^1 \subseteq A^2 \subseteq \ldots \subseteq A^\omega$$

hence

$$\vec{F}^*(A^0) \subseteq \vec{F}^*(A^1) \subseteq \vec{F}^*(A^2) \subseteq \ldots \subseteq \vec{F}^*(A^\omega)$$

holds. Therefore,

$$
\begin{array}{ll}
 & x \in \vec{F}^*(A^\omega) \\
\Rightarrow & x \in \vec{F}^*(A^m) \text{ for some m} \\
\Rightarrow & x \in (\vec{F}^*)^{m+1}(\vec{\mathbf{0}}^*) \text{ for some m} \\
\Rightarrow & x \in A^\omega
\end{array}
$$

Then $\vec{F}^*(A^\omega) \subseteq A^\omega$ holds, hence A^ω is a fixpoint of $\vec{F}^*$. $\Box$

3.3 Specific Model

We can choose the following triple $(M, \bot, m_M)$ for a specific model[12]. M is a monoid whose underlying set is a set of multisets of formulas composed of message predicates, and whose operation is a concatenation on multisets. $\bot$ and m_M are given by:

$$
\begin{array}{l}
\bot = \{\Gamma \mid\vdash \Gamma \text{ is provable in linear sequent calculus}\} \subseteq M \\
m_M = \{\Gamma \mid\vdash \Gamma, m \text{ is provable in linear sequent calculus}\}
\end{array}
$$

Then for process predicates A_is, it holds that

$$
A_i^* = \{\Gamma \mid\vdash \Gamma, A_i \text{ is provable in linear sequent calculus with ACL clause rule (Cl1)}\}
$$

Unfortunately,

$$
A_i^* = \{\Gamma \mid\vdash \Gamma, A_i \text{ is provable by ACL inference rules}\}
$$

does not hold, because A_i^* may contain non-body formula. The following condition, however, holds:

$$
\Gamma \in A_i^* \text{ and } \Gamma \text{ is an ACL body formula } \Leftrightarrow \vdash \Gamma, A_i \text{ is provable by ACL inference rules}
$$

A_i^*, therefore, gives a set of testers which succeed when running with A_i in parallel, hence this specific model derives an equivalence similar to *testing equivalence*[22]. From the another point of view, A_i^* represents 'what interaction A_i performs with its environment', because A_i^* contains only message predicates. For example, $\{m^\bot \otimes \top\} \in A_i^*$ implies that $\vdash A_i, m^\bot \otimes \top$ is provable, hence A_i creates a message m or is a terminator($\top$).

4 Extensions of ACL

This section gives first-order and second-order extensions of ACL. These extensions can enjoy rich mechanisms for concurrent computation. We also gives another extension where processes are stratified so that processes are treated as *resources* and can be consumed as ordinary messages. Section 4.4 discusses the use of incomplete models as *types* of concurrent processes.

4.1 First Order Extension

ACL can be extended to include first order formulas. Existential quantification and universal quantification provide mechanisms for value passing and identifier creations, respectively.

4.1.1 First-Order Existential Quantification for Value Passing

We introduce first-order existential quantification to the receiver part of a message. Then, communication rules (C2)-(C3) are modified as follows:

$$\textbf{(C2}') \ \frac{(\vdash m_i(a), m_i(a)^\perp) \qquad \vdash A_i(a), \Gamma}{\vdash \oplus_j \exists X(m_j(X)^\perp \otimes A_j(X)), m_i(a), \Gamma}$$

$$\textbf{(C3}') \ \frac{(\vdash m_i(a), m_i(a)^\perp) \qquad \vdash A_i(a), ?m_i(a), \Gamma}{\vdash \oplus_j \exists X(m_j(X)^\perp \otimes A_j(X)), ?m_i(a), \Gamma}$$

The rule (C2) corresponds to the following inference in linear sequent calculus:

$$\dfrac{\dfrac{\dfrac{(\vdash m_i(a), m_i(a)^\perp) \qquad \vdash A_i(a), \Gamma}{\vdash m_i(a)^\perp \otimes A_i(a), m_i(a), \Gamma}\otimes}{\vdash \exists X(m_i(X)^\perp \otimes A_i(X)), m_i(a), \Gamma}\exists}{\vdash \oplus_j \exists X(m_j(X)^\perp \otimes A_j(X)), m_i(a), \Gamma}\oplus$$

The formula $\exists X(m(X)^\perp \otimes P(X))$ represents a process which waits for the values of X via m, and becomes $P(a)$ after receiving message $m(a)$. This extension allows processes to send values in messages.

Arithmetic operations (for example) can be included in ACL by providing rewriting systems on first order terms. We consider *convergent* term rewriting system[9] R, and add the following inference rule:

- Rewriting Rule

  (R1) $\dfrac{\vdash p(t), \Gamma}{\vdash p(s), \Gamma}$ if $s \to t \in R$ where 'a' in Rule (C2′) is restricted to the normal form in R.

Generally, R can be any programming language, including functional languages and imperative languages, whose output is uniquely determined. Then ACL with a rewriting system R can be considered as an extension of R to a concurrent language with asynchronous communication facilities.

4.1.2 First-Order Universal Quantification for Identifier Creation

First-order universal quantification works as a mechanism for *identifier creation*. Identifier creation is often very important in concurrent computing[2][20][25][26]. Identifiers work as *pointers* to access resources including processes such that resources can be accessed only by *acquaintances*, i.e., processes which know their pointers. By passing identifiers in messages, acquaintances can be dynamically changed. In the actor model[2], an identifier is used as a mail address which is unique to each actor.

Let us look at $\forall$-rule in linear sequent calculus:

$$\frac{\vdash A(X), \Gamma}{\vdash \forall X.A(X), \Gamma} \ \text{X not free in } \Gamma$$

We modify this rule as

- Identifier Rule

(ID1) $\dfrac{\vdash A(id), \Gamma}{\vdash \forall X.A(X), \Gamma}$

where id is a unique identifier which does not appear in $A(X)$ and Γ.

It is trivial that the modified rule is equivalent to the original rule. This rule corresponds exactly to an identifier creation.

4.1.3 Model for First-Order Extension

We briefly overview the model for the first order extension in the analogy of *Herbrand model* in logic programming[15]. Herbrand model is given as a mapping from Herbrand base (a set of ground atomic formulas) to the truth values $\{true, false\}$. In ACL model, each ground atomic formula is mapped to a *fact* in phase model, instead of $\{true, false\}$. Then, the fixpoint construction given in section 3 corresponds to the construction of the *least Herbrand model* in logic programming.

4.2 Second-Order Universal Quantification as Hiding Operator

In the ACL defined section 2, all messages are visible to all processes. To organize a large program, we need some form of encapsulation mechanism. In this section, we introduce second-order universal quantification for message formulas. It works as a *hiding operator* as in CCS. To see how it works, let us look at $\forall$-rule in linear sequent calculus:

$$\frac{\vdash A, \Gamma}{\vdash \bigwedge X.A, \Gamma} \text{ X not free in } \Gamma$$

We use the symbol $\bigwedge$, instead of $\forall$ to distinguish from the first-order universal quantification. Notice the side condition. Quantified variable cannot be free outside the scope of $\bigwedge$. It is, therefore, invisible from outside. We introduce the following ACL rules instead of the above original rules in linear sequent calculus.

- Hiding Rules

  (H1) $\dfrac{\vdash \bigwedge m.(A, n), \Gamma}{\vdash \bigwedge m.(A), n, \Gamma}$ where n contains neither m nor process predicates.

  (H2) $\dfrac{\vdash \bigwedge m.(A), n, \Gamma}{\vdash \bigwedge m.(A, n), \Gamma}$

  (H3) $\dfrac{\vdash \Gamma}{\vdash \bigwedge m.(), \Gamma}$

$C[\,]$ in context rules are also extended to include the form $\bigwedge m.(C[\,])$. Then, again this extension can be proven to be equivalent to linear sequent calculus. This equivalence is due to the lack of the rule for second-order existential quantification. If we introduce the second-order existential quantification, they are not equivalent any more. We, therefore, must use the original rules. The introduction of second-order existential quantification allows us to pass names of messages, hence may lead to the asynchronous version of Milner's π-calculus[20][21], though we do not investigate it here.

4.3 Stratified Process — Processes as Resources

Up to now, we distinguished messages and processes completely and forbidden from *receiving processes* (process predicates were not allowed to appear in the guard part). In this subsection, we stratify processes so that processes can consume processes in the lower classes as if they were ordinary messages. For example, $P^{\perp} \otimes Q$ represents a process which consumes a process P, and becomes a new process Q. Then, messages are nothing but processes in the lowest class.

We modify the ACL program syntax as follows:

$$
\begin{aligned}
\text{Clause}_i &::= \text{Head}_i \circ\!\!- \text{Body}_i \\
\text{Head}_i &::= A_{Pi} \\
\text{Guard}_i &::= A^{\perp}_{Pj} \mid \text{Guard}_i \otimes A^{\perp}_{Pj}\ (j < i) \\
\text{Statement}_i &::= A_{Pk} \mid \top \mid \bot \mid \text{Body}_i \parr \text{Body}_i \\
&\quad \mid \text{Body}_i \,\&\, \text{Body}_i \mid ?A_{Pj}\ (j < i, k \leq i)
\end{aligned}
$$

Intuitively, a process in A_{Pi} can produce and consume processes in $A_{P0}(= A_m), \ldots, A_{Pi-1}$. By using this mechanism, we can change the structure of concurrent processes *dynamically*, which would be effective for processes to adapt to dynamical changes of environments such as the number of processors, and communication networks.

Extension to stratified processes provides the same effect as allowing multiple atoms[4][5] in a head to some extent. In fact,

$$A \circ\!\!- B^{\perp} \otimes C$$

is logically equivalent to

$$A \parr B \circ\!\!- C.$$

4.4 Incomplete Models as Types

The specific phase model given in section 3.3 is complete w.r.t. linear sequent calculus. We can also consider *incomplete* phase models. Then its extensions give *non-standard semantics*, a kind of *types of concurrent processes*.

For example, let us consider an abstract predicate m' for each message predicate m, whose argument is a name of type such as INT and REAL. Then we assign a fact to each message predicate as follows:

$$p(x)^* = \{\Gamma \mid\!\vdash \Gamma, p'(t) \text{ is provable, type of } x \text{ is } t \text{ and } \Gamma \text{ is composed of abstract predicates}\}$$

In this model, $p(x)$ and $p(y)$ are identified as far as types of x and y are the same.

Then, if a process A is defined as

$$A \circ\!\!- \exists X(p(X)^{\perp} \otimes q(X+1))$$

the model of A is given as follows:

$$A^* = \{p'(INT) \parr q'(INT)^{\perp} \otimes \top, \ldots\}$$

This model means that a process A accepts an integer via p, then returns an integer via q.

5 Translations of Actors, CCS and π-calculus

In this section, we show that the actor model[3] and asynchronous CCS[18] can be directly translated into the ACL framework.

5.1 Actor

In the actor model[3], computation is performed by concurrent agents, called *actors*, communicating each other by message passing. Upon receiving a message, an actor can send new messages to other actors, create new actors, and become itself a new behavior. An actor can be represented in ACL as follows:

$$Actor(id) \circ\!\!- \bigoplus_i (m_i(id)^\perp \otimes (m_{i1} \parr \ldots \parr m_{ij} \\ \parr \forall id_1 \ldots \forall id_k (Newactor_1(id_1) \parr \ldots \parr Newactor_k(id_k) \\ \parr Newbehabivor(id))))$$

The above description means that upon receiving message m_i, *Actor* with its mail address *id* sends new messages $m_{i1}, \ldots, m_{ij}$, creates new actors $Newactor_1, \ldots, Newactor_k$ and becomes a new behavior with the same *id*. $\forall$ is used to create new identifiers as mail addresses of created actors. Therefore, actor model can be directly translated into the first order extension of ACL.

5.2 CCS

An asynchronous version of CCS is translated into ACL in the following way:

$$\begin{aligned} Tr(A) &= A \text{ (if } A \text{ is a constant)} \\ Tr(P+Q) &= Tr(P) \oplus Tr(Q) \\ Tr(a.P) &= a^\perp \otimes Tr(P) \\ Tr(\overline{a}.P) &= a \parr Tr(P) \\ Tr(P \mid Q) &= Tr(P) \parr Tr(Q) \\ Tr(P \backslash l) &= \bigwedge l.Tr(P) \\ Tr(A \stackrel{\text{def}}{=} P) &= A \circ\!\!- Tr(P) \end{aligned}$$

where Tr is the translation function.

The asynchronous CCS, therefore, corresponds to the basic fragment of ACL + second order universal quantifiers.

5.3 π-calculus $-$ CCS $\approx$ 2nd Order Existential Quantifier?

An asynchronous version of Milner's π-calculus[20][21] can be encoded in ACL by using second order quantifications.

We encode a sender process $\overline{y}x.P$ and a receiver process $y(x).P$ of π-calculus as follows:

$$Tr(\overline{y}x.P) = ((y \parr a) \otimes (x \parr b)) \parr Tr(P)$$
$$Tr(y(x).P) = \bigvee x(((y \parr a) \otimes (x \parr b))^\perp \otimes Tr(P))$$

where $\bigvee$ is the second order existential quantifier and a and b are special predicates such that $a \neq b$.

Communication rules are:

$$\frac{\vdash (y \parr a) \otimes (x \parr b), P, \Gamma}{\vdash ((y \parr a) \otimes (x \parr b)) \parr P, \Gamma}$$

$$\frac{\vdash P\{x/z\}, \Gamma}{\vdash (y \parr a) \otimes (x \parr b), \bigvee z(((y \parr a) \otimes (z \parr b))^{\perp} \otimes P), \Gamma}$$

where $P\{x/z\}$ is the substitution of z for all free occurrences of x in P. Note that the restriction $(x)P$ is translated into $\bigwedge x.Tr(P)$.

Recall that in the previous subsection asynchronous CCS was encoded by using only second order universal quantifier. Our translations into ACL, therefore, indicate that the use of second order existential quantifier distinguishes π-calculus from CCS.

6 ACL as a Concurrent Programming Paradigm

ACL provides attractive features as a concurrent programming paradigm. In this section, we summarize such characteristics of ACL.

6.1 Waiting Multiple Messages

A process which waits multiple messages $m_1, \ldots, m_n$ and becomes P is represented as:

$$m_1^{\perp} \otimes \ldots \otimes m_n^{\perp} \otimes P$$

It can receive messages $m_1, \ldots, m_n$ in *arbitrary order*, because $\otimes$ is associative and commutative, hence the following logical equivalence holds:

$$m_1^{\perp} \otimes \ldots \otimes m_n^{\perp} \otimes P = m_i^{\perp} \otimes m_1^{\perp} \otimes \ldots \otimes m_{i-1}^{\perp} \otimes m_{i+1}^{\perp} \otimes \ldots \otimes m_n^{\perp} \otimes P$$

This mechanism is useful for synchronization between multiple processes and for splitting large data into multiple messages. It is provided in concurrent logic programming[24] by a head unification mechanism, whereas it is not provided in the actor model[2], CCS[19] and CSP[7].

Example 1. A process $sync(i)^{\perp} \otimes sync(j)^{\perp} \otimes (go(i) \parr go(j))$, which receives messages $sync(i)$ and $sync(j)$ and send messages $go(i)$ and $go(j)$, works as a synchronizer between two processes. Suppose that $P(i)$ and $P(j)$ want to synchronize each other. $P(i)$ (resp. $P(j)$) only has to send a message $sync(i)$ (resp. $sync(j)$) and wait a message go.

Example 2. Let $Sum(i, j, radr)$ be a process that computes the summation of integers from i to j and replies the answer to $radr$. Sum can be defined as follows:

$$\begin{aligned} Sum(i,j,radr) \circ\!\!- \ & \underline{\text{if}}\ i = j \\ & \underline{\text{then}}\ ans(radr, i) \\ & \underline{\text{else}}\ \forall adr1 \forall adr2 (Sum(i, \lfloor (i+j)/2 \rfloor, adr1) \parr Sum(\lfloor (i+j)/2 \rfloor + 1, j, adr2) \\ & \parr \exists x \exists y\ ans(adr1, x)^{\perp} \otimes ans(adr2, y)^{\perp} \otimes ans(radr, x+y)) \end{aligned}$$

if X then Y else Z is an abbreviated form of $(ifTrue(X) \otimes Y) \oplus (ifFalse(X) \otimes Z)$, where $ifTrue$(resp. $ifFalse$) is a special predicate that maps truth values $true$ to $\mathbf{1}$(resp. $\mathbf{0}$) and $false$ to $\mathbf{0}$(resp. $\mathbf{1}$).

Sum in the above definition creates two child processes and makes them compute the halves of the sum, then gather the results and add them. $Ans(adr1, x)$ and $ans(adr2, y)$, the results of child processes, can be received in any order.

6.2 Partial Reception of a Message

The ACL context rule allows a receiver of a message to start computation immediately after receiving a part of a message. This mechanism enables us to exploit full parallelism between senders and receivers and makes turn-around time (the time from starting to send a message until receiving the reply) to be minimal, which would be effective when large data need to be sent.

Example. Let P be a process: $\exists X1 \exists Y1 \ldots \exists Xn \exists Yn\ x(1, X1)^{\perp} \otimes y(1, Y1)^{\perp} \otimes \ldots \otimes x(n, Xn)^{\perp} \otimes ans(X1 * Y1 + \ldots + Xn * Yn)$. P receives two vectors $\vec{x} = (X1, \ldots, Xn)$ and $\vec{y} = (Y1, \ldots, Yn)$, and computes the inner product of them. The context rule and rewriting rule allows P to start computation before having received the whole data of $\vec{x}$ and $\vec{y}$. Note that the calculation of $X1 * Y1 + \ldots + Xn * Yn$ can be started from anywhere owing to the rewriting rule $(R1)$ in section 4.1.1.

6.3 Message destination address is not necessarily needed

Message passing in ACL is based on pattern matching. Messages, therefore, do not necessarily have their destination addresses. The lack of destination address may make the communication slow compared with object-oriented concurrent languages such as ABCL[25], where messages always have their destination addresses. The advantage is its flexibility. The sender of a message need not know exactly who is the receiver. This is effective, for instance, in the client-server models where the server is a pool of multiple processes. A client only throws a request into the pool of server processes, instead of specifying a specific server. Then one of idle servers picks up the request and replies to the client.

This feature differentiates messages in ACL from those in object-oriented concurrent languages such as ABCL. Intuitively, 'to send messages' in ACL is 'to throw messages into the sea of processes' rather than 'to send messages to specific processes.'

6.3.1 Should process have its unique identifier?

In the actor model[2], each actor has its *unique* identifier and it works as the mail address of an actor. Should processes always have their unique identifiers and should messages contain identifiers of receivers as in actor? It sometimes makes difficult to change structures of processes. For example, if we decompose a process P into two processes P_1 and P_2, processes that send messages to this process P also must be changed accordingly, because the messages need to be received by P_1 or P_2 instead of P. In ACL, the sender part need not be changed, because a message need not to have the identifier of receiver process. In general, ACL allows multiple views of processes. If several processes have the same mail address, these processes can be viewed as if they were one process from outside. As is described in the following

subsection, this mechanism with the hiding operator provides hierarchical construction of processes, where each process is composed of multiple subprocesses.

6.4 Hierarchical construction of processes — Encapsulation by hiding operator

Encapsulation is an important mechanism for programming languages to reduce bugs of programs by forbidding invalid access to resources, and to increase the readability of programs. The hiding operator explained in section 4.2 provides a kind of *encapsulation mechanism* by making some messages invisible from outside. A group of processes encapsulated by the hiding operator is viewed as one large process because the internal computation is invisible. By using the hiding operator hierarchically, processes can be constructed hierarchically, i.e., a large process is composed of several subprocesses, and such a subprocess is again composed of its smaller subprocesses.

6.5 Modal messages for sharing information

The property that contraction and weakening are not allowed in linear logic assures that every normal message is consumed by exactly one process. Modal message $?m$, however, can be consumed by several processes and several times because weakening and contraction are allowed for them. They can be used to broadcast and share information among multiple processes.

Modal messages are useful for DP(Dynamic Programming). Andreoli and Pareschi[5] have pointed out that *forum-based communication*, the communication mechanism similar to our modal message, was useful for DP.

Example : Fibonacci Let us define a clause:

$$F(n) \circ\!\!- \exists X \exists Y (fib(n,X)^{\perp} \otimes fib(n+1,Y)^{\perp} \otimes (?fib(n+2,X+Y) \parr F(n+1))$$

$fib(n,m)$ means that the nth element of the fibonacci sequence is m. Process $F(n)$ reads the nth and $(n+1)$th elements (they are not deleted after read operations because they are modal messages) and creates the $(n+2)$th element. If we execute the following goal:

$$\vdash F(1), ?fib(1,1), ?fib(2,1)$$

then $?fib(3,2), ?fib(4,3), \ldots$ are generated.

6.6 Inheritance

The extended ACL where processes are stratified can provide a kind of inheritance mechanism in the same sense as in LO[4]. Let A be a process which is already defined. If we want to define a new process by adding a new functionality to A, all we have to do is to define the following clause:

$B \circ\!\!- A^{\perp} \otimes (\ldots)$ (which is equivalent to: $A \parr B \circ\!\!- \ldots$)

where B is the part of a new process differentiating itself from A.

Example. Let us define a counter process by

$$Counter(n) \circ\!\!- \exists radr(inc(radr)^{\perp} \otimes (ack(radr) \parr Counter(n+1))) \\ \oplus \exists radr(read(radr)^{\perp} \otimes (reply(n, radr) \parr Counter(n)))$$

$Counter$ process has two methods inc and $read$. Then we can define a new process which has another method $reset$ by

$$Counter2 \circ\!\!- \exists radr \exists n(reset(radr)^{\perp} \otimes Counter(n)^{\perp} \otimes (ack(radr) \parr Counter(0) \parr Counter2))$$

Process $Counter2 \parr Counter(n)$ now has three methods inc, $read$, and $reset$.

6.7 Dynamic Restructuring of Processes

Computing environments such as a group of available computational resources may change while programs are running. Concurrent processes should dynamically adjust themselves to such change of their environment, because an optimal configurations of processes in a certain environment may not be optimal in another environment. We can define a process that dynamically composes or decomposes other processes by using multiple atoms in a head. By using this mechanism, processes can be dynamically restructured and their granularity can be adjusted.

For example, let A, B, C be processes and $A \parr B$ do the same task as C. If we define the following clauses:

$$C \parr D \circ\!\!- decompose^{\perp} \otimes (A \parr B \parr D)$$

$$A \parr B \parr D \circ\!\!- compose^{\perp} \otimes (C \parr D)$$

then, D is a process that decomposes C into A and B when receiving message 'decompose', and composes A and B into C when receiving message 'compose.'

Example 1 Let Sum be a process defined in section 6.1. Let us define process $Seqsum(i, j, sum, radr)$, which computes the summation of integers from i to j sequentially, as follows:

$$Seqsum(i, j, sum, radr) \circ\!\!- \underline{\text{if}}\ i = j\ \underline{\text{then}}\ ans(radr, sum{+}i)\ \underline{\text{else}}\ Seqsum(i{+}1, j, sum{+}i, radr).$$

We can define processes which exchange $Seqsum$ and Sum dynamically as follows:

$$Seq_to_par \circ\!\!- \exists i \exists j \exists sum \exists radr(Seqsum(i, j, sum, radr)^{\perp} \\ \otimes \forall adr(Sum(i, j, adr) \parr \exists x(ans(adr, x)^{\perp} \otimes ans(radr, x + sum))))$$
$$Par_to_seq \circ\!\!- \exists i \exists j \exists radr(Sum(i, j, radr)^{\perp} \otimes Seqsum(i, j, 0, radr))$$

Seq_to_par changes $Seqsum$ (a process which computes the sum sequentially) to Sum (a process which computes in parallel). Par_to_seq is converse.

Example 2 Let $Filter(from, to, n)$ be a filtering process which receives message $m(from, i)$ and sends message $m(to, i)$ only if i is a multiple of n. This is defined as follows:

$$Filter(from, to, n) \circ\!\!- \exists i(m(from, i)^{\perp} \otimes (\ \underline{\text{if}}\ (i\ mod\ n) = 0 \\ \underline{\text{then}}\ m(to, i) \parr Filter(from, to, n) \\ \underline{\text{else}}\ Filter(from, to, n)))$$

We can compose $Filter(from, x, k)$ and $Filter(x, to, l)$ into $Filter(from, to, lcm(k, l))$. A process which does this dynamically can be defined as follows:

$$Composer \circ\!\!- \exists from \exists x \exists to \exists k \exists l (Filter(from, x, k)^{\perp} \otimes Filter(x, to, l)^{\perp} \otimes (Filter(from, to, lcm(k, l))))$$

where $lcm(m, n)$ is a function to compute a least common multiple of m and n.

7 Related Work

Girard's linear logic[12] has been drawing great attentions in recent years. There are two major approaches to modelling concurrent computation by using linear logic. One is the approach from functional programming[1][14] which is based on *'Formulae as Types'* notion of Curry-Howard Isomorphism[13] where computation is described in terms of proof normalization. The other is *'Formulae as States, Proofs as Computations'* approach[16]. In this approach, connections between Petri Nets and linear logic have been investigated[16][11][10][6]. They relate Petri Nets to theories in linear logic using only $\otimes$. Later, [6] extended this approach to the implication($\multimap$). The latter approach, *'Proofs as Computations'*, is also investigated in a rather different way, in the context of logic programming[4][5][17] and concurrent constraint programming[8]. Andreoli and Pareschi[4][5] pointed out that 'Proof search as computation' analogy for linear logic corresponds to a reactive paradigm. Our ACL follows this line and gives a connection between concurrent computation based on asynchronous communication and a large set of linear logic including connectives $(\otimes, \parr, \oplus, \&, \multimap, (\ )^{\perp}, ?, \forall, \exists)$.

Communication in the language *LO*, proposed in [5], is based on a kind of broadcast mechanism, called *forum-based communication*. It is very powerful and shown to be effective in applications such as dynamic programming. Unfortunately, the model-theoretic semantics for ACL cannot be applied to *LO*, because *LO* is based on an *extra-logical* operator, called *tell marker*. The communication mechanism in our ACL is based on *pure logical* operators and is yet powerful. Modal messages in ACL can provide the similar effect as forum-based communication in *LO* as discussed in section 6.5. The advantage of our pure logical approach is that the properties of processes are treated uniformly by the logical semantics.

Miller[17] describes the connection between π-calculus and linear logic. The translation in [17] uses *non-logical* constants for synchronous communication. He also introduced *co-agent* and showed that they can specify some testing equivalences for a subset of CCS. The specific model in our ACL (in section 3.3) provides a similar model to co-agent in the more general setting of computational framework, including value passing, hiding, and identifier creation.

The concurrent definitional constraint programming(DCP) proposed in [8] is formulated in the style of concurrent constraint programming[23], and the logical semantics is given by linear logic. Our model theoretic semantics would also be applied to concurrent DCP, although in DCP A transits to B if A entails B while in ACL A transits to B if A is entailed by B.

8 Conclusion

We have proposed a logical framework ACL for concurrent programming languages based on linear logic. We gave an operational semantics of ACL by restricting inference rules in linear sequent calculus, and also gave a model theoretic semantics by extending phase semantics. A specific instance of our model derives a testing equivalence. In ACL, message passing style communication, identifier creation, and hiding operator are formulated pure logically, hence these mechanisms are uniformly treated by the logical semantics. ACL also offers a novel

concurrent programming paradigm, providing mechanisms for modal messages, compositional processes, dynamic restructuring of processes, etc. Our future work includes the application of techniques for traditional logic programming to transformation, reasoning and verification of concurrent programs written in ACL.

Acknowledgement We would like to thank Mitsuhiro Okada and Satoshi Matsuoka for their helpful comments and suggestions. We are also thankful to Jean-Marc Andreoli for the discussions during his stay in Japan.

References

[1] Abramsky, S., "Computational Interpretations of Linear Logic," *Theoretical Computer Science(to appear)*, 1993.

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

[3] Agha, G., "The Structure and Semantics of Actor Language," in *Proceedings of the School/Workshop on Foundations of Object-Oriented Languages*, Springer Verlag, 1991.

[4] Andreoli, J.-M., and R. Pareschi, "Linear Objects: Logical processes with built-in inheritance," *New Generation Computing*, 1991.

[5] Andreoli, J.-M., and R. Pareschi, "Communication as Fair Distribution of Knowledge," in *Proceedings of OOPSLA '91*, pp. 212–229, 1991.

[6] Asperti, A., G. L. Ferrari, and R. Gorrieri, "Implicative Formulae in the "Proofs as Computations Analogy"," in *Proceedings of SIGACT/SIGPLAN Symposium on Principles of Programming Language*, pp. 59–71, 1990.

[7] Brookes, S. D., C. A. R. Hoare, and A. W. Roscoe, "A Theory of Communicating Sequential Processes," *Journal of ACM*, vol. 31, no. 3, pp. 560–599, July 1984.

[8] Darlington, J., and Y. Guo, "Definitional Constraint Programming for Parallel Computing: An Introduction," in *Workshop of FGCS '92*, pp. 1–17, 1992.

[9] Dershowitz, N., and J.-P. Jouannaud, "Rewrite Systems," in *Handbook of Theoretical Computer Science Volume B* (J. V. Leeuwen, ed.), ch. 6, pp. 243–320, The MIT press/Elsevier, 1990.

[10] Engberg, U., and G. Winskel, "Petri Nets as Models of Linear Logic," in *Proceedings of CAAP'90*, vol. 431 of *Lecture Notes in Computer Science*, 1990.

[11] Gehlot, V., and C. Gunter, "Normal Process Representatives," in *Proceedings of IEEE Symposium on Logic in Computer Science*, pp. 200–207, 1990.

[12] Girard, J.-Y., "Linear Logic," *Theoretical Computer Science*, vol. 50, pp. 1–102, 1987.

[13] Girard, J.-Y., Y. Lafont, and P. Taylor, *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1990.

[14] Lafont, Y., "Interaction Nets," in *Proceedings of Seventeenth ACM SIGPLAN/SIGACT Symposium on Principles Of Programming Language*, pp. 95–108, 1990.

[15] Lloyd, J., *Foundations of Logic Programming.* Springer-Verlag, 2nd ed., 1987.

[16] Marti-Oliet, N., and J. Meseguer, "From Petri Nets to Linear Logic," in *Category Theory and Computer Science*, vol. 389 of *Lecture Notes in Computer Science*, pp. 313–337, Springer Verlag, 1989.

[17] Miller, D., "The π-calculus as a theory in linear logic: Preliminary results," Tech. Rep. MS-CIS-92-48, Computer Science Department, University of Pennsylvania, 1992. To appear in the 1992 Workshop on Extensions to Logic Programming, LNAI Series.

[18] Milner, R., "Calculi for Synchrony and Asynchrony," *Theoretical Computer Science*, vol. 25, pp. 267–310, 1983.

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

[20] Milner, R., J. Parrow, and D. Walker, "A Calculus of Mobile Processes, Part I," Tech. Rep. ECS-LFCS-89-85, University of Edinburgh, 1989.

[21] Milner, R., J. Parrow, and D. Walker, "A Calculus of Mobile Processes, Part II," Tech. Rep. ECS-LFCS-89-86, University of Edinburgh, 1989.

[22] Nicola, R. D., and M. C. B. Hennessy, "Testing Equivalence for Processes," *Theoretical Computer Science*, vol. 34, pp. 83–133, 1984.

[23] Saraswat, V. A., "Concurrent Constraint Programming," in *Proceedings of SIGACT/SIGPLAN Symposium on Principles of Programming Language*, pp. 232–244, ACM, 1990.

[24] Shapiro, E., "The Family of Concurrent Logic Programming Languages," *ACM Computing Surveys*, vol. 21, no. 3, pp. 413–510, September 1989.

[25] Yonezawa, A., *ABCL: An Object-Oriented Concurrent System.* MIT Press, 1990.

[26] Yonezawa, A., and M. Tokoro, *Object-Oriented Concurrent Programming.* The MIT Press, 1987.

Appendix A Inference Rules in Linear Sequent Calculus

We review inference rules in linear sequent calculus[12].

- Logical axioms

$$\vdash A, A^{\perp}$$

- Cut rule

$$\frac{\vdash A, \Gamma \qquad \vdash A^{\perp}, \Delta}{\vdash \Gamma, \Delta}$$

- Exchange rule

$$\frac{\vdash \Gamma}{\vdash \Delta}$$

  where Γ is a permutation of Δ.

- Additive rules

$$\vdash \top, A$$

$$\frac{\vdash A, \Gamma \qquad \vdash B, \Gamma}{\vdash A \,\&\, B, \Gamma}$$

$$\frac{\vdash A, \Gamma}{\vdash A \oplus B, \Gamma} \qquad \frac{\vdash B, \Gamma}{\vdash A \oplus B, \Gamma}$$

- Multiplicative rules

$$\vdash \mathbf{1} \qquad \frac{\vdash A}{\vdash \bot, A}$$

$$\frac{\vdash A, \Gamma \qquad \vdash B, \Delta}{\vdash A \otimes B, \Gamma, \Delta} \qquad \frac{\vdash A, B, \Gamma}{\vdash A \parr B, \Gamma}$$

- Exponential rules

$$\frac{\vdash A, \Gamma}{\vdash ?A, \Gamma} \text{ dereliction}$$

$$\frac{\vdash \Gamma}{\vdash ?A, \Gamma} \text{ weakening}$$

$$\frac{\vdash ?A, ?A, \Gamma}{\vdash ?A, \Gamma} \text{ contraction}$$

$$\frac{\vdash A, ?\Gamma}{\vdash !A, ?\Gamma}$$

- Quantifier rules

$$\frac{\vdash A, \Gamma}{\vdash \forall x.A, \Gamma} \text{ x not free in } \Gamma$$

$$\frac{\vdash A[t/x], \Gamma}{\vdash \exists x.A, \Gamma}$$

Appendix B Proof of Proposition 1

Lemma 5 *Let P be a program and A be a body. $\vdash A$ is provable in ACL inference rules if and only if $\vdash A$ is provable by using inference rules in linear sequent calculus and clause rule for ACL.*

Proof of Lemma 5 ($\rightarrow$) trivial.
($\leftarrow$)

We show that every proof in linear sequent calculus can be translated to a ACL proof. We can assume that a proof in linear sequent calculus is cut-free by cut-elimination theorem.

Exchange rule, $\top$-rule, $\bot$-rule, and &-rule are unchanged, because they exist also in ACL inference rules.

1-rule and !-rule cannot appear, because **1** and ! never appears in ACL syntax.

Rests are axiom($\vdash A, A^{\bot}$), $\oplus$-rule, $\otimes$-rule, and ?-rule(dereliction, weakening, contraction). In the following we show that these rules can be eliminated.

- $\vdash A, A^{\bot}$
  From the syntax of ACL, $A^{\bot}$ must come from *guard*. Then, the proof must be the following structure:

$$\dfrac{\dfrac{\dfrac{\vdash A, A^{\bot}}{\cdots\cdots}}{\vdash A^{\bot}, \Gamma} \qquad \dfrac{\cdots}{\vdash B, \Delta}}{\dfrac{\vdash A^{\bot} \otimes B, \Gamma, \Delta}{\cdots}}$$

  We can translate this proof into

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash A, A^{\bot}}{\vdash A^{\bot} \otimes B, A, \Delta} \qquad \dfrac{\cdots}{\vdash B, \Delta}}{\cdots}}{\vdash A^{\bot} \otimes B, \Gamma, \Delta}}{\cdots} ACL \text{ ACL Rule (C2)}$$

- $\oplus$-rule
  We translate $\oplus$-rule by the case analysis of rule before $\oplus$-rule. If the proof is the following form:

$$\dfrac{\dfrac{\vdash A, \Gamma'}{\vdash A, \Gamma}}{\vdash A \oplus B, \Gamma}$$

  we can convert it to:

$$\dfrac{\dfrac{\vdash A, \Gamma'}{\vdash A \oplus B, \Gamma'}}{\vdash A \oplus B, \Gamma}$$

  From the syntax of ACL, A must be of the form $m_1^{\bot} \otimes \ldots \otimes m_n^{\bot} \otimes A'$. If $\oplus$-rule follows &-rule:

$$\dfrac{\dfrac{\vdash A, C, \Gamma' \qquad \vdash A, D, \Gamma'}{\vdash A, C \,\&\, D, \Gamma'}}{\vdash A \oplus B, C \,\&\, D, \Gamma'}$$

we can convert it to:

$$\dfrac{\dfrac{\vdash A, C, \Gamma'}{\vdash A \oplus B, C, \Gamma} \qquad \dfrac{\vdash A, D, \Gamma'}{\vdash A \oplus B, D, \Gamma'}}{\vdash A \oplus B, C \,\&\, D, \Gamma'}$$

If $\oplus$-rule follows $\top$-rule:

$$\dfrac{\dfrac{}{\vdash A, \top, \Gamma'}}{\vdash A \oplus B, \top, \Gamma'}$$

we can convert it to:

$$\dfrac{}{\vdash A \oplus B, \top, \Gamma'}$$

If $\oplus$-rule follows from $\oplus$-rule:

$$\dfrac{\dfrac{\vdash A, \Gamma}{\vdash A \oplus B, \Gamma}}{\vdash A \oplus B \oplus C, \Gamma}$$

we can convert it to:

$$\dfrac{\vdash A, \Gamma}{\vdash A \oplus B \oplus C, \Gamma}$$

The rest is $\otimes$-rule ($\otimes is from A$):

$$\dfrac{\dfrac{\vdash m_1^\perp \otimes \ldots \otimes m_i^\perp, \Gamma \quad \vdash m_{i+1}^\perp \otimes \ldots \otimes m_n^\perp \otimes A', \Delta}{\vdash m_1^\perp \otimes \ldots \otimes m_n^\perp \otimes A', \Gamma, \Delta}}{\vdash (m_1^\perp \otimes \ldots \otimes m_n^\perp \otimes A') \oplus B, \Gamma, \Delta}$$

If $i > 1$, we can assume that the inference rule above the left leaf is again $\otimes$-rule:

$$\dfrac{\vdash m_1^\perp \otimes \ldots \otimes m_j^\perp, \Gamma_1 \qquad \vdash m_{j+1}^\perp \otimes \ldots \otimes m_i^\perp, \Gamma_2}{\vdash m_1^\perp \otimes \ldots \otimes m_i^\perp, \Gamma (= \Gamma_1, \Gamma_2)}$$

because we can move the other rules below $\oplus$-rule. Then, we can convert it to:

$$\dfrac{\dfrac{\vdash m_1^\perp \otimes \ldots \otimes m_j^\perp, \Gamma_1 \qquad \dfrac{\vdash m_{j+1}^\perp \otimes \ldots \otimes m_i^\perp, \Gamma_2 \quad \vdash m_{i+1}^\perp \otimes \ldots \otimes m_n^\perp \otimes A', \Delta}{\vdash m_{j+1}^\perp \otimes \ldots \otimes m_n^\perp \otimes A', \Gamma_2, \Delta}}{\vdash m_1^\perp \otimes \ldots \otimes m_n^\perp \otimes A', \Gamma_1, \Gamma_2, \Delta}}{\vdash (m_1^\perp \otimes \ldots \otimes m_n^\perp \otimes A') \oplus B, \Gamma_1, \Gamma_2, \Delta}$$

Therefore, we can assume $i = 1$:

$$\dfrac{\dfrac{\vdash m_1^\perp, \Gamma \qquad \vdash m_2^\perp \otimes \ldots \otimes m_n^\perp \otimes A', \Delta}{\vdash m_1^\perp \otimes \ldots \otimes m_n^\perp \otimes A', \Gamma, \Delta}}{\vdash (m_1^\perp \otimes \ldots \otimes m_n^\perp \otimes A') \oplus B, \Gamma, \Delta}$$

and we can also assume that Γ is m_1. Then we can convert it to:

$$\dfrac{\vdash m_1, m_1^\perp \qquad \vdash m_2^\perp \otimes \ldots \otimes m_n^\perp \otimes A', \Delta}{\vdash (m_1^\perp \otimes \ldots \otimes m_n^\perp \otimes A') \oplus B, m_1, \Delta}\ \text{rule (C2)}$$

- $\otimes$-rule
  From the syntax of ACL, $\otimes$ appears only in the forms of $m_1^{\perp} \otimes \ldots \otimes m_n^{\perp} \otimes A$. Then, the same argument holds as in the case of $\oplus$-rule.
- ?-rule(contraction)
  By the above translation, the proof contains only ACL inference rules and ?-rules. We can assume that a contraction rule follows weakening or dereliction. because we can move other rules below the contraction rule.
  - Case I: contraction rule follows weakening rule
    In this case, the proof is of the following form:

$$\dfrac{\dfrac{\dfrac{\cdots\cdots}{\vdash ?A, \Gamma}}{\vdash ?A, ?A, \Gamma}}{\vdash ?A, \Gamma}$$

    We can convert it to :

$$\dfrac{\cdots\cdots}{\vdash ?A, \Gamma}$$

  - Case II: contraction rule follows dereliction rule
    The proof is of the form:

$$\dfrac{\dfrac{\vdash ?A, A, \Gamma}{\vdash ?A, ?A, \Gamma}}{\vdash ?A, \Gamma}$$

    we can assume the dereliction rule follows the rule that involves the formula 'A', because in other cases we can again move it below the contraction rule. From the syntax of ACL, A must be an atomic formula $m \in A_m$. The proof is, therefore, of the following form:

$$\dfrac{\dfrac{\dfrac{\vdash m, m^{\perp} \qquad \vdash ?m, A, \Gamma'}{\vdash ?m, m, (m^{\perp} \otimes A) \oplus B, \Gamma'}\ \text{ACL rule (C2)}}{\vdash ?m, ?m, (m^{\perp} \otimes A) \oplus B, \Gamma'}}{\vdash ?m, (m^{\perp} \otimes A) \oplus B, \Gamma'}$$

    Then we can convert it to:

$$\dfrac{\vdash m, m^{\perp} \qquad \vdash ?m, A, \Gamma'}{\vdash ?m, (m^{\perp} \otimes A) \oplus B, \Gamma'}\ \text{ACL rule (C3)}$$

- ?-rule(dereliction)
  We can assume that the dereliction rule follows ACL rule (C2):

$$\dfrac{\dfrac{\vdash m, m^{\perp} \qquad \vdash A, \Gamma'}{\vdash m, (m^{\perp} \otimes A) \oplus B, \Gamma'}\ \text{ACL rule (C2)}}{\vdash ?m, (m^{\perp} \otimes A) \oplus B, \Gamma'}$$

  Then we can convert it to:

$$\dfrac{\vdash m, m^{\perp} \qquad \dfrac{\vdash A, \Gamma'}{\vdash ?m, A, \Gamma'}\text{Weakening}}{\vdash ?m, (m^{\perp} \otimes A) \oplus B, \Gamma'}\ \text{ACL rule (C3)}$$

- ?-rule(weakening) Now the proof contains only ACL rules and weakening rules. If the weakening rule follows ACL-rule (T1)($\top$-rule):

$$\dfrac{\dfrac{}{\vdash \Gamma, \top}\ \text{ACL rule (T1)}}{\vdash ?A, \Gamma, \top}$$

we can convert it to:

$$\dfrac{}{\vdash ?A, \Gamma, \top}\ \text{ACL rule (T1)}$$

In other cases, the proof is of the form:

$$\dfrac{\dfrac{\dfrac{\vdash \cdots}{\vdash \Gamma'}}{\vdash \Gamma}}{\vdash ?A, \Gamma}$$

Then we convert it to:

$$\dfrac{\dfrac{\vdash ?A, \cdots}{\vdash ?A, \Gamma'}}{\vdash ?A, \Gamma}$$

□

Proof Sketch of Proposition 1 From the above lemma, it is enough to show that $\vdash A$ is provable by inference rules in linear sequent calculus and ACL clause rule (Cl1) if and only if $\vdash ?P^{\perp}, A$ is provable in linear sequent calculus.

$\rightarrow$ is trivial.

($\leftarrow$)

Suppose that we have a proof of $\vdash ?P^{\perp}, A$. We only have to consider the translation of inferences on the formula $?P^{\perp}$. Program P is in the form of $\&_i(Head_i \circ\!\!- Body_i)$, hence $P^{\perp} = \oplus_i(Head_i^{\perp} \otimes Body_i)$. By the similar discussion in the proof of the above lemma, we can assume that the inference goes on as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash Head_i^{\perp}, Head_i \qquad \vdash ?P^{\perp}, Body_i, \Gamma}{\vdash ?P^{\perp}, Head_i^{\perp} \otimes Body_i, Head_i, \Gamma}}{\vdash ?P^{\perp}, P^{\perp}(= \oplus_j(Head_j^{\perp} \otimes Body_j)), Head_i, \Gamma}}{\vdash ?P^{\perp}, ?P^{\perp}, Head_i, \Gamma}}{\vdash ?P^{\perp}, Head_i, \Gamma}$$

Then we can convert it to:

$$\dfrac{\vdash Body_i, \Gamma}{\vdash Head_i, \Gamma}\ \text{ACL rule (Cl1)}$$

□