Logical, Testing, and Observation Equivalence for Processes in a Linear Logic Programming

Naoki Kobayashi Akinori Yonezawa
Department of Information Science
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

Abstract

Linear logic programming has recently been proposed and shown to be able to integrate a wide range of mechanisms for concurrent computation uniformly in terms of proof search. This paper introduces several process equivalence relations for linear logic programming and investigate relationships between them in order to establish foundations for program transformation in linear logic programming. We also introduce unfold/fold transformation procedure for linear logic programming and prove its soundness with respect to a testing equivalence.

1 Introduction

Linear logic programming, where computation is described in terms of a bottom-up proof search in linear logic[5], has recently been investigated as frameworks for novel concurrent programming languages[3][2][6][7][8][12]. The general goal of this paper is to investigate several equivalence relations for processes in linear logic programming, by which establishing foundations for program transformation in linear logic programming. In this paper, we consider a subset of ACL[7], a variant of linear logic programming.

One of our particular interests is in an issue: "What does *logical equivalence* of formulas mean in terms of computation?" In ACL (and also in linear `cc`[12]), each formula of linear logic is viewed as a process, hence logical equivalence is expected to imply some equivalence for processes. The first simple observation is that logical equivalence implies a *weak* testing equivalence[11]. A logical equivalence of two formulas A and B implies that for any multiset of formulas Γ, $\vdash A, \Gamma$ is provable if and only if $\vdash B, \Gamma$ is provable. From a computational point of view, this implies that under any context Γ, a process A can succeed if and only if a process B can succeed. Thus, logical equivalence naturally derives a testing equivalence in linear logic programming. It is very interesting that the structural equivalence in Milner's π-calculus is also naturally derived from logical equivalence. However, the derived testing equivalence generally captures only the notion of "*may* success", but not of "*must* success", due to non-determinism; "A is provable" does not necessarily mean that "A is provable by any proof strategy". This equivalence is, therefore, unsatisfactory for the use in program transformation. In spite of that, for a more restricted fragment of processes, we can prove that logical equivalence implies a testing equivalence in terms of "must success". In order to prove it, we also introduce a bisimulation equivalence. Then, the logical equivalence implies this observation equivalence as well as three testing equivalences. The logical equivalence thus gives a fine equivalence for processes. In fact, from the proof of the above fact, we can see logical equivalence just corresponds to *structural equivalence*.

Although logical equivalence gives a fine equivalence on a certain set of processes, we practically need to consider a broader set of processes as a basis of program transformations. Therefore, we also

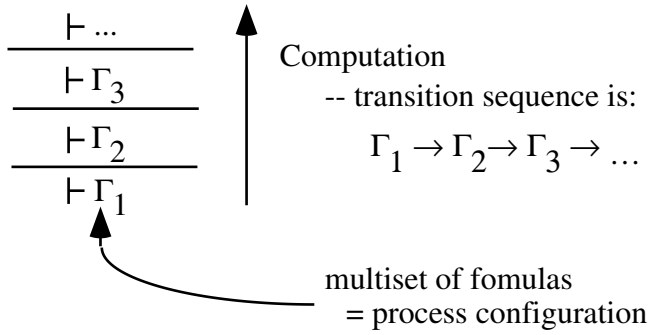

Figure 1: Inference Figure and Computation in ACL

introduce three testing equivalences, bisimulation equivalence, and trace equivalence in linear logic programming, and investigate relationships between them. Among them, a *strong* testing equivalence seems most suitable as a behavioral equivalence for processes. Then, we give a unfold/fold program transformation procedure and prove its soundness with respect to the strong testing equivalence. Since process equivalences for many other concurrent programming languages based on asynchronous communication, such as the Actor[1] and Linda[4], can be obtained by restricting *testers* in ACL testing equivalence, the soundness of transformation procedures in ACL means also the soundness in those languages, that is, the program transformation techniques devised in ACL can be applied to those languages.

Although we investigate a subset of propositional ACL for the simplicity, most of the results in this paper can be carried over to (first-order) full ACL. The full ACL integrates many mechanisms for concurrent computation, such as value passing and name creation, uniformly in terms of proof search in linear logic, and provides a variety of communication models: point-to-point, one-to-many, and one-to-one-of-many communication[7]. A model theoretic semantics based on phase semantics is given in [6].

The remainder of this paper is organized as follows. Section 2 defines the core language we work on in this paper. Section 3, Section 4, and Section 5 respectively introduces a logical equivalence, testing equivalences, and bisimulation equivalence. Section 6 investigates relationships between those equivalence relations. Section 7 introduces an unfold/fold program transformation procedure and proves its soundness. Section 8 concludes this paper. Notations used in this paper are summarized in Appendix A.

2 The Core Language

First, we introduce a simple language to work on in this paper, which is a subset of ACL[6][7]. We call this language the core-ACL.

2.1 core-ACL

The core-ACL is a concurrent programming paradigm where concurrent processes run in parallel, communicating with each other via asynchronous message passing. Each formula of (propositional) linear logic is viewed as a *process* or *message*. Each sequent (multiset of formulas) $\vdash \Gamma$ is viewed as a *process configuration* in some state. As in other linear logic programming languages[3][12], bottom-up proof construction for a sequent $\vdash \Gamma$ corresponds to computation beginning with an initial process configuration Γ. For example, given the inference figure shown in Figure 1, we read it as Γ_1 transits to Γ_2, Γ_2 transits to Γ_3, and so on. Transition rules in ACL are, therefore, given in a form of inference rules which give these inferences. These correspondences are summarized in Table 1.

Linear Logic	ACL
formula	process or message
sequent	process configuration
proof	trace of computation
inference rule	transition rule

Table 1: Correspondence between Linear Logic and ACL

2.2 Syntax

2.2.1 Communicating agents

Sender A sender of a message m is represented by a formula $m \parr A$ where A stands for a process after sending the message.

Receiver A receiver of a message m is represented by a formula $m^{\perp} \otimes B$, where a formula B stands for a process after receiving the message. Generally, a receiver process is represented by the following formula:

$$(m_1^{\perp} \otimes B_1) \oplus \cdots \oplus (m_n^{\perp} \otimes B_n)$$

This formula represents a process which waits for any one of messages $m_1, \ldots, m_n$ and then becomes B_i upon receiving a message m_i.

2.2.2 Parallel Composition

Parallel composition of processes A and B is represented by a formula $A \parr B$.

2.2.3 Special Constants

$\top$ and $\mathbf{0}$, special predicates in linear logic, are introduced. $\top$ represents a terminator which terminates a whole computation, while $\mathbf{0}$ only vanishes by itself.

2.2.4 Recursive Agents

Processes can be recursively defined by introducing a *clause*. A clause is of the form: $A \multimap B$, where $\multimap$ is an implication($\leftarrow$) in linear logic. The intuitive meaning of $A \multimap B$ is "a process A can become B". A must be an atomic formula. $A \multimap B$ is called a definition clause of A. We call a set of clauses *program*. We use a symbol $\mathbf{P}$ for a program. We assume that in any program $\mathbf{P}$, each predicate has at most one definition clause.

We use an alphabetic string starting with a capital letter to denote a predicate which has its definition clause and call it a *process predicate*. For other predicates, we use small letters and call them *message predicates*.

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

$$EmptyBuffer \multimap put^{\perp} \otimes FullBuffer$$

$$FullBuffer \multimap get^{\perp} \otimes (reply \parr EmptyBuffer)$$

The first clause means that $EmptyBuffer$ waits for a message put, and after the reception, it becomes a $FullBuffer$. The second clause means that $FullBuffer$ waits for a message get, then, after reception, sends a message $reply$, and becomes a $EmptyBuffer$. This definition is quite similar to the following description in CCS[9],

$$EmptyBuffer = put.FullBuffer$$

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

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

2.2.5 Summary of Syntax

The set of goal formulas, ranged by G, is defined by the following syntax:

$G ::= \mathbf{0} \mid \top \mid A_m \mid A_P \mid G \parr G \mid R$
$R ::= A_m^{\perp} \otimes G \mid R \oplus R$
$A_m ::=$ an atomic formula whose predicate is a message predicate.
$A_P ::=$ an atomic formula whose predicate is a process predicate.
$Cl ::= A_P \circ\!\!- G$

We denote a set of goal formulas by $\mathcal{P}$, and a set of process configurations, that is, a set of multisets of goal formulas by $\mathcal{PC}$.

2.3 Operational Semantics

A multiset of goal formulas Γ is called a process configuration. We define transitions relations $\longrightarrow \subset \mathcal{PC}^+ \times \mathcal{PC}^+$ over process configurations ($\mathcal{PC}^+ \equiv \mathcal{PC} \cup \{\mathbf{Success}\}$) below. $\uplus$ denotes a multiset union. Statement $\mathbf{P} \vdash \Gamma \longrightarrow \Delta$ means that Γ can transit to Δ, under a program $\mathbf{P}$. **Success** represents a successful termination of the computation.

- Parallel

  (P) $\dfrac{}{\mathbf{P} \vdash \{A \parr B\} \uplus \Gamma \longrightarrow \{A, B\} \uplus \Gamma}$

- Communication Rules

  (C) $\dfrac{}{\mathbf{P} \vdash \{\oplus_i(\vec{m_i}^{\perp} \otimes (\vec{n_i} \parr A_i)), \vec{m_j}\} \uplus \Gamma \longrightarrow \{A_j, \vec{n_j}\} \uplus \Gamma}$
  where $\vec{m}$ stands for a multiset of message predicates, and $\{m_1, \ldots, m_k\}^{\perp} \otimes (\{n_1, \ldots, n_l\} \parr A)$ stands for a formula: $m_1^{\perp} \otimes \cdots \otimes m_k^{\perp} \otimes (n_1 \parr \cdots \parr n_l \parr A)$. For example, $\vec{m_1}^{\perp} \otimes (\vec{n_1} \parr A)$ means $m_{11}^{\perp} \otimes m_{12}^{\perp} \otimes (n_{11} \parr A)$ where $\vec{m_1} = \{m_{11}, m_{12}\}$ and $\vec{n_1} = \{n_{11}\}$.

- Termination Rules

  (T1) $\dfrac{}{\mathbf{P} \vdash \{\top\} \uplus \Gamma \longrightarrow \mathbf{Success}}$

  (T2) $\dfrac{}{\mathbf{P} \vdash \{\mathbf{0}\} \uplus \Gamma \longrightarrow \Gamma}$ $\cdots$(suicide)

- Clause Rule

  (Cl1) $\dfrac{A \circ\!\!- G \in \mathbf{P} \text{ and } \mathbf{P} \vdash \Gamma[G/A] \longrightarrow \Delta}{\mathbf{P} \vdash \Gamma \longrightarrow \Delta}$
  where $\Gamma[G/A]$ is a multiset of formulas obtained by replacing one occurrence of A in Γ by G.

We write $\longrightarrow^*$ for a reflexive and transitive closure of $\longrightarrow$.

The following proposition establishes the connection between core-ACL and linear logic. The proof is given in [6].

Proposition 2.1 Let $\Gamma \in \mathcal{PC}$. A sequent $!\mathbf{P} \vdash \Gamma$ is provable in linear logic if and only if $\mathbf{P} \vdash \Gamma \longrightarrow^* \mathbf{Success}$.

Definition 2.1 (Transition sequence) A sequence $\Gamma_0 \longrightarrow \Gamma_1 \longrightarrow \cdots \longrightarrow \Gamma_n$ is called a *(finite) transition sequence* of Γ_0 under a program $\mathbf{P}$, if $\mathbf{P} \vdash \Gamma_{i-1} \longrightarrow \Gamma_i$ for all $i (1 \leq i \leq n)$. An infinite sequence $\Gamma_0 \longrightarrow \Gamma_1 \longrightarrow \cdots$ is called a *(infinite) transition sequence* of Γ_0 under a program $\mathbf{P}$, if $\mathbf{P} \vdash \Gamma_{i-1} \longrightarrow \Gamma_i$ for all $i (\geq 1)$.

This definition will be used for defining equivalence relations for processes in the remainder of this paper.

2.4 Normal processes

In this section, we introduce two subsets of processes, $\mathcal{P}^-$ and $\mathcal{NP}$, on which we give equivalence relations in the remainder of this paper. $\mathcal{P}^-$ is a set of processes which do not contain a terminator $\top$. $\mathcal{NP}$ is a more restricted set of processes; processes in $\mathcal{NP}$ have less non-determinism and cannot fall into infinite transitions by itself.

Before introducing $\mathcal{P}^-$ and $\mathcal{NP}$, we need the following definition.

Definition 2.2 (Finite expansion, Finite approximation) Let F be a goal formula and $\mathbf{P} = \{A_i \circ\!\!- G_i \mid 1 \leq i \leq n\}$ a program. A formula obtained by substituting G_i for each occurrence of A_i is called an *expansion* of F by $\mathbf{P}$. A formula obtained by expanding F by $\mathbf{P}$ finite times is called a *finite expansion* of F by $\mathbf{P}$. A formula obtained by replacing all A_is in a finite expansion of F by $\mathbf{0}$ is called a *finite approximation* of F.

A finite approximation of a recursively defined process P gives an approximation of P in some sense. For example, if P is defined by a clause $P \circ\!\!- m^\perp \otimes (n \parr P)$, an approximation $m^\perp \otimes (n \parr m^\perp \otimes (n \parr \mathbf{0}))$, which is obtained by expanding P twice, behaves the same with P in two steps of transitions.

Definition 2.3 ($\mathcal{P}^-$) A subset $\mathcal{P}^-$ of $\mathcal{P}$ is defined as follows.

$$\mathcal{P}^- = \{F \mid F \in \mathcal{P} \text{ and any finite expansion of } F \text{ does not contain } \top\}$$

Intuitively, $\mathcal{P}^-$ is a set of processes which can be defined without a terminator $\top$.

Definition 2.4 (Normal processes $\mathcal{NP}$) *Normal processes* are defined as follows:

1. $\mathbf{0}$ is a normal process.
2. If $A_i (1 \leq i \leq k)$ are normal processes, $A_1 \parr \cdots \parr A_k$ is a normal process.
3. If $A_i (1 \leq i \leq k)$ are normal processes, $(\vec{m_1}^\perp \otimes (\vec{n_1} \parr A_1)) \oplus \cdots \oplus (\vec{m_k}^\perp \otimes (\vec{n_k} \parr A_k))$ is a normal process, where $\vec{m_i}$ and $\vec{n_i}$ are multisets of message formulas, and there is no i and j $(i \neq j)$ such that $\vec{m_i} \subseteq \vec{n_i}, \vec{n_i} \subseteq \vec{m_i}$, or $\vec{m_i} \subseteq \vec{m_j}$.
4. If all finite approximations of A are normal processes, and if for any finite subformula of finite expansions A', $\{A'\} (\in \mathcal{PC})$ has no infinite transition sequence, A is a normal process.

We write $\mathcal{NP}$ for a set of normal processes.

The third restriction rules out a process such as $(m^\perp \otimes A) \oplus (m^\perp \otimes B)$, which reveals non-deterministic behavior upon receiving a message. The fourth restriction rules out a process which falls into an infinite computation by itself.

Example 2.1 $(m_1^\perp \otimes (n_1 \parr \mathbf{0})) \oplus (m_2^\perp \otimes (n_2 \parr \mathbf{0})) \in \mathcal{NP}$, but $m_1^\perp \otimes (m_1 \parr \mathbf{0}) \notin \mathcal{NP}$, and $(m_1^\perp \otimes (n_1 \parr \mathbf{0})) \oplus (m_1^\perp \otimes m_2^\perp \otimes (n_2 \parr \mathbf{0})) \notin \mathcal{NP}$.

Example 2.2 Let $\mathbf{P}$ be $\{A \circ\!\!- m^\perp \otimes (n \parr A), B \circ\!\!- m^\perp \otimes B\}$. $A \in \mathcal{NP}$, but $B \notin \mathcal{NP}$.

Example 2.3 Let $\mathbf{P}$ be $\{A \circ\!\!- m \parr A\}$. $A \notin \mathcal{NP}$, because $\{A\}$ has an infinite transition sequence:

$$\{A\} \longrightarrow \{m, A\} \longrightarrow \{m, m, A\} \longrightarrow \cdots$$

3 Logical Equivalence

As is described in the previous section, each formula in linear logic is viewed as a process or a message. Logical equivalence of formulas, therefore, can be viewed as a process equivalence from computational point of view.

Definition 3.1 (Logical equivalence) Let A and B be goal formulas, and $\mathbf{P}$ a program. A and B are said *logically equivalent* under a program $\mathbf{P}$ and written as $!\mathbf{P} \vdash A \circ\!\!-\!\!\circ B$, iff both $!\mathbf{P} \vdash A \multimap B$ and $!\mathbf{P} \vdash B \multimap A$ are provable in linear sequent calculus. $!\mathbf{P} \vdash A \circ\!\!-\!\!\circ B$ can be written as $A \circ\!\!-\!\!\circ B$ if $\mathbf{P}$ is trivial from context or unimportant.

The above logical equivalence is too fine, especially for recursive processes. For example, let $\mathbf{P}$ be $\{A \circ\!\!- m^{\perp} \otimes (n \parr A), B \circ\!\!- m^{\perp} \otimes (n \parr B)\}$. Then, A and B are clearly equivalent from computational point of view, but neither $!\mathbf{P} \vdash A \multimap B$ nor $!\mathbf{P} \vdash B \multimap A$ is provable. Therefore, we go on to the following *weak* logical equivalence.

Definition 3.2

$$[\![A]\!]_{\mathbf{P}} = \{\Gamma \mid \Gamma \text{ contains no process predicate, and } !\mathbf{P} \vdash A, \Gamma \text{ is provable in linear logic.}\}^{1}$$

Definition 3.3 (Weak logical equivalence) $A \cong_{LL} B$ *iff* $[\![A]\!]_{\mathbf{P}} = [\![B]\!]_{\mathbf{P}}$

Then, it is easy to see that $A \cong_{LL} B$ in the above example.

The following proposition is trivial.

Proposition 3.1 $A \circ\!\!-\!\!\circ B \Longrightarrow A \cong_{LL} B$

The following proposition may be useful to prove $\cong_{LL}$.

Proposition 3.2 $[\![A]\!]_{\mathbf{P}} \supseteq [\![B]\!]_{\mathbf{P}}$ *iff for any finite approximation* A' *of* A, $!P \vdash A'^{\perp}, B$ *is provable in linear logic.*

The proof is easy from cut-elimination theorem.

$[\![\bullet]\!]'_{\mathbf{P}}$, which is given by the following two definitions, would be of more help to understand operational semantics.

Definition 3.4 (Coprocesses $\mathcal{CP}$) A set of *coprocesses* $\mathcal{CP}$, ranged over by G', is given by:

$$G' ::= G \mid G' \otimes G' \mid G' \,\&\, G'$$

where G ranges over $\mathcal{P}$.

Definition 3.5 ($[\![\bullet]\!]'_{\mathbf{P}}$) A map $[\![\bullet]\!]'_{\mathbf{P}}$ from $\mathcal{P} \cup \{\mathbf{Success}\}$ to a power set of $\mathcal{CP}$ is given by:

$[\![A]\!]'_{\mathbf{P}} = [\![A]\!]_{\mathbf{P}} \cap \mathcal{CP}$ if $A \in \mathcal{P}$
$[\![\mathbf{Success}]\!]'_{\mathbf{P}} = \{\Gamma \mid \Gamma \in \mathcal{CP}$ and Γ contains no process predicates.$\}$

Then, the following proposition holds.

Proposition 3.3 Let $\Gamma, \Delta \in \mathcal{PC}^{+}$. If $\Gamma \longrightarrow \Delta$, then

$$\begin{cases} [\![\parr\Gamma]\!]'_{\mathbf{P}} \supseteq [\![\parr\Delta]\!]'_{\mathbf{P}} & \text{if } \longrightarrow \text{ is an application of communication rule} \\ & \text{(Particularly, if } \parr\,\Gamma \in \mathcal{NP}, \text{ then } [\![\parr\Gamma]\!]'_{\mathbf{P}} \supset [\![\parr\Delta]\!]'_{\mathbf{P}}) \\ [\![\parr\Gamma]\!]'_{\mathbf{P}} = [\![\parr\Delta]\!]'_{\mathbf{P}} & \text{otherwise} \end{cases}$$

From the definition, if we interpret **Success** as a successful state of *test*, $[\![A]\!]_{\mathbf{P}} \cap \mathcal{P}$ represents a set of testers of A which *may* succeed, hence $\cong_{LL}$ can be considered a kind of *testing equivalence*[11]. However, both $\circ\!\!-\!\!\circ$ and $\cong_{LL}$ fail to capture some property of processes. For example, let A be $m^{\perp} \otimes n$ and B be $(m^{\perp} \otimes n) \oplus (m^{\perp} \otimes \mathbf{0})$. Although A and B are logically equivalent (in a sense of $\circ\!\!-\!\!\circ$), A and B are not equivalent from the computational point of view. In the next section, we introduce a more strict testing equivalence.

[1] Actually, this set corresponds to a model based on the phase semantics given in [6].

4 Testing Equivalence

This section introduces three testing equivalences[11]. In testing equivalences, given a certain set of tests, two processes are equivalent if results of any tests are the same. We follow the notation used in [1]. The notion of *success* of a test is represented by **Success**.

Definition 4.1 A transition sequence: $\Gamma_0 \longrightarrow \Gamma_1 \longrightarrow \Gamma_2 \longrightarrow \cdots \longrightarrow \Gamma_n$ is called *successful computation* if $\Gamma_n \equiv$ **Success**. A transition sequence is called *failure computation* if it is not a successful computation and,

(i) it is an infinite transition sequence, or
(ii) it is a finite transition sequence: $A_0 \rightarrow A_1 \rightarrow \cdots \rightarrow A_n$, and there is no A_{n+1} such that $A_n \longrightarrow A_{n+1}$.

A tester is a process configuration $\Gamma \in \mathcal{PC}$. A process A is tested by executing A and Γ in parallel and seeing if $\{A\} \cup \Gamma$ transits to **Success**.

Definition 4.2 (Test) Let A be in $\mathcal{P}^-$, and Γ a process configuration. We define $Test : \mathcal{P}^- \times \mathcal{PC} \rightarrow \{\mathtt{s}, \mathtt{sf}, \mathtt{f}\}$ as follows:

$$Test(A,\Gamma) = \begin{cases} \mathtt{s} & \text{if } A,\Gamma \text{ has no failure computation.} \\ \mathtt{sf} & \text{if } A,\Gamma \text{ has both successful computation and failure computation.} \\ \mathtt{f} & \text{if } A,\Gamma \text{ has no successful computation.} \end{cases}$$

We can infer some properties of processes from the result of tests. We give several examples of inferring properties of processes below.

Example 4.1 $Test(A, \{p^\perp \otimes \top\}) = \mathtt{s}$ implies that A eventually sends a message p, because $\{A, p^\perp \otimes \top\}$ transits to **Success** only if $p^\perp \otimes \top$ receives a message p.

Example 4.2 $Test(A, \{p^\perp \otimes \top\}) = \mathtt{sf}$ implies that A *may* eventually send a message p. Because $\{A, p^\perp \otimes \top\}$ can fail, A may not send a message p.

Combinations of multiple tests give more complex information as follows.

Example 4.3 $Test(A, \{p^\perp \otimes \top\}) = \mathtt{f}$ and $Test(A, \{q, p^\perp \otimes \top\}) = \mathtt{sf}$ implies that A *may* eventually receive a message q and send a message p.

The above examples indicate that $Test$ gives some equivalence relation for processes.

Now, we define three testing equivalences, following the definition in Actor[1].

Definition 4.3 (Testing equivalence ($\cong_{1,2,3}$)) Let $A, B \in \mathcal{P}^-$.

1. $A \cong_1 B$ iff $Test(A,\Gamma) = Test(B,\Gamma)$ for all $\Gamma \in \mathcal{PC}$.
2. $A \cong_2 B$ iff $Test(A,\Gamma) = \mathtt{s} \Leftrightarrow Test(B,\Gamma) = \mathtt{s}$ for all $\Gamma \in \mathcal{PC}$.
3. $A \cong_3 B$ iff $Test(A,\Gamma) = \mathtt{f} \Leftrightarrow Test(B,\Gamma) = \mathtt{f}$ for all $\Gamma \in \mathcal{PC}$.

Then, the following propositions are trivial from the definition.

Proposition 4.1 *Let* $A, B \in \mathcal{P}^-$. $A \cong_{LL} B \Longrightarrow A \cong_3 B$

Proposition 4.2 *Let* $A, B \in \mathcal{P}^-$.

1. $A \cong_1 B \Longrightarrow A \cong_2 B$
2. $A \cong_1 B \Longrightarrow A \cong_3 B$

In a practical programming language, it might be disrable that all transition sequences are *fair*, that is, no event is postponed for an infinitely long time. That assumption affects relationships between testing equivalences.

Definition 4.4 (Fair transition sequence) A formula A in a process configuration Γ is *active* if, for some Γ', $\Gamma \longrightarrow \Gamma'$ and Γ' does not contain the same A. A transition sequence: $\Gamma_0 \longrightarrow \Gamma_1 \longrightarrow \cdots$ is *fair* if there is no formula A such that A is active in infinitely many $\Gamma_i (i \geq 0)$.

$\cong_1$ and $\cong_2$ collapse if we consider only fair transition sequences, which is the same result as that of actor model[1]

Proposition 4.3 If transition sequences are always fair, $\cong_1 = \cong_2$ holds.

Proof We show that $A \ncong_1 B \Longrightarrow A \ncong_2 B$. It is sufficient to show that $Test(A,t) = \mathtt{f}$ and $Test(B,t) = \mathtt{sf}$ for some t implies that $A \ncong_2 B$. Let p, q be fresh message predicates which do not occur in A, B or t. Form t' by replacing all occurrences of $\top$ in t by p, and let $t'' = t' \parr ((p^{\perp} \otimes q) \oplus (q^{\perp} \otimes \top)) \parr q$. Then, $Test(A,t'') = \mathtt{s}$ and $Test(B,t'') = \mathtt{sf}$, hence $A \ncong_2 B$. □

5 Bisimulation and Trace Equivalence

In this section, we introduce bisimulation equivalence over normal processes, following Milner's definition in CCS[9].

Definition 5.1 (Labelled interaction: $\stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow}$) If $\vec{p}, A \stackrel{\tau}{\longrightarrow}^* \longrightarrow \vec{q}, B$ where $\stackrel{\tau}{\longrightarrow}^*$ is a reflexive and transitive closure of $\longrightarrow$ except for communication rule, A and B are normal processes, and $\vec{p}$ and $\vec{q}$ are multisets of messages, then we write $A \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} B$. $\downarrow \vec{p} \uparrow \vec{q}$ is called a label of this transition.

Intuitively, $A \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} B$ means that A receives messages $\vec{p}$, then sends messages $\vec{q}$.

Definition 5.2 (Interaction sequence) We write $A \stackrel{s}{\Rightarrow}^* A_n$ for $A \stackrel{\downarrow\vec{p_1}\uparrow\vec{q_1}}{\Rightarrow} A_1 \stackrel{\downarrow\vec{p_2}\uparrow\vec{q_2}}{\Rightarrow} \cdots \stackrel{\downarrow\vec{p_n}\uparrow\vec{q_n}}{\Rightarrow} A_n$, where s is a sequence of labels: $\downarrow \vec{p_1} \uparrow \vec{q_1} \cdots \downarrow \vec{p_n} \uparrow \vec{q_n}$. n is called a *length* of s and written as $length(s) = n$. For all $A \in \mathcal{NP}, A \stackrel{\epsilon}{\Rightarrow}^* A$ and $length(\epsilon) = 0$, where ϵ is a null string.

Definition 5.3 (Trace) $traces(A) = \{s \mid A \stackrel{s}{\Rightarrow}^* A' \text{ for some } A'\}$

Definition 5.4 (Trace equivalence) $A \cong_{tr} B$ iff $traces(A) = traces(B)$

Now, the definition of bisimulation equivalence is straightforward.

Definition 5.5 (Bisimulation) A binary relation $\mathcal{S} \subseteq \mathcal{NP} \times \mathcal{NP}$ is a *bisimulation* if $(A,B) \in \mathcal{S}$ implies, for any label $\downarrow \vec{p} \uparrow \vec{q}$,

(i) Whenever $A \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} A'$ then, for some B', $B \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} B'$ and $(A',B') \in \mathcal{S}$
(ii) Whenever $B \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} B'$ then, for some A', $A \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} A'$ and $(A',B') \in \mathcal{S}$

Definition 5.6 (Bisimulation equivalence) Bisimulation equivalence $\cong_{bi}$ is the largest bisimulation relation.

The following proposition is useful for proving bismulation equivalence.

Proposition 5.1 $A \cong_{bi} B$ iff for all $\downarrow \vec{p} \uparrow \vec{q}$,

(i) Whenever $A \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} A'$ then, for some B', $B \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} B'$ and $A' \cong_{bi} B'$
(ii) Whenever $B \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} B'$ then, for some A', $A \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} A'$ and $A' \cong_{bi} B'$

Bisimulation equivalence is strictly finer than trace equivalence.

Proposition 5.2 $A \cong_{bi} B \Longrightarrow A \cong_{tr} B$

Proof Suppose that $A \cong_{bi} B$ and $s \in traces(A)$. We show $s \in traces(B)$ by induction on the length of s. If $length(s) = 0$, then $s = \epsilon \in traces(B)$. Suppose that $s = \downarrow \vec{p} \uparrow \vec{q}s'$. Then $A \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} A' \stackrel{s}{\Rightarrow}{}^* A''$ for some A', A'', and $s' \in traces(A')$. Because $A \cong_{bi} B$, for some B' $B \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} B'$ and $A' \cong_{bi} B'$. By induction hypothesis, $s' \in traces(A') = traces(B')$. Hence $s \in traces(B)$.
□

The converse of the above proposition does not hold. For example, let A be $(p^\perp \otimes q) \parr (p^\perp \otimes (q \parr r^\perp \otimes s))$ and B be $p^\perp \otimes (q \parr r^\perp \otimes s \parr p^\perp \otimes q)$. Then, $A \cong_{tr} B$ but $A \not\cong_{bi} B$.

6 Relationships between Equivalence Relations

In this section, we investigate relationships among logical, testing, and bisimulation equivalences, over normal processes $\mathcal{NP}$. We can prove that logical equivalence over normal processes is strictly finer than any of three testing equivalences and it corresponds to a structural congruence. The whole results are summarized in the diagram given in Figure 2. We assume that A and B range over $\mathcal{NP}$, unless explicitly specified.

We first prove that the trace equivalence is strictly finer than the the coarsest testing equivalence ($\cong_3$).

Proposition 6.1 $A \cong_{tr} B \Longrightarrow A \cong_3 B$

Proof Suppose $A \cong_{tr} B$ and $Test(A,t) = \mathtt{s}$ or $\mathtt{sf}$ for $t \in \mathcal{PC}$. Then, there is a transition sequence:

$$A, t \longrightarrow A_1, t_1 \longrightarrow \cdots \longrightarrow A_n, \top$$

and

$$A \stackrel{\downarrow\vec{p_1}\uparrow\vec{q_1}}{\Rightarrow} A_1 \stackrel{\downarrow\vec{p_2}\uparrow\vec{q_2}}{\Rightarrow} \cdots \stackrel{\downarrow\vec{p_n}\uparrow\vec{q_n}}{\Rightarrow} A_n$$

Since $trace(A) = trace(B)$, for some $B_1, \ldots, B_n$,

$$B \stackrel{\downarrow\vec{p_1}\uparrow\vec{q_1}}{\Rightarrow} B_1 \stackrel{\downarrow\vec{p_2}\uparrow\vec{q_2}}{\Rightarrow} \cdots \stackrel{\downarrow\vec{p_n}\uparrow\vec{q_n}}{\Rightarrow} B_n$$

hence there is a transition sequence:

$$B, t \longrightarrow B_1, t_1 \longrightarrow \cdots \longrightarrow B_n, \top$$

Therefore, $Test(B,t) = \mathtt{s}$ or $\mathtt{sf}$. □

Corollary 6.2 $A \cong_{bi} B \Longrightarrow A \cong_3 B$

Proposition 6.3 $\cong_3 \neq \cong_{tr}$

Proof Let A be $(p^\perp \otimes q) \parr (p^\perp \otimes q) \parr (p^\perp \otimes q)$ and B be $(p^\perp \otimes p^\perp \otimes (q \parr q)) \parr (p^\perp \otimes q)$. Then, $A \cong_3 B$ but $A \not\cong_{tr} B$.
□

We prove that the bisimulation equivalence is strictly finer than the finest testing equivalence ($\cong_1$).

Proposition 6.4 $A \cong_{bi} B \Longrightarrow A \cong_1 B$

Proof It is sufficient to show that $A \cong_{bi} B$ and $Test(A,t) = \mathtt{s}$ implies $Test(B,t) = \mathtt{s}$. Suppose that $A \cong_{bi} B$ and $Test(A,t) = \mathtt{s}$, but $Test(B,t) = \mathtt{sf}$ or $\mathtt{f}$. Then, B,t has a failure computation, which is an infinite transition sequence or deadlock sequence.
(Case 1: infinite sequence)

Let the infinite sequence be

$$B,t \longrightarrow B_1,t_1 \longrightarrow B_2,t_2 \longrightarrow \cdots.$$

Then, B has an interaction sequence:

$$B \overset{\downarrow\vec{p_1}\uparrow\vec{q_1}}{\Rightarrow} B_1 \overset{\downarrow\vec{p_2}\uparrow\vec{q_2}}{\Rightarrow} B_2 \overset{\downarrow\vec{p_3}\uparrow\vec{q_3}}{\Rightarrow} \cdots.$$

From Proposition 5.2, $A \cong_{bi} B$ implies $A \cong_{tr} B$. Hence A has an interaction sequence:

$$A \overset{\downarrow\vec{p_1}\uparrow\vec{q_1}}{\Rightarrow} A_1 \overset{\downarrow\vec{p_2}\uparrow\vec{q_2}}{\Rightarrow} A_2 \overset{\downarrow\vec{p_3}\uparrow\vec{q_3}}{\Rightarrow} \cdots.$$

Therefore, there is an infinite sequence

$$A,t \longrightarrow A_1,t_1 \longrightarrow A_2,t_2 \longrightarrow \cdots$$

which contradicts the assumption.

(Case 2: deadlock sequence)

Induction on the length of deadlock sequence of B,t. If $A \cong_{bi} B$ and $Test(A,t) = \mathtt{s}$, B,t is not in deadlock state, that is, there is no deadlock sequence of length 0. Suppose that $B,t \longrightarrow B_1,t_1 \longrightarrow \cdots$ is a deadlock sequence of length $n(\geq 0)$. Then, $B_1 \longrightarrow \cdots$ is a deadlock sequence of length $n-1$. Because $A \cong_{bi} B$, $A,t \longrightarrow A_1,t_1$ for some A_1 and $A_1 \cong_{bi} B_1$. $Test(A,t) = \mathtt{s}$ implies $Test(A_1,t_1) = \mathtt{s}$ By induction hypothesis, there is no deadlock sequence of B_1,t_1 with length $n-1$.

□

Proposition 6.5 $\cong_{bi} \neq \cong_1$

Proof Let $\mathbf{P} = \{A \circ\!\!- p^\perp \otimes (q \parr A), B \circ\!\!- p^\perp \otimes p^\perp \otimes (q \parr q)\}$. Then, $A \parr B \cong_1 A$, but $A \parr B \ncong_{bi} A$.
□

Now, we prove that logical equivalence is finer than any of three testing equivalences. Logical equivalence, therefore, plays a role in the core-ACL that is similar to the role played by the structural congruence does in Milner's π-calculus[10]. By Proposition 6.4, it suffices to show that logical equivalence is finer than bisimulation equivalence.

Proposition 6.6 $A \cong_{LL} B \Longrightarrow A \cong_{bi} B$

To prove this proposition, we need the following lemma.

Lemma 6.7 *Let A and B be normal processes. If there is infinitely many finite approximations A' of A such that $[\![A']\!]_{\mathbf{P}} \subseteq [\![B]\!]_{\mathbf{P}}$, then $[\![A]\!]_{\mathbf{P}} \subseteq [\![B]\!]_{\mathbf{P}}$.*

Proof of Proposition 6.6 We show that $\cong_{LL}$ is a bisimulation by the case analysis on the structure of A. For simplicity, we assume that A and B are already expanded several times, so that process predicates do not appear at the outermost positions. Suppose that $A \cong_{LL} B$.
(Case 1) $A \equiv \mathbf{0}$

B also must be $\mathbf{0}$. Therefore $A \cong_{bi} B$.

(Case 2) $A \equiv \vec{p}^\perp \otimes (\vec{q} \parr A')$

Because $\vec{p} \parr (\vec{q}^\perp \otimes \top) \in [\![A]\!]_{\mathbf{P}}$ and $\vec{p'} \parr (\vec{q}^\perp \otimes \top) \notin [\![A]\!]_{\mathbf{P}}$ for any $\vec{p'} \subseteq \vec{p}$, B must be of the form $\vec{p}^\perp \otimes (\vec{r} \parr B')$. By the hypothesis, $A^\perp (\equiv \vec{p} \parr (\vec{q}^\perp \otimes A')) \in [\![A]\!]_{\mathbf{P}} = [\![B]\!]_{\mathbf{P}}$, which implies $\vdash A^\perp, B$ is provable. The cut free proof must be of the following form:

$$\dfrac{\vdash \vec{p},\vec{p}^\perp \qquad \dfrac{\vdash \vec{q}^\perp \otimes A'^\perp, \vec{q}, B''}{\cdots \quad \vdash \vec{q}^\perp \otimes A'^\perp, \vec{r}, B'}}{\vdash \vec{p}, \vec{q}^\perp \otimes A'^\perp, \vec{p}^\perp \otimes (\vec{r} \parr B')}$$

Suppose that $\vec{r}, B' \longrightarrow^* \vec{q}, B''$ in which the communication rule **(C)** is applied at least once. Then from proposition 3.3, $[\![\vec{q} \parr B'']\!]'_{\mathbf{P}} \subset [\![\vec{r} \parr B']\!]'_{\mathbf{P}}$, hence $[\![\vec{q} \parr A']\!]'_{\mathbf{P}} \subset [\![\vec{r} \parr B']\!]'_{\mathbf{P}}$, which contradicts with $[\![A]\!]'_{\mathbf{P}} = [\![B]\!]'_{\mathbf{P}}$. Therefore, $B \equiv \vec{p}^{\perp} \otimes (\vec{q} \parr B')$ and hence $B \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} B'$, $A \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} A'$, and $A' \cong_{LL} B'$.

(Case 3) $A \equiv A_1 \oplus \cdots \oplus A_n$

We assume $n = 2$. A_i is of the form: $\vec{p_i}^{\perp} \otimes (\vec{q_i} \parr A_{i1})$. Because $\vec{p_1} \parr (\vec{q_1}^{\perp} \otimes \top), \vec{p_2} \parr (\vec{q_2}^{\perp} \otimes \top) \in [\![A]\!]_{\mathbf{P}} = [\![B]\!]_{\mathbf{P}}$ and $(\vec{p_1} \parr (\vec{q_1}^{\perp} \otimes \top)) \otimes (\vec{p_2} \parr (\vec{q_2}^{\perp} \otimes \top)) \notin [\![B]\!]_{\mathbf{P}}$ hold, B must be of the form: $B_1 \oplus B_2 \equiv (\vec{p_1}^{\perp} \otimes B_{11}) \oplus (\vec{p_2}^{\perp} \otimes B_{21})$ (or $B_2 \oplus B_1$). For any finite approximation $A'_1 \oplus A'_2$ of A, $(A'_1 \oplus A'_2)^{\perp} \equiv A'^{\perp}_1 \mathbin{\&} A'^{\perp}_2 \in [\![B]\!]_{\mathbf{P}}$, hence for infinitely many $A'_1 \oplus A'_2$, $A'^{\perp}_1 \in [\![B_1]\!]_{\mathbf{P}}$ and $A'^{\perp}_2 \in [\![B_2]\!]_{\mathbf{P}}$ hold, which implies

$$[\![A'_1]\!]_{\mathbf{P}} \subseteq [\![B_1]\!]_{\mathbf{P}}, [\![A'_2]\!]_{\mathbf{P}} \subseteq [\![B_2]\!]_{\mathbf{P}}.$$

By Lemma 6.7, we have

$$[\![A_1]\!]_{\mathbf{P}} \subseteq [\![B_1]\!]_{\mathbf{P}} \text{ and } [\![A_2]\!]_{\mathbf{P}} \subseteq [\![B_2]\!]_{\mathbf{P}}.$$

The converse ($\supseteq$) also holds by the same argument. Therefore,

$$A_1 \cong_{LL} B_1 \text{ and } A_2 \cong_{LL} B_2$$

hold. From Case 2, for any $A \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} C$, there exists D such that $B \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} D$ and $C \cong_{LL} D$.

(Case 4) $A \equiv A_1 \parr \cdots \parr A_n$, the outermost connective of $A_i (1 \leq i \leq n)$ is not $\parr$.

We assume $n = 2$ for simplicity. Because for any finite approximations $A' (\equiv A'_1 \parr A'_2)$ of A, $A'^{\perp} \in [\![B]\!]_{\mathbf{P}}$, B must be of the form $B_1 \parr B_2$ (or $B_2 \parr B_1$) and the cut-free proof of $\vdash A'^{\perp}, B$ is of the form:

$$\frac{\dfrac{\cdots}{\vdash A'^{\perp}_1, B_1} \quad \dfrac{\cdots}{\vdash A'^{\perp}_2, B_2}}{\vdash A'^{\perp}_1 \otimes A'^{\perp}_2, B_1 \parr B_2}$$

Therefore, for infinitely many finite approximations $A'^{\perp}_1$ and $A'^{\perp}_2$ of A_1 and A_2, $A'^{\perp}_1 \in [\![B_1]\!]_{\mathbf{P}}$ and $A'^{\perp}_2 \in [\![B_2]\!]_{\mathbf{P}}$ hold, which implies

$$[\![A_1]\!]_{\mathbf{P}} \subseteq [\![B_1]\!]_{\mathbf{P}}, [\![A_2]\!]_{\mathbf{P}} \subseteq [\![B_2]\!]_{\mathbf{P}}.$$

The converse also holds by the same argument. Therefore,

$$A_1 \cong_{LL} B_1, A_2 \cong_{LL} B_2$$

hold. From Case 2 and Case 3 for any $A \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} C$, there exists D such that $B \stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Rightarrow} D$ and $C \cong_{LL} D$.

□

Corollary 6.8 $A \cong_{LL} B \Longrightarrow A \cong_1 B$

From the proof of Proposition 6.6, we can see that logical equivalence ($\cong_{LL}$) is just a *structural congruence*. We need the following definition.

Definition 6.1 (Precongruence, Congruence) A relation $\sqsubseteq$ over $\mathcal{NP}$ is a precongruence if for any context $C[\,]$, $A \sqsubseteq B$, $C[A] \in \mathcal{NP}$ and $C[B] \in \mathcal{NP}$ implies $C[A] \sqsubseteq C[B]$, where $C[\,]$ is defined as follows:

$$C[\,] = [\,] \mid C[\,] \parr F \mid F \parr C[\,] \mid \vec{p}^{\perp} \otimes (\vec{q} \parr C[\,]) \mid C[\,] \oplus F$$

F ranges over $\mathcal{NP}$ and $C[X]$ represents a formula obtained by substituting X for $[\,]$. A precongruence is a *congruence* if it is symmetric.

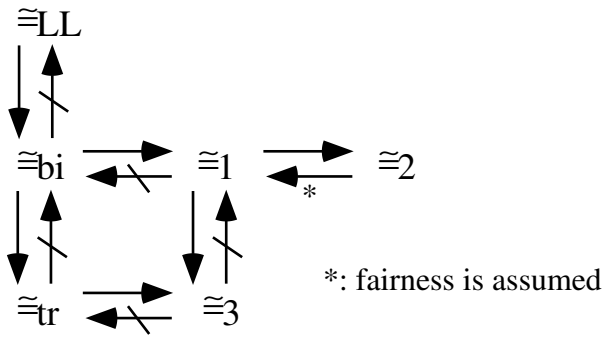

Figure 2: Relationship between Equivalence Relations

Then, $\cong_{LL}$ is a congruence which satisfies the following equations for processes in $\mathcal{NP}$.

commutativity

1. $A \parr B \cong_{LL} B \parr A$
2. $A \oplus B \cong_{LL} B \oplus A$
3. $m^{\perp} \otimes n^{\perp} \otimes A \cong_{LL} n^{\perp} \otimes m^{\perp} \otimes A$

associativity

1. $(A \parr B) \parr C \cong_{LL} A \parr (B \parr C)$
2. $(A \oplus B) \oplus C \cong_{LL} A \oplus (B \oplus C)$
3. $m^{\perp} \otimes (n^{\perp} \otimes A) \cong_{LL} (m^{\perp} \otimes n^{\perp}) \otimes A$

recursive process

1. $A \cong_{LL} B$ if $A \circ\!\!- B \in \mathbf{P}$.
2. Let $\sqsubseteq_{LL}$ be the smallest precongruence which includes $\cong_{LL}$ and satisfies $\mathbf{0} \sqsubseteq_{LL} A$ for all A. Then, $A \cong_{LL} B$ if for any finite approximation A' of A there is a finite approximation B' of B and $A' \sqsubseteq_{LL} B'$, and vice versa.

The following proposition implies that logical equivalence is strictly finer than testing equivalences.

Proposition 6.9 $\cong_{LL} \neq \cong_1$

Proof It is enough to show that $\cong_{LL} \neq \cong_{bi}$. Let A be $(p^{\perp} \otimes q) \parr (r^{\perp} \otimes s)$ and B be $(p^{\perp} \otimes (q \parr (r^{\perp} \otimes s))) \oplus (r^{\perp} \otimes ((p^{\perp} \otimes q) \parr s))$. Then, $A \cong_{bi} B$ but $A \ncong_{LL} B$.
□

The diagram on the relationships between equivalence relations is shown in Figure 2. $\cong \rightarrow \cong'$ means that $\cong$ is finer than $\cong'$. Although $\cong_{LL}$ is the finest equivalence relation for normal processes, it is not true in the case of $\mathcal{P}^{-}$, as is mentioned in Section 3.

7 Unfold/Fold Program Transformation

We think that $\cong_1$ is the most suitable one for a basis of program transformation among the above equivalence relations, because it well captures observational behavior of processes, and also because its definition can be easily extended to full ACL[6][7] including modal messages, value passing, and name creation (on the other hand, the definition of bisimulation equivalence would become much more complicated).

In this section, we introduce unfold/fold program transformation for the core-ACL, and prove its soundness with respect to the equivalence relation $\cong_1$. The following definition is necessary to state our transformation rule.

Definition 7.1 (Guarded subformula) Let $A, B \in \mathcal{P}^-$. If $\vec{p}^{\perp} \otimes (\vec{q} \parr A)$ is a subformula of B for some (nonempty) multisets $\vec{p}, \vec{q}$, then A is called a *guarded subformula* of B.

Proposition 7.1 (Unfold/Fold Transformation) Let $\mathbf{P}$ be a program, $G \in \mathcal{P}^-$, and A a new predicate which does not appear in $\mathbf{P}$. Transform G to its equivalent process G' w.r.t. $\cong_1$ under a program $\mathbf{P}$. If some guarded subformula of G' matches G, replace it by A and obtain the resulting formula G''. Then, $G \cong_1 A$ under a program $\mathbf{P} \cup \{A \circ\!\!- G''\}$.

Proof sketch Let $G''[X/A]$ be a formula obtained from G'' by substituting A with X, and

$$\mathbf{P}' = \mathbf{P} \cup \{A \circ\!\!- G''\} \cup \{A_{n+1} \circ\!\!- G''[A_n/A] \mid n \geq 0\} \cup \{A_0 \circ\!\!- G\}$$

From the definition of transformation, $G''[G/A] = G'$ and $G' \cong_1 G$ hold, which implies $A_1 \cong_1 A_0$. Then by induction on n, $A_n \cong_1 A_0 \cong_1 G$ under a program $\mathbf{P}'$ for any n. On the other hand, A and A_n behaves the same at least until n transitions. Because n is an arbitrary number, we have $A \cong_1 A_n \cong_1 G$ under a program $\mathbf{P}'$. Since neither A nor G requires the clauses on A_n, $A \cong_1 G$ under a program $\mathbf{P} \cup \{A \circ\!\!- G''\}$. □

Example 7.1 Let $\mathbf{P} = \{A \circ\!\!- p^{\perp} \otimes (q \parr A), B \circ\!\!- r^{\perp} \otimes (s \parr B)\}$. A repeatedly receives a message p and sends a message q, while B repeatedly receives a message r and sends a message s. In order to compose A and B, define a new predicate by $C \circ\!\!- A \parr B$.

$$\begin{aligned} A \parr B &\cong_1 (p^{\perp} \otimes (q \parr A)) \parr (r^{\perp} \otimes (s \parr B)) \\ &\cong_1 (p^{\perp} \otimes (q \parr A \parr (r^{\perp} \otimes (s \parr B)))) \oplus (r^{\perp} \otimes (s \parr (p^{\perp} \otimes (q \parr A)) \parr B)) \\ &\cong_1 (p^{\perp} \otimes (q \parr A \parr B) \oplus (r^{\perp} \otimes (s \parr A \parr B))) \end{aligned}$$

By replacing $A \parr B$ by C, we obtain a formula $(p^{\perp} \otimes (q \parr C)) \oplus (r^{\perp} \otimes (s \parr C))$. Therefore, $A \parr B \cong_1 C$ under a program $\mathbf{P} \cup \{C \circ\!\!- (p^{\perp} \otimes (q \parr C)) \oplus (r^{\perp} \otimes (s \parr C))\}$.

Our transformation procedure can be naturally extended to the full ACL[6]. Although the above example is trivial, we will find interesting transformations in the full ACL.

8 Conclusion

This paper defines and investigates several notions of equivalence relations, namely, logical equivalence, three testing equivalences, bisimulation equivalence, and trace equivalence for processes in a linear logic programming called core-ACL. Strong testing equivalence seems most suitable among those investigated here. We introduced unfold/fold transformation and proved its soundness with respect to the testing equivalence. Because process equivalence in many (asynchronous) concurrent programming paradigms can be derived by restricting testers in testing equivalence of ACL (see encoding of the Actor, Linda, etc. into ACL given in [6]), the soundness of the program transformation rule in ACL implies that of the corresponding rule in those paradigms. It indicates that some program transformation procedure developed in ACL can be commonly used in other those paradigms. We can devise common program transformation procedures as a first step, then refine them or invent new ones according to specific features of each concrete programming language.

References

[1] Agha, G., I. A. Mason, S. Smith, and C. Talcott, "Towards a Theory of Actor Computations," in *Proceedings of CONCUR'92*, vol. 630 of *Lecture Notes in Computer Science*, pp. 565–579, 1992.

[2] Andreoli, J.-M., "Logic Programming with Focusing Proofs in Linear Logic," *Journal of Logic and Computation*, vol. 2, no. 3, 1992.

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

[4] Gelernter, D., "Generative Communication in Linda," *ACM Transactions on Programming Language and Systems*, vol. 7, no. 1, pp. 80–112, 1985.

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

[6] Kobayashi, N., "Asynchronous Communication Model Based on Linear Logic," Master's thesis, Department of Information Science, University of Tokyo, March 1993. (preliminary version is available as Tech. Rep. 92-5, Department of Informatin Science, University of Tokyo).

[7] Kobayashi, N., and A. Yonezawa, "ACL – A Concurrent Linear Logic Programming Paradigm," in *Proceedings of International Logic Programming Symposium (to appear)*, MIT Press, 1993.

[8] 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.

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

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

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

[12] Saraswat, V., and P. Lincoln, "Higher-order, linear, concurrent constraint programming," 1992. draft.

Appendix A Summary of Notations

The following table summarizes notations used in this paper.

Notation	Brief description	Definition
$\mathcal{P}$	a set of goal formulas	Section 2.2.5
$\mathcal{PC}$	a set of process configurations	Section 2.2.5
Success	a successful state	Section 2.3
$\mathcal{PC}^+$	$\mathcal{PC} \cup \{\mathbf{Success}\}$	Section 2.3
$\longrightarrow$	transition relation	Section 2.3
$\mathcal{P}^-$	a subset of $\mathcal{P}$	Section 2.4
$\mathcal{NP}$	a set of normal processes	Section 2.5
$[\![\bullet]\!]_{\mathbf{P}}$	a map from $\mathcal{P}$ to $\mathcal{PC}$	Section 3
$\mathcal{CP}$	a set of coprocesses	Section 3
$[\![\bullet]\!]'_{\mathbf{P}}$	a map from $\mathcal{P} \cup \{\mathbf{Success}\}$ to a power set of $\mathcal{CP}$	Section 3
$\cong_{LL}$	logical equivalence	Section 3
$\cong_1, \cong_2, \cong_3$	testing equivalence	Section 4
$\cong_{tr}$	trace equivalence	Section 5
$\cong_{bi}$	bisimulation equivalence	Section 5
$\stackrel{\downarrow\vec{p}\uparrow\vec{q}}{\Longrightarrow}$	labelled interaction	Section 5