修 士 論 文

分散コミュニケーションのための Pi-calculus の拡張とその抽象機械の提案

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

福田 博之

2002 年 3 月

修 士 論 文

分散コミュニケーションのための Pi-calculus の拡張とその抽象機械の提案

指導教官　大堀淳 教授

審査委員主査　大堀淳 教授
審査委員　田島敬史 助教授
審査委員　二木厚吉 教授

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

010098 福田 博之

提出年月: 2002 年 2 月

要旨

現在，並行プログラミング言語の計算モデルとして Pi-calculus が注目されている．そして，Pict という純粋に Pi-calculus に基づく言語が開発された．しかし，Pict の動作はローカルな環境に限定されており，分散環境への実装が今後の課題となっている．本研究では，分散環境へ実装可能な Pi-calculus の拡張モデルとその抽象機械を提案した．これによって，分散コミュニケーションを明確に形式化することができるようになった．さらに，その拡張モデルに基づくことによって，分散コミュニケーションを容易に，かつ柔軟に記述することができる言語システムを実装することができた．

目 次

第1章　序論

1.1　研究の背景及び目的

現在，並行プログラミング言語の計算モデルとしてPi-calculus [1] が注目されている．Pi-calculusは1990年にMilner，Parrow,Walkerによって提案された．この計算モデルは，並行動作するプロセス間でのチャネルを介したコミュニケーションを形式化する．また，チャネルによるチャネルの伝達を許すことで，モバイリティの要素を導入し，動的にコミュニケーショントポロジーを変化させる並行システムの仕様の記述と正当性の検証を可能にする．さらに，チャネルによるモバイリティは，表現力を著しく増大させ，簡潔な意味論と扱いやすい代数理論を伴って，Pi-calculusを高水準な並行性を幅広く記述することができる計算モデルへと導いた．

一方で，λ-calculusは簡潔さと表現力を伴って，さかんに理論的な研究が行われ，逐次的な関数型言語に対する基礎理論となった．それならば，Pi-calculusからどのような高級言語が構築されるのか興味がひかれるところである．そのような動機によって，Pi-calculusに基づく言語のあり方を探るべく実験的に設計された言語としてPict [2] がある．

Pictは，PierceとTurnerによって1992年から開発された．Pictは，計算の唯一のメカニズムとしてコミュニケーションを利用し，純粋にPi-calculusに基づく言語と言える．これまでのPi-calculusを用いた言語は，関数型言語にPi-calculusライクなコミュニケーションを組み合わせたものであり，その点でPictは大きく異なる．Pictは，Turnerが提案した抽象機械 [3] を採用することで，Pi-calculusでの基本的な計算メカニズムであるチャネルを介したコミュニケーションを効率よく実装している．しかし，Turnerの抽象機械はローカルな環境でのコミュニケーションを実現するものであり，分散環境への実装が今後の研究課題となっている．

そこで本研究では，Pi-calculusの意味論について考察し，分散環境へ実装する上でどのような問題が起こるのかを明らかにし，その解決方法を探る．そして，Pi-calculusを拡張した分散環境へ実装可能な計算モデルを提案し，Turnerの抽象機械を拡張して，その計算モデルの実装を試みる．

この研究によって期待される効果は，今日の大規模，複雑化する傾向にある分散アプリケーションの動作の正当性や安全性の検証が可能になるということである．なぜなら，計算モデルに基づいて言語を設計することによって，その言語によって書かれたプログラムの意味付けに対して数学的な基礎を与えることができるからである．これによって，プログラムの振る舞いの特性を証明することができるようになり，プログラムの信頼性の向上につながる．そのため，プログラムの基礎理論に対する研究は，今後ますますその重要性を増す．特に，分散・並行プログラミング言語に対する基礎理論はまだまだ不十分であり，今後の大きな研究課題である．

1.2　関連研究

現在に至るまでにPi-calculusを拡張した分散モデルは数多く提案されているが，その中の代表的なものとして，Distributed Join-calculus[5] とMobile Ambients[6] がある．

Distributed Join-calculus は，非同期な Pi-calculus の拡張である join-calculus に基づいた並行分散言語システムである．このモデルでは，プロセスは非同期な出力を行うだけで，プロセスとは別にインタラクションを定義するルールが存在する．プロセスはこのルールに従って，置き換えられていくことで，消費され，コミュニケーションが行われる．分散環境では，プロセスはメッセージとしてそのプロセスのインタラクションを定義しているサイトへ移動することができる．これによって非同期な分散コミュニケーションを実現する．また，ネットワークノードなどのプロセスが動作する局所的な環境を抽象した” ロケーション” という概念を備えている．ロケーションは木構造を持ったまま別のサイトへ移動可能であり，これによって移動エージェントを表現している．さらに，ロケーションの故障・停止も表現できる．この計算モデルの抽象機械は，ルールとプロセスの集合によって構成され，可逆の構造ルールを持つケミカルスタイルで仕様が与えられている．

Mobile Ambients は，計算が行われる境界域である ambient という概念を持つ領域間移動プロセス計算モデルである．ambient は名前を持ち，プロセスをそれ自身に包括する構造である．この ambient に対して侵入，退出，解放といった capability と呼ばれる演算を行うことで，操作的意味論が与えられる．複雑なネットワーク構造やモバイル計算の動作を理解するために用いられる．

これらの計算モデルは，プロセス移住による移動計算を主体としており，本研究で提案するメッセージパッシングを主体とした計算モデルとは異なる．

1.3 本稿の構成

まず第 2 章において，本研究の基礎理論となる Pi-calculus について説明し，さらにその実装を与える Turner の抽象機械について説明する． 3 章では，Pi-calculus を分散環境へ実装可能なモデルへと拡張し，その操作的意味論について述べた後，その実装を与えるための Tuner の抽象機械の拡張について述べる． 4 章では， 3 章で提案したモデルに基づく言語システムを実装する方式について述べる．最後に 5 章において，本研究の結論と今後の課題について述べる．

第2章　Pi-calculus

本章では，Pi-calculus の構文，操作的意味論について説明した後，その実装を与える Turner の抽象機械 [3] について説明する．

2.1　構文

Pi-calculus の構文は以下のように定義される．

$P, Q, R, S ::=$	$P \mid Q$	Parallel composition
	$(\nu x)P$	Restriction
	$x?(y).P$	Input
	$x!y.P$	Output
	$x?*(y).P$	Replication
	0	Nil

Parallel composition は，プロセス P,Q が並行に動作することを表す．Restriction は，チャネル x をプロセス P のスコープ内に限定することを表す．Input は，入力チャネル x から値またはチャネル名を y で受信することを表す．Output は，出力チャネル x へ引数 y を送信することを表す．Replication は，プロセス $x?(y).P$ を複製することを表す．Nil は，停止したプロセスを表す．一般的に入出力の後に 0 が続く場合は省略する．また，これらの演算の優位性は次のように定義されている．

$$\text{Input/Output} < \text{Replication} < \text{Parallel composition} < \text{Restriction}$$

2.2　操作的意味論

ここでは，Pi-calculus の動作や性質を記述する操作的意味論について述べる．

2.2.1　簡約

Pi-calculus の簡約は以下のように定義される．

$$\text{COM:} \quad x?(y).P|x!z.Q \rightarrow P\{z/y\}|Q$$

$$\text{PAR:} \quad \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

$$\text{RES:} \quad \frac{P \rightarrow P'}{(\nu x)P \rightarrow (\nu x)P'}$$

$$\text{STRUCT:} \quad \frac{Q \equiv PP \rightarrow P'P' \equiv Q'}{Q \rightarrow Q'}$$

COM は，入力プロセスと出力プロセスの間で行われるコミュニケーションを定義している．PAR は，並行構成における簡約を定義している．RES は，制限における簡約を定義している．STRUCT は，構文的に一致する動作式集合を導入した簡約を定義している．

2.2.2 構造合同性

構造的に一致する動作式集合は以下のように定義される．

(1) $P \equiv Q$ whenever P is alpha-convertible to Q
(2) $P|0 \equiv P, P|Q \equiv Q|P, P|(Q|R) \equiv (P|Q)|R$
(3) $*P \equiv P|*P$
(4) $(\nu x)0 \equiv 0, (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$
(5) $(\nu x)(P|Q) \equiv P|(\nu x)Q$ if x not free in P

(1) は，P が Q に α 変換可能であるならば，P と Q は構造合同であることを表す．(2) は，並行構成の可換則と結合則と Nil を含む場合の構造合同を定義している．(3) は，複製に対する構造合同を定義している．(4) は，制限に対する構造合同を定義している．(5) は，制限におけるスコープ拡大に対する構造合同を定義している．

2.2.3 基本的な動作式例

以下に Pi-calculus の基本的な例を示す．ここでは，プロセス間のチャネルを介したコミュニケーションをリンク構造で表現している．

Example 1: Link passing

P は R へのリンク x を持ち，Q へのリンク y を通して x を渡す．Q はそれを受け取る．よって，P は $y!x.P'$ となり，Q は $y?(z).Q'$ となる．この場合の遷移は以下のようになる．

$$y!x.P'|y?(z).Q'|R \xrightarrow{t} P'|Q'\{x/z\}|R$$

図 2.1 は $x \notin fn(Q)$ である場合を表し，遷移前では Q はリンク x を持たない．しかし，$x \in fn(Q)$ であったとしても，遷移は同じである．すでに存在するリンクを受け取るべきではないという理由はない．また，$x \notin fn(P')$ の場合，P' は遷移の後，リンク x を持たない．この条件も遷移には影響しない．

P と Q 間のリンク y が P と Q のそれぞれに制限されるとき，状況は変わる．この場合の遷移図は図 2.2 のようになる．

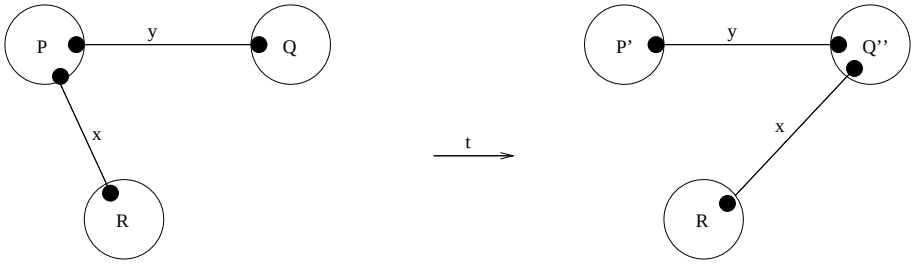

図 2.1: Link passing 1

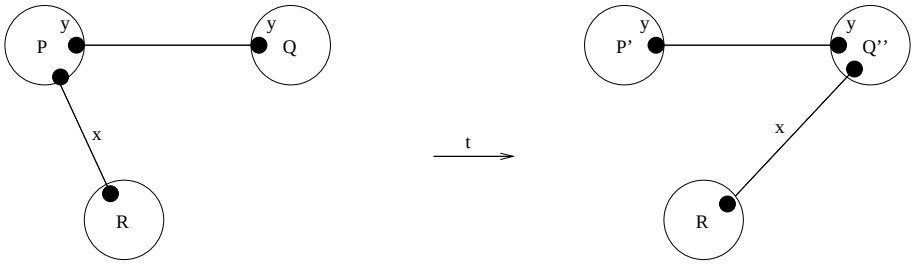

図 2.2: Link passing 2

リンク y のプライバシーは制限によって表現され，遷移は以下のようになる．

$$(\nu y)(y!x.P'|y?(z).Q')|R \xrightarrow{t} (\nu y)(P'|Q'\{x/z\})|R$$

実際には，以下の等価性を証明することができる．

$$(\nu y)(y!x.P'|y?(z).Q') = \tau.(\nu y)(P'|Q'\{x/z\})$$

Example 2: Scope intrusion

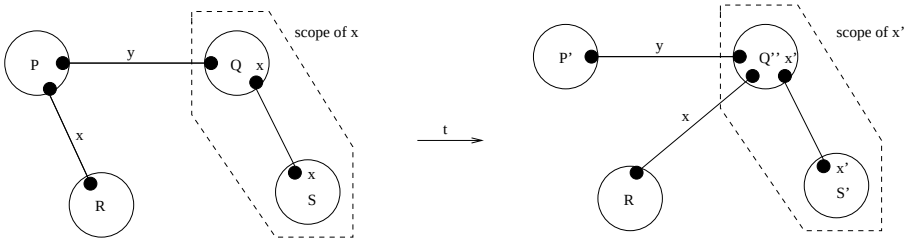

図 2.3: Scope intrusion

Example 1 のように P は R へのリンク x を持ち，Q へリンク y を通して x を渡す．Q はそれを受け取るが，すでに S へのプライベートなリンク x をすでに持っている．P を $y!x.P'$，Q を $y?(z).Q'$ とすると，遷移は以下のようになる．

$$y!x.P'|R|(\nu x)(y?(z).Q'|S) \xrightarrow{t} P'|R|(\nu x')(Q'\{x'/x\}\{x/z\}|S\{x'/x\})$$

図 2.3 の Q" と S' は，それぞれ Q'{x'/x}{x/z} と S{x'/x} であり，x' は新しい名前である．これは，λ-calculus の α 変換に相当し，以下の $alpha$ 等価性が成り立つ．

$$(\nu x)(Q|S) = (\nu x')(Q\{x'/x\}|S\{x'/x\})$$

Example 3: Scope extrusion

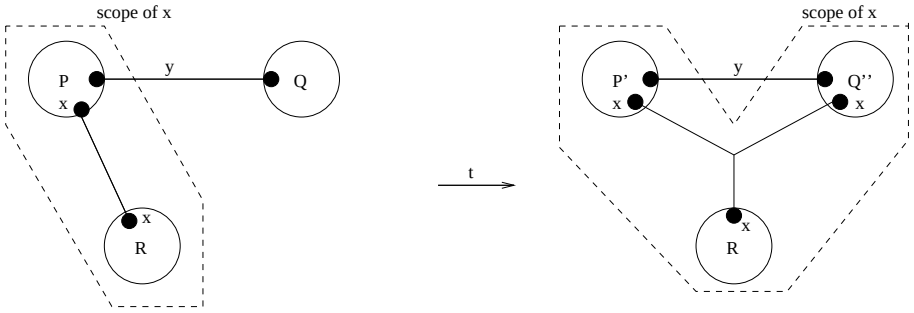

図 2.4: Scope extrusion 1

P は,Example 1 のように R へのリンク x を持つが，ここでは，このリンクは P と R で制限されている．しかし，P は Q へリンク y を通して x を渡す．Q はそれを受け取る．P を $y!x.P'$，Q を $y?(z).Q'$ とすると，遷移は以下のようになる．

$$(\nu x)(y!x.P'|R)|y?(z).Q' \xrightarrow{t} (\nu x)(P'|R|Q'\{x/z\})$$

Example 1 のように Q" は Q'{x/z} である．ここでの違いは，P の R へのリンク x のプライバシーである．このリンクは Q へ伝達され，制限のスコープは拡張される．

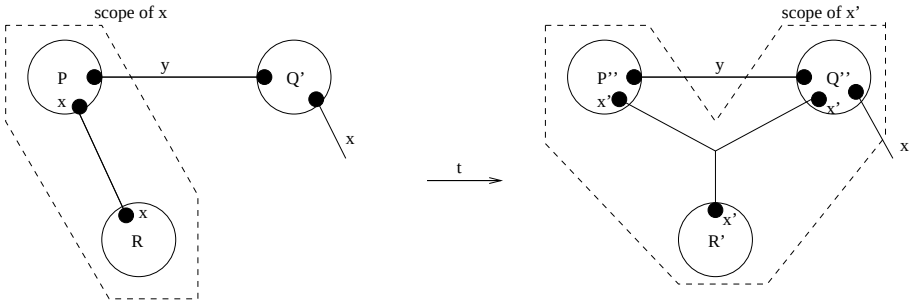

図 2.5: Scope extrusion 2

Example 1 とは対照的に，Q が遷移前にパブリックなリンク x を持つならば，その状況は異なる．その時，パブリックなリンクとの違いを保存するためにプライベートなリンクの名前を変えなければならない．その遷移は以下のようになる．

$$(\nu x)(y!x.P'|R)|y?(z).Q' \xrightarrow{t} (\nu x')(P'\{x'/x\}|R\{x'/x\}|Q'\{x'/z\})$$

図 2.5 の P" は P'{ x'/x },Q" は Q'{ x'/z },R' は R{ x'/x } である．

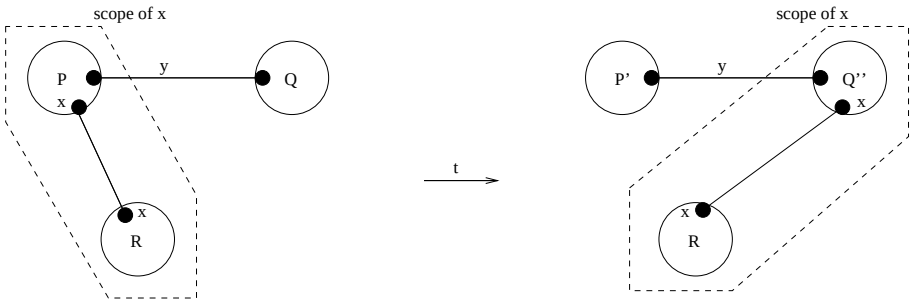

図 2.6: Scope extrusion 3

$x \notin fn(Q)$ の場合に戻り，$x \notin fn(P')$ とすると，遷移図は図 2.6 のようになる．この遷移は図 2.4 と同じであるが，結果を次の法則によって変換できる．

$$(\nu x)(P_1|P_2) = P_1|(\nu x)P_2 \quad \text{if } x \notin fn(P_1)$$

これを図 2.4 の結果に適用すると，以下のようになる．

$$(\nu x)(P'|R|Q'\{x/z\}) = P'|(\nu x)(R|Q'\{x/z\})$$

これは，extrusion の特別な場合であり，スコープの移住と呼ぶ．制限 x のスコープは P から Q へ移住したことになる．

Example 4: Molecular actions

プロセス P が 2 つのプロセス Q か R に名前のペア (u, v) を渡したいとき，次の 3 つのプロセスでその試みを考えてみる．

$$\begin{aligned} P &\equiv x!u.x!v.P' \\ Q &\equiv x?(y).x?(z).Q' \\ R &\equiv x?(y).x?(z).R' \end{aligned}$$

このような 3 つのプロセスの $Q|P|R$ を考えると，Q,R が (u,v) のどちらか一方しか受け取らない場合が考えられる．

$$\begin{aligned} Q|P|R &\equiv x?(y).x?(z).Q'|x!u.x!v.P'|x?(y).x?(z).R' \\ &\xrightarrow{t} x?(z).Q'\{u/y\}|x!v.P'|x?(y).x?(z).R' \\ &\xrightarrow{t} x?(z).Q'\{u/y\}|P'|x?(z).R'\{v/y\} \end{aligned}$$

これを解決するには，u と v を直接送らないで，u と v を送るためだけのプライベートなチャネルを Q または R へ送る．

$$\begin{aligned} P &\equiv (\nu w)(x!w.P'|w!u.w!v.0) \\ Q &\equiv x?(w).w?(y).w?(z).Q' \\ R &\equiv x?(w).w?(y).w?(z).R' \end{aligned}$$

$w \notin fn(P', Q', R')$ であり，$Q|P|R$ には2つの遷移が考えられる．

$$\begin{aligned} Q|P|R &\xrightarrow{t} (\nu w)(w?(y).w?(z).Q'|P'|w!u.w!v.0)|R \\ &= \tau.\tau.(Q'\{u/y\}\{v/z\}|P'|R) \end{aligned}$$

$$\begin{aligned} Q|P|R &\xrightarrow{t} Q|(\nu w)(P'|w!u.w!v.0|w?(y).w?(z).R') \\ &= \tau.\tau.(Q|P'|R'\{u/y\}\{v/z\}) \end{aligned}$$

2つの遷移は Q と R へのペアの伝達を表現している．

2.3 Pi-calculus に対する抽象機械

ここで説明する抽象機械は，Turner によって提案されたものであり，Pict [2] で採用されている．

2.3.1 機械の状態

この抽象機械はヒープ (Heap) と実行キュー (Run queue) から構成される．ヒープには，チャネル名に対応するチャネルキュー (Channel queue) が格納される．チャネルキューは，この抽象機械における重要な構成要素である．ここに出力，入力，複製プロセスが一時的に格納され，実行キューの先頭にあるプロセスとそのチャネルにおいてコミュニケーションが行われることを待つ．実行キューには，並行動作している実行可能なプロセスが格納されている．以下に機械の状態を定義する．

(Channel queues)

$C ::=$	$S_1 :: ... :: S_n$	Channel queue
$S ::=$	$?(x).P$	Reader
	$!x.P$	Writer
	$?*(x).P$	Replicated reader

(Machine state components)

$H ::=$	$x_1 \mapsto C_1, ..., x_n \mapsto C_n$	Heap
$R ::=$	$P_1 :: ... :: P_n$	Run queue

C はチャネルキューを表し，そこに S で表現されるプロセス (Reader,Writer,Replicated reader) が格納される．H はヒープを表し，そこに $x \mapsto C$ で対応関係が表現されたチャネル名 x とチャネルキュー C が格納される．R は実行キューを表し，そこに P で表現されるプロセスが格納される．

2.3.2 リダクションルール

この抽象機械は，$H, R \to H', R'$ の形式のリダクションルールの集合によって定義される．それぞれのルールは，実行キューの先頭のプロセスを参照して，そのプロセスの1ステップのリダクションを実行す

る．実行キューが空であるならば，$H, R \nrightarrow$ であり，実行は終了する．ここで，rs は入力プロセスのキューを表し，ws は出力プロセスのキューを表す．$bullet$ はキューが空であることを表す．$H\{c \mapsto C\}$ は，ヒープ H においてチャネル c に対応したチャネルキューが C に更新されることを表す．

- Nil:
  nil プロセスは無動作であり，ただ実行キューから削除し，次のプロセスを実行可能にする．

$$\frac{}{H, \mathbf{0} :: R \rightarrow H, R}$$

- Prl:
  並行構成 $P|Q$ は非対称な手法で解釈する．プロセス Q を実行キューの後ろに置き，P の実行に続く．

$$\frac{}{H, (P|Q) :: R \rightarrow H, P :: R :: Q}$$

- Res:
  制限オペレーター $(\nu x)P$ は，ヒープに新しいチャネル c を割り当て，c を束縛変数 x に代入する．そして，P へと実行が続く．新しいチャネルは，初期には空である．

$$\frac{cfresh}{H, (\nu x)P :: R \rightarrow H\{c \mapsto \bullet\}, \{c/x\}P :: R}$$

- Inp-W:
  入力プロセス $c?(x).P$ を実行するとき，チャネル c がそのキューにすでにブロックされている出力プロセスを持っているならば，最初にチャネルキューから出力プロセスを削除し，P において束縛変数 x に対して a を代入する．そして，Q を実行キューの最後に置き，P の実行へと続く．

$$\frac{H(c) = !a.Q :: ws}{H, c?(x).P :: R \rightarrow H\{c \mapsto ws\}, \{a/x\}P :: R :: Q}$$

- Inp-R:
  入力プロセス $c?(x).P$ を実行するとき，チャネル c がそのキューにすでにブロックされている入力プロセスを持っているならば，現在のプロセスを一時停止して，それをチャネルキューの最後に置く．このルールは c に対応するチャネルキューが空である場合を包含する．

$$\frac{H(c) = rs}{H, c?(x).P :: R \rightarrow H\{c \mapsto rs :: ?(x).P\}, R}$$

- Out-R:
  出力プロセス $c!a.P$ を実行するとき，チャネル c がそのキューにすでにブロックされている入力プロセスを持っているならば，最初にチャネルキューから入力プロセスを削除し，Q において束縛変数 x に対して a を代入する．そして，Q を実行キューの最後に置き，P の実行へと続く．

$$\frac{H(c) = ?(x).Q :: rs}{H, c!a.P :: R \rightarrow H\{c \mapsto rs\}, P :: R :: \{a/x\}Q}$$

- Out-W:
  出力プロセス $c!x.P$ を実行するとき，チャネル c がそのキューにすでにブロックされている出力プロセスを持っているならば，現在のプロセスを一時停止して，それをチャネルキューの最後に置く．このルールは c に対応するチャネルキューが空である場合を包含する．

$$\frac{H(c) = ws}{H, c!a.P :: R \rightarrow H\{c \mapsto ws ::!a.P\}, R}$$

- Repl-R:
  複製入力プロセス $c?*(x).P$ を実行するとき，チャネル c がそのキューに入力プロセスのみ持っているならば，複製入力プロセスをそのチャネルキューの最後に置く．

$$\frac{H(c) = rs}{H, c?*(x).P :: R \rightarrow H\{c \mapsto rs ::?*(x).P\}, R}$$

- Repl-W:
  複製入力プロセス $c?*(x).P$ を実行するとき，チャネル c がそのキューにすでにブロックされている出力プロセスを持っているならば，P のコピーを生成して，その P において束縛変数 x に対して a を代入する．そして，Q を実行キューの最後に置き，再び複製入力プロセス $c?*(x).P$ を実行する．このルールはチャネル c から全ての出力プロセスを削除する効果を持つ．

$$\frac{H(c) =!a.Q :: ws}{H, c?*(x).P :: R \rightarrow H\{c \mapsto ws\}, c?*(x).P :: R :: \{a/x\}P :: Q}$$

- Out-R*:
  出力プロセス $c!a.P$ を実行するとき，チャネル c がそのキューに複製入力プロセスを持っているならば，その複製入力プロセスのコピーを実行キューの最後に置き，そのプロセスにおいて x に対して a を代入する．チャネルキューの複製入力プロセスは削除しないが，そのプロセスをチャネルキューの最後に置く．これによって，c の他の入力プロセスを実行することが可能になる．

$$\frac{H(c) =?*(x).Q :: rs}{H, c!a.P :: R \rightarrow H\{c \mapsto rs ::?*(x).Q\}, P :: R :: \{a/x\}Q}$$

2.3.3 動作例

次の例は，プロセス $(\nu x)(x![\,].P|x?[\,].Q)$ がチャネル x を介してどのようにインタラクションが行われるかを例示するものである．最初に，新しいチャネル c を生成して，束縛変数 x に対してそれを代入する．ここでは，単純のため，$x \notin fv(P, Q)$ であるとする．

$$\begin{array}{ll} & H, (\nu x)(x![\,].P|x?[\,].Q) \\ \rightarrow & H\{c \mapsto \bullet\}, c![\,].P|c?[\,].Q \end{array}$$

実行キューに $c![\,].P$ と $c![\,].P$ を置く．

$$\rightarrow \quad H\{c \mapsto \bullet\}, c![\,].P :: c?[\,].Q$$

出力プロセス $c![\,].P$ を実行し，c のキューに $c![\,].P$ を置く．

$\rightarrow \quad H\{c \mapsto ![\,].P\}, c?[\,].Q$

入力プロセス $c?[\,].Q$ を実行し，プロセス P を実行キューの最後に置いて，Q の実行へと続く．

$\rightarrow \quad H\{c \mapsto \bullet\}, Q :: P$

次の 2 つの例は，複製ルールの振る舞いを示す．最初の例は，プロセス $(\nu x)(x![\,].P|x? * [\,].Q)$ のリダクションを示す．単純のため，$x \notin fv(P, Q)$ であるとする．プロセス $c![\,].P$ は，プロセス $c? * [\,].Q$ の前に実行されるので，$c? * [\,].Q$ が実行されるまでチャネル c にブロックされる．

	$H, (\nu x)(x![\,].P|x? * [\,].Q)$	
$\rightarrow$	$H\{c \mapsto \bullet\}, c![\,].P|c? * [\,].Q$	Res
$\rightarrow$	$H\{c \mapsto \bullet\}, c![\,].P :: c? * [\,].Q$	Prl
$\rightarrow$	$H\{c \mapsto ![\,].P\}, c? * [\,].Q$	Out-W
$\rightarrow$	$H\{c \mapsto \bullet\}, c? * [\,].Q :: P :: Q$	Repl-W
$\rightarrow$	$H\{c \mapsto ? * [\,].Q\}, P :: Q$	Repl-R

次の例は，先程の逆で $c![\,].P$ が実行される前に $c? * [\,].Q$ が実行される場合である．そのため，$c![\,].P$ を実行するとき，Q のコピーを生成するために Out-R*を使うことができる．前の例のようにプロセス P を一時停止する必要はない．

	$H, (\nu x)(x? * [\,].Q|x![\,].P)$	
$\rightarrow$	$H\{c \mapsto \bullet\}, c![\,].P|c? * [\,].Q$	Res
$\rightarrow$	$H\{c \mapsto \bullet\}, c? * [\,].Q :: c![\,].P$	Prl
$\rightarrow$	$H\{c \mapsto ? * [\,].Q\}, c![\,].P$	Repl-R
$\rightarrow$	$H\{c \mapsto ? * [\,].Q\}, P :: Q$	Out-R*

第3章　分散コミュニケーションのための拡張

本章では，2章で説明したPi-calculus [1] を拡張した分散環境へ実装可能な新しい計算モデルを提案し，その操作的意味論について述べた後，その実装を与えるためのTurnerの抽象機械 [3] の拡張について述べる．

3.1　問題点及び解決策

Pi-calculusを実装する上で重要な事は，この計算モデルでの基本的な計算メカニズムであるコミュニケーションをいかに効率よく実装するかという事である．Pi-calculusでは，コミュニケーションは以下の簡約によって定義される．

$$x!a.P|x?(y).Q \rightarrow P|\{a/y\}Q$$

この簡約では，同一チャネルに対する入力プロセスと出力プロセスが現れたとき，値またはチャネルの送受信が行われるということを表している．ここで，この簡約を分散環境で実装する事を考える．この簡約は，見ての通り並行動作している2つのプロセスの状態に依存している．もし，この2つのプロセスが別々のサイトに存在していた場合，コミュニケーションを行うには，2つのサイトの状態を把握しておく必要があるということになる．分散環境では，不特定多数のサイトが存在し，しかもサイトは動的に起動，停止されるものであり，一時的に存在するものも含め，これら全ての状態を把握するのは困難である．つまり，上の簡約を分散環境で効率よく実装するのは不可能と言える．

そこで，同期通信のメカニズムを導入して，他のサイトの状態に依存しない簡約によって分散コミュニケーションを実現する．例えば，サイトAにプロセス $x?(y)$, サイトBにプロセス $x!a.Q$ が存在し，チャネルxはサイトBに存在しているとする．このとき，まず最初にプロセスPは自身のプリフィックス $x?(y)$ をサイトBへと送る．それから，ユニークに設定された名前を持つチャネルiにおいて入力待ちになることでロックされ，待機状態になる．一方，サイトBでは，受け取ったプリフィックス $x?(y)$ とプロセスQがチャネルxにおいてコミュニケーションする．そして，プリフィックスは，そのコミュニケーションの結果受け取った値aをサイトAへと返す．サイトAは，結果の値を受け取ると，チャネルiへ結果の値aを出力するプロセス $i!a$ を生成する．そして，プロセス $i?(y).P$ とプロセス $i!a$ においてコミュニケーションが行われ，プロセスPは実行可能状態となり，値aをプロセスQから受け取ったことになる．以上を図で表すと，図3.1のようになる．このように同期通信のメカニズムを導入することによって，実際のチャネルを介したコミュニケーションはあるサイト内に限定され，Pi-calculusの特性をそのままにして，分散コミュニケーションを実現することができる．

3.2　Pi-calculusの拡張

ここでは，以上で提案した分散コミュニケーションを実現するためのPi-calculusの拡張について述べる．

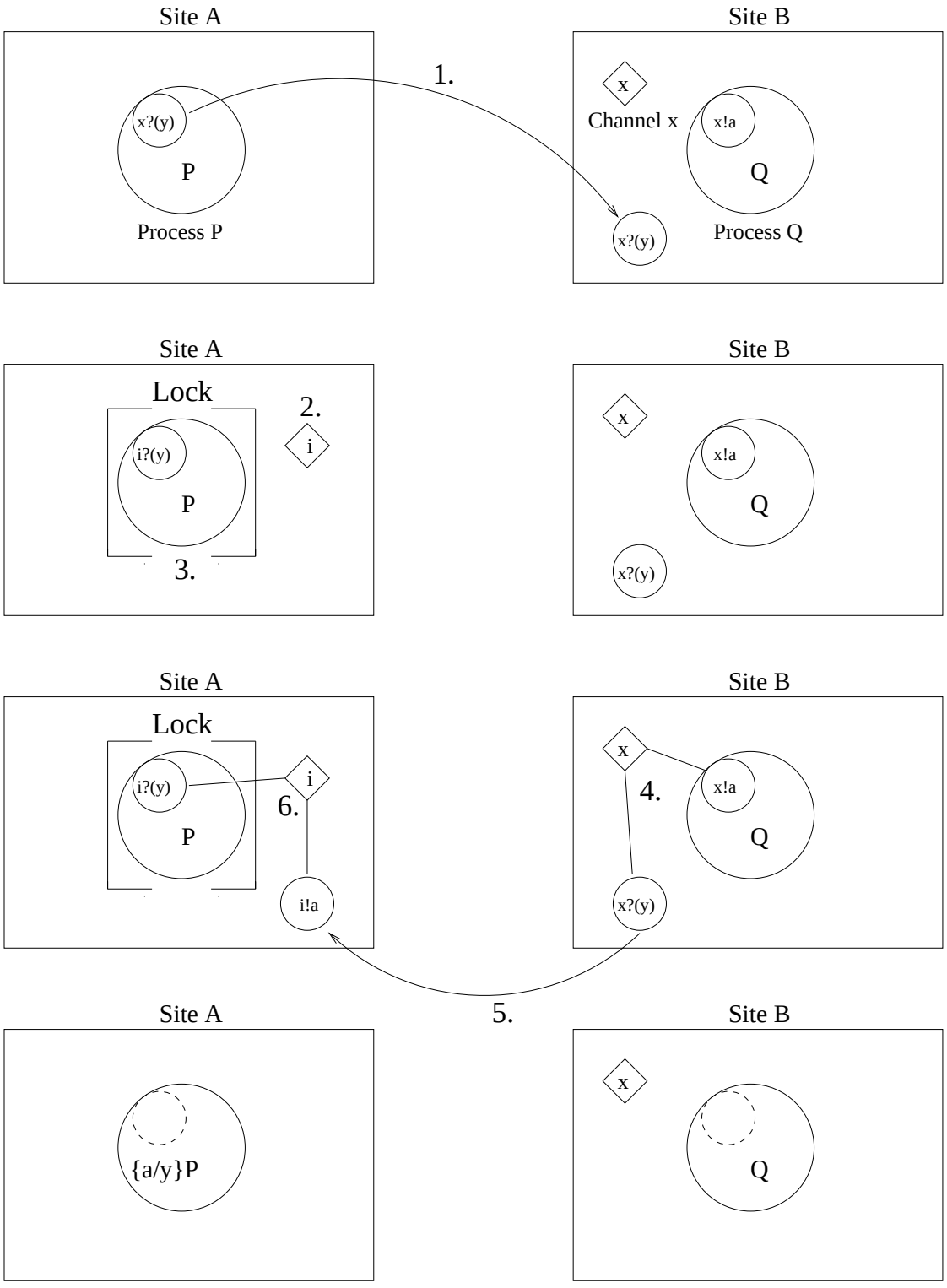

図 3.1: 同期通信による分散コミュニケーションの実現

3.2.1 構文

Pi-calculus の構文は以下のように拡張する．

$$
\begin{array}{lll}
P, Q, R, S ::= & \ldots & \\
& x@l?(y).P & \text{Remote Input} \\
& x@l!y.P & \text{Remote Output} \\
& x@l?*(y).P & \text{Remote Replication} \\
& i@l \Leftarrow v & \text{Return}
\end{array}
$$

Remote Input は，サイト l のチャネル x から値またはチャネル名を y で受信することを表す．Remote Output は，サイト l のチャネル x へ引数 y を送信することを表す．Remote Replication は，プロセス $x@l?(y).P$ を複製することを表す．Return は，遠隔サイトにおいて受信した値またはチャネル名 v をサイト l のチャネル i へ出力することを表す．これが同期通信による分散コミュニケーションにおいて重要な役割を果たす．

3.2.2 簡約

Pi-calculus の簡約は以下のルールによって拡張される．ここで，$[\,]_l$ は，l という名前のサイトを表し，その中にプロセスを包括する．$\|$ は，サイトを分割し，それぞれのサイトは分散環境上の任意の場所に存在することを表す．

$$\text{RCOM1:} \quad [x@l_2?(y).P]_{l1} \| [Q]_{l2} \rightarrow [i?(y).P]_{l1} \| [Q|x?(y).i@l_1 \Leftarrow y]_{l2}$$

$$\text{RCOM2:} \quad [x@l_2!a.P]_{l1} \| [Q]_{l2} \rightarrow [i?[\,].P]_{l1} \| [Q|x!a.i@l_1 \Leftarrow [\,]]_{l2}$$

$$\text{RCOM3:} \quad [x@l_2?*(y).P]_{l1} \| [Q]_{l2} \rightarrow [i?*(y).P]_{l1} \| [Q|x?*(y).i@l_1 \Leftarrow y]_{l2}$$

$$\text{RETURN:} \quad [P]_{l1} || [i@l_1 \Leftarrow y|Q]_{l2} \rightarrow [P|i!y]_{l1} \| [Q]_{l2}$$

RCOM1 は，遠隔入力プロセス $x@l_2?(y).P$ が現れたときの簡約を定義している．このとき，プリフィックス $x?(y)$ はリターンプロセス $i@l_1 \Leftarrow y$ を伴って，サイト l_2 へと送られる．そして，ユニークに設定されたチャネル i においてプロセス $?(y).P$ は待機状態になる．. RCOM2 は，遠隔出力プロセス $x@l_2!a.P$ が現れたときの簡約を定義している．このとき，プリフィックス $x!a$ はリターンプロセス $i@l_1 \Leftarrow [\,]$ を伴って，サイト l_2 へと送られる．そして，ユニークに設定されたチャネル i においてプロセス $?[\,].P$ は待機状態になる．RCOM3 は，遠隔複製プロセス $x@l_2?*y.P$ が現れたときの簡約を定義している．このとき，プリフィックス $x?*(y)$ はリターンプロセス $i@l_1 \Leftarrow y$ を伴って，サイト l_2 へと送られる．そして，ユニークに設定されたチャネル i においてプロセス $?*(y).P$ は待機状態になる．RETURN は，リターンプロセス $i@l_1 \Leftarrow y$ が現れたときの簡約を定義している．このとき，値またはチャネル名 y が送り元のサイト l_1 へと送られ，チャネル i に y を出力するプロセス $i!y$ がサイト l_1 において生成される．これらのルールは，分散環境上の任意の場所に存在するプロセスによるサイトを越えた同期的な通信を定義するものである．チャネルを介したコミュニケーションはサイト内において Pi-calculus のルールに従う．

3.2.3 動作式例

以下の例は，基本的な分散コミュニケーションを拡張した Pi-calculus によって形式化したものである．

Example 1: Remote Input

これは，サイト l1 に遠隔チャネルにおいて分散コミュニケーションを行う入力プロセス $x@l_2?(y).P$ が現れ，サイト l2 に出力プロセス $x!a.R$ が現れた場合の例である．

$$
\begin{array}{ll}
 & [x@l_2?(y).P]_{l1} \| [x!a.R]_{l2} \\
\rightarrow & [i?(y).P]_{l1} \| [x!a.R | x?(y).i@l_1 \Leftarrow y]_{l2} \quad (1) \\
\rightarrow & [i?(y).P]_{l1} \| [R | \{a/y\}i@l_1 \Leftarrow y]_{l2} \quad (2) \\
\rightarrow & [i?(y).P | i!a]_{l1} \| [R]_{l2} \quad (3) \\
\rightarrow & [\{a/y\}P]_{l1} \| [R]_{l2} \quad (4)
\end{array}
$$

(1) では，RCOM1 によって簡約が行われる．(2) では，COMM によってプロセス $x!a.R$ とプロセス $x?(y) \Leftarrow y$ の間でコミュニケーションが行われ，プロセス $i@l_1 \Leftarrow y$ において y に a が束縛される．(3) では，RETURN によって簡約が行われる．(4) では，COMM によってプロセス $i?(y).P$ とプロセス $i!a$ の間でコミュニケーションが行われ，プロセス P において y に a が束縛される．

Example 2: Remote Output

これは，サイト l1 に遠隔チャネルにおいて分散コミュニケーションを行う出力プロセス $x@l_2!a.P$ が現れ，サイト l2 に入力プロセス $x?(y).R$ が現れた場合の例である．

$$
\begin{array}{ll}
 & [x@l_2!a.P]_{l1} \| [x?(y).R]_{l2} \\
\rightarrow & [i?[\,].P]_{l1} \| [x?(y).R | x!a.i@l_1 \Leftarrow [\,]]_{l2} \quad (1) \\
\rightarrow & [i?[\,].P]_{l1} \| [\{a/y\}R | i@l_1 \Leftarrow [\,]]_{l2} \quad (2) \\
\rightarrow & [i?[\,].P | i![\,]]_{l1} \| [\{a/y\}R]_{l2} \quad (3) \\
\rightarrow & [P]_{l1} \| [\{a/y\}R]_{l2} \quad (4)
\end{array}
$$

(1) では，RCOM2 によって簡約が行われる．(2) では，COMM によってプロセス $x?(y).R$ とプロセス $x!a.i@l_1 \Leftarrow [\,]$ の間でコミュニケーションが行われ，プロセス R において y に a が束縛される．(3) では，RETURN によって簡約が行われる．(4) では，COMM によってプロセス $i?[\,].P$ とプロセス $i![\,]$ の間でコミュニケーションが行われる．

Example 3: Remote Replication

これは，サイト l1 に遠隔チャネルにおいて分散コミュニケーションを行う複製プロセス $x@l_2?*(y).P$ が現れ，サイト l2 に出力プロセス $x!a.R$ が現れた場合の例である．

$$
\begin{array}{ll}
 & [x@l_2?*(y).P]_{l1} \| [x!a.R]_{l2} \\
\rightarrow & [i?*(y).P]_{l1} \| [x!a.R | x?*(y).i@l_1 \Leftarrow y]_{l2} \quad (1) \\
\rightarrow & [i?*(y).P]_{l1} \| [x!a.R | x?(y).i@l_1 \Leftarrow y | x?*(y).i@l_1 \Leftarrow y]_{l2} \quad (2) \\
\rightarrow & [i?*(y).P]_{l1} \| [R | \{a/y\}i@l_1 \Leftarrow y | x?*(y).i@l_1 \Leftarrow y]_{l2} \quad (3) \\
\rightarrow & [i?*(y).P | i!a]_{l1} \| [R | x?*(y).i@l_1 \Leftarrow y]_{l2} \quad (4) \\
\rightarrow & [i?*(y).P | i?(y).P | i!a]_{l1} \| [R | x?*(y).i@l_1 \Leftarrow y]_{l2} \quad (5) \\
\rightarrow & [i?*(y).P | \{a/y\}P]_{l1} \| [R | x?*y.i@l_1 \Leftarrow y]_{l2} \quad (6)
\end{array}
$$

(1) では，RCOM3 によって簡約が行われる．(2) では，プロセス $x?(y).i@l_1 \Leftarrow y$ が複製される．(3) では，COMM によってプロセス $x!a.R$ とプロセス $x?(y).i@l_1 \Leftarrow y$ の間でコミュニケーションが行われ，プロセス $i@l_1 \Leftarrow y$ において y に a が束縛される．(4) では，RETURN によって簡約が行われる．(5) では，プロセス $i?(y).P$ が複製される．(5) では，COMM によってプロセス $i?(y).P$ とプロセス $i!a$ の間でコミュニケーションが行われ，プロセス P において y に a が束縛される．

3.3 拡張 Pi-calculus に対する抽象機械

ここでは，以上で提案した計算モデルを実装するための Turner の抽象機械の拡張について述べる．

3.3.1 リダクションルール

以下のルールによって，Turner の抽象機械を拡張する．ここで，$send(n)\,to\,l$ は引数 n をサイト l へ送ることを表し，$receive(n)$ はそのサイトにおいて引数 n を受け取ったことを表す．この 2 つのプリミティブによって分散環境上の任意の場所で実行されている抽象機械の間での通信を定義する．また，以下のルールは，抽象機械がサイト l1 で実行されているものとして定義されている．

- Send-Inp:
  実行キューの先頭に $c@l_2?(y).P$ が現れたとき，サイト l_2 に $c?(y).i@l_1 \Leftarrow y$ をメッセージとして送る．それから，$?(y).P$ がユニークに設定された名前を持つチャネル i に対するキューにおいて待機状態になる．

$$\frac{send(c?(y).i@l_1 \Leftarrow y)\,to\,l_2}{H, c@l_2?(y).P :: R \rightarrow H\{i \mapsto ?(y).P\}, R}$$

- Send-Out:
  実行キューの先頭に $c@l_2!a.P$ が現れたとき，サイト l_2 に $c!a.i@l_1 \Leftarrow [\,]$ をメッセージとして送る．それから，$?[\,].P$ はユニークに設定された名前を持つチャネル i に対するキューにおいて待機状態になる．

$$\frac{send(c!a.i@l_1 \Leftarrow [\,])\,to\,l_2}{H, c@l_2!a.P :: R \rightarrow H\{i \mapsto ?[\,].P\}, R}$$

- Send-Repl:
  実行キューの先頭に $c@l_2?*(y).P$ が現れたとき，サイト l_2 に $c?*(y).i@l_1 \Leftarrow y$ をメッセージとして送る．それから，$?*(y).P$ はユニークに設定された名前を持つチャネル i に対するキューにおいて待機状態になる．

$$\frac{send(c?*(y).i@l_1 \Leftarrow y)\,to\,l_2}{H, c@l_2?*(y).P :: R \rightarrow H\{i \mapsto ?*(y).P\}, R}$$

- Send-Ret:
  実行キューの先頭に $i@l_2 \Leftarrow v$ が現れたとき，サイト l_2 に値またはチャネル名 v とチャネル名 i を送る．

$$\frac{send(v, i)\,to\,l_2}{H, i@l_2 \Leftarrow v :: R \rightarrow H, R}$$

- Rec-IOR:
  ある他のサイトからメッセージとしてリターンプロセスを伴ったプリフィックス $prefix.return$ が送られてくると，それを実行キューの最後尾に置く．

$$\frac{receive(prefix.return)}{H, R \rightarrow H, R :: prefix.return}$$

- Rec-Ret:
  ある他のサイトから値またはチャネル名 v とチャネル名 i が送られてくると，出力プロセス $i!v$ を生成し，それを実行キューの最後尾に置く．

$$\frac{receive(v,i)}{H, R \rightarrow H, R :: i!v}$$

3.3.2 動作例

次の 3 つの例は，3.3.1 節で定義したリダクションルールによって分散環境上に存在する 2 つの抽象機械が相互作用しながらプロセスを実行する様子を示すものである．ここで，サイト l_1 とサイト l_2 のそれぞれにおいて抽象機械が動作しているものとする．

Example 1: Remote Input

最初の例は，3.2.3 節で挙げた Example 1:Remote Input が拡張した抽象機械によって実現されていることを示す．

$$\begin{array}{lll|lll}
 & [H, x@l_2?(y).P]_{l1} & & & [H\{x \mapsto \bullet\}, x!a.R]_{l2} & \\
\rightarrow & [H\{i \mapsto ?(y).P\}]_{l1} & (1) & \rightarrow & [H\{x \mapsto \bullet\}, x!a.R :: x?(y).i@l_1 \Leftarrow y]_{l2} & (2) \\
\rightarrow & [H\{i \mapsto ?(y).P\}, i!a]_{l1} & (6) & \rightarrow & [H\{x \mapsto !a.R\}, x?(y).i@l_1 \Leftarrow y]_{l2} & (3) \\
\rightarrow & [H\{i \mapsto \bullet\}, \{a/y\}P]_{l1} & (7) & \rightarrow & [H\{x \mapsto \bullet\}, \{a/y\}i@l_1 \Leftarrow y :: R]_{l2} & (4) \\
 & & & \rightarrow & [H\{x \mapsto \bullet\}, R]_{l2} & (5)
\end{array}$$

(1)Send-Inp (2)Rec-IOR (3)Out-W (4)Inp-W (5)Send-Ret (6)Rec-Ret (7)Out-R

Example 2: Remote Output

次の例は，3.2.3 節で挙げた Example 2:Remote Output が拡張した抽象機械によって実現されていることを示す．

$$\begin{array}{lll|lll}
 & [H, x@l_2!a.P]_{l1} & & & [H\{x \mapsto \bullet\}, x?(y).R]_{l2} & \\
\rightarrow & [H\{i \mapsto ?[\,].P\}]_{l1} & (1) & \rightarrow & [H\{x \mapsto \bullet\}, x?(y).R :: x!a.i@l_1 \Leftarrow [\,]]_{l2} & (2) \\
\rightarrow & [H\{i \mapsto ?[\,].P\}, i![\,]]_{l1} & (6) & \rightarrow & [H\{x \mapsto ?(y).R\}, x!a.i@l_1 \Leftarrow [\,]]_{l2} & (3) \\
\rightarrow & [H\{i \mapsto \bullet\}, P]_{l1} & (7) & \rightarrow & [H\{x \mapsto \bullet\}, i@l_1 \Leftarrow [\,] :: \{a/y\}R]_{l2} & (4) \\
 & & & \rightarrow & [H\{x \mapsto \bullet\}, \{a/y\}R]_{l2} & (5)
\end{array}$$

(1)Send-Out (2)Rec-IOR (3) Inp-R (4)Out-R (5)Send-Ret (6)Rec-Ret (7)Out-R

Example 3: Remote Replication

次の例は，3.2.3 節で挙げた Example 3:Remote Replication が拡張した抽象機械によって実現されていることを示す．

$$\begin{array}{lll|lll}
 & [H, x@l_2?*(y).P]_{l1} & & & [H\{x \mapsto \bullet\}, x!a.R]_{l2} & \\
\rightarrow & [H\{i \mapsto ?*(y).P\}]_{l1} & (1) & \rightarrow & [H\{x \mapsto \bullet\}, x!a.R :: x?*(y).i@l_1 \Leftarrow y]_{l2} & (2) \\
\rightarrow & [H\{i \mapsto ?*(y).P\}, i!a]_{l1} & (7) & \rightarrow & [H\{x \mapsto !a.R\}, x?*(y).i@l_1 \Leftarrow y]_{l2} & (3) \\
\rightarrow & [H\{i \mapsto ?*(y).P\}, \{a/y\}P]_{l1} & (8) & \rightarrow & [H\{x \mapsto \bullet\}, x?*(y).i@l_1 \Leftarrow y :: \{a/y\}i@l_1 \Leftarrow y :: R]_{l2} & (4) \\
 & & & \rightarrow & [H\{x \mapsto ?*(y).i@l_1 \Leftarrow y\}, \{a/y\}i@l_1 \Leftarrow y :: R]_{l2} & (5) \\
 & & & \rightarrow & [H\{x \mapsto ?*(y).i@l_1 \Leftarrow y\}, R]_{l2} & (6)
\end{array}$$

(1)Send-Repl (2)Rec-IOR (3)Out-W (4)Repl-W (5)Repl-R (6)Send-Ret (7)Rec-Ret (8)Out-R*

第4章　実装

本章では，3 章で提案した計算モデルに基づく言語システムをどのように実装したかについて述べる．

4.1　システムの構成

拡張 Pi-calculus に基づく言語システムは，Lexer/Parser,Interpreter,Messenger から構成される．それぞれの構成要素は，マルチスレッドによって非同期に動作していている．このシステムの一連の動作の流れは次のようになる．まず，ユーザから入力されたプログラムは Lexer/Parser によって Interpreter が解釈可能な構造を持つ抽象構文木 (abstract syntax tree:AST) に変換され，Interpreter に渡される．Interpreter は，AST を実行キューに受け取り，それを解釈しながら 2.4.2 節と 3.3.1 節で定義されたリダクションルールに従って動作することになる．この言語システムは分散環境に実装されるものであり，他のサイトで動作している別の言語システムとメッセージの送受信を行う．メッセージの送信は，Interpreter によって行われる．これはリダクションルール Send-Inp,Send-Out,Send-Repl,Send-Ret に従うものである．メッセージの受信は，Messenger というサーバによって行われる．このサーバは他のサイトから送られてきたメッセージを受信すると，そのメッセージを解釈して，それに対応する AST を生成して，それを Interpreter に渡す．Interpreter はそれを実行キューで受け取る．これはリダクションルール Rec-IOR,Rec-Ret に従うものである．以上のように，この言語システムは，それぞれの構成要素が相互作用しながら動作する．この言語システムの全体像を図 4.1 に示す．

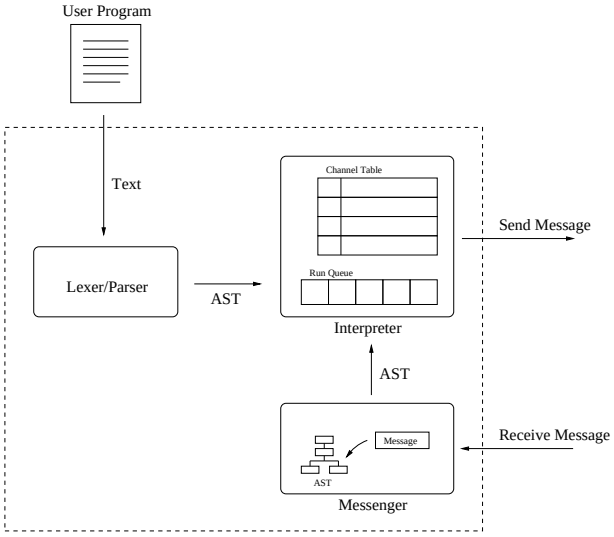

図 4.1: 拡張 Pi-calculus に基づく言語システム

4.2 ソースコードの説明

本研究では，4.1 節で説明した言語システムを Java によって実装した．以下に，それぞれの構成要素をどのように実装したかソースコードの主要な部分を載せながら説明していく．

4.2.1 Lexer/Parser

Lexer/Parser の実装には，JavaCC と JJTree というツールを用いた．JavaCC(Java Compiler Compiler) は，字句解析・構文解析を行うクラスのソースコードを自動生成してくれるツールである．Unix で言うところの「yacc/lex」に相当する．JJTree は JavaCC のためのプリプロセッサであり，JavaCC ソースの様々な場所にパースツリーを構築するためのアクションを挿入する．JJTree の出力ファイルは字句解析・構文解析器を生成するために JavaCC に通すことができる．以下に，これらのツールを用いて実装した手順に従って説明していく．

文法定義：

拡張 Pi-calculus の文法は，以下のように定義する．

$$
\begin{array}{rcl}
Processes & ::= & Parallel\langle EOF\rangle \\
Parallel & ::= & Process(\text{“|”}\ Process)* \\
Process & ::= & Nil|ChannelCreation|IORProcess(\text{“.”}\ IORProcess)* \\
Nil & ::= & \text{“}nil\text{”} \\
ChannelCreation & ::= & \text{“(”“}new\text{”}\ Name\ \text{“)”“(”}\ Parallel\ \text{“)”} \\
IORProcess & ::= & Name \\
 & & (\text{“!”}\ (Element|Null)|\ \text{“?”}\ ((\text{“(”}\ Element\ \text{“)”}\ |Null)|\ \text{“}*\text{”}\ (\text{“(”}\ Element\ \text{“)”}\ |Null))| \\
 & & \text{“@”}\ Name \\
 & & (\text{“!”}\ (Element|Null)|\ \text{“?”}\ ((\text{“(”}\ Element\ \text{“)”}\ |Null)|\ \text{“}*\text{”}\ (\text{“(”}\ Element\ \text{“)”}\ |Null))| \\
 & & \text{“}<-\text{”}\ (Element|Null))) \\
Null & ::= & \text{“[”}\ \text{“]”} \\
Element & ::= & Name[\text{“@”}\ Name]|Digits \\
Name & ::= & \text{英字で始まり英数字からなる文字列} \\
Digits & ::= & \text{数字列}
\end{array}
$$

抽象構文木の構造：

上の文法定義に従って生成される抽象構文木の構造は，図 4.2 のようになる．図 4.2 で，IOProcess と Element のノードは破線で表記してあるが，これは構文解析用の便宜のためであり，実際はこの 2 つのノードは生成されないことを表す．また，Output,Input,Replication,ROutput,RInput,RReplication,Return は同じ構造を持つのでまとめて表記した．

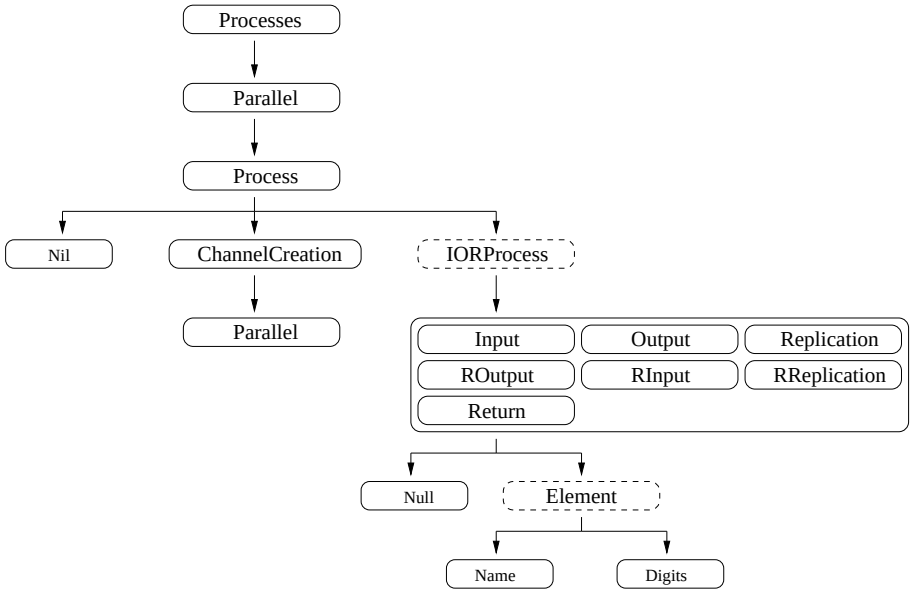

図 4.2: 生成される抽象構文木の構造

Lexer/Parser の生成：

次に，上で定義した文法と抽象構文木の構造に従って JJTree/JavaCC 用の文法定義ファイル (.jjt ファイル) を作成する．このファイルは一部 Java コード，一部 JJTree/JavaCC 専用コードを使って記述する．そのファイルは次のように作成した．以下のコードは重要な部分だけを抜粋したものである．

```
// 構文の定義 Java のメソッドのような形式で構文を定義する.
ASTProcesses Processes() :{}{Parallel() <EOF>}
void Parallel() :{}{Process() (<PARALLEL> Process())*}
void Process() :{}{Nil()|ChannelCreation()|(IORProcess()
                   (<PERIOD> IORProcess())*)}
void Nil() :{}{"nil"}
void ChannelCreation() #void:{}{<LPAREN> "new" <NAME> <RPAREN> <LPAREN>
      Parallel() <RPAREN> #ChannelCreation(1)}
void IORProcess() #void:{}{<NAME> (<OUTPUT> (Element() | Null()) #Output(1) |
      <INPUT> ((<LPAREN> Element() <RPAREN> | Null()) #Input(1) |<ASTERISK>
      (<LPAREN> Element() <RPAREN> | Null())#Replication(1)) | <AT> <NAME>
      (<OUTPUT> (Element()| Null()) #ROutput(1) | <INPUT> ((<LPAREN> Element()
       <RPAREN> | Null()) #RInput(1)|<ASTERISK> (<LPAREN> Element() <RPAREN> |
       Null()) #RReplication(1)) | <RETURN> (Element() | Null()) #Return(1)))}
void Null() :{}{<LFLOOR> <RFLOOR>}
void Element() #void:{}{<NAME> [<AT> <NAME>]#Name | <DIGITS> #Digits}

// 字句の定義
TOKEN :{<OUTPUT: "!"> | <INPUT: "?"> | <LPAREN: "("> | <RPAREN: ")"> |
        <LFLOOR: "["> | <RFLOOR: "]"> | <PARALLEL: "|"> | <AT: "@"> |
        <COMMA: ","> | <PERIOD: "."> | <ASTERISK: "*"> | <RETURN: "<-">}
TOKEN :{< #DIGIT: ["0"-"9"] > | < #LETTER: ["a"-"z","A"-"Z"] > |
        < STRING: "\"" (~["\"","\n","\r"])* "\"" > |
        < NAME: <LETTER> (<LETTER> | <DIGIT> | "_")* > |
        < DIGITS: (<DIGIT>)+ >}
```

上の定義ファイルを JJTree/JavaCC でコンパイルすることによって，Lexer/Parser と構文木のそれぞれのノードに対応したクラス (e.g.ASTProcesses,ASTParallel,ASTProcess,...) のソースコードが生成される．

構文木のノードに対応するクラスは，Node インターフェースを実装した SimpleNode クラスの拡張として与えられる．これによって，それぞれのノードに対応するクラスには親ノードの追加，子ノードの追加・参照などのメソッドが提供される．これらのメソッドを用いることで，生成された構文木を解釈・操作することができるようになる．以下に，それらのメソッドについて説明する．

- public void jjtSetParent(Node n)
  そのノードに親ノードとしてノード n を加える．
- public Node jjtGetParent()
  そのノードの親ノードを返す.
- int jjtGetNumChildren()
  そのノードが持つ子ノードの数を返す.
- Node jjtGetChild(int i)
  そのノードが持つ i 番目の子ノードを返す.
- void jjtAddChild(Node n,int i)
  そのノードに i 番目の子ノードとしてノード n を加える．

4.2.2 Interpreter

Interpreter は，実行キュー，環境，チャネルテーブル，解釈・実行部から構成される．この構成は Turner の抽象機械に対応するものであり，その仕様は 2.3 節と 3.3 節で定義される．以下にぞれぞれの構成要素の実装について説明する．

実行キュー：

実行キューは，プロセスと環境の組を格納するものである．そのコードは次のようになる．

```
class RunQueue
{
    static Queue runqueue = new Queue();
}
```

Runqueue クラスは Queue クラスのオブジェクト runqueueu を static に宣言している．Queue クラスは，キューを実装したものであり，以下のメソッドによって実行キューへの同期的なアクセスを行うことができる．

- Object get_front()
  実行キューの先頭のオブジェクトを取り出す.
- Object refer_front()
  実行キューの先頭のオブジェクトを参照する．オブジェクトの取り出しは行わない.
- void append_node(Object new_data)
  実行キューの最後尾に引数として渡されたオブジェクトを格納する．ここでは，Process ノードのオブジェクトと環境のオブジェクトを Vector オブジェクトに追加することによって一つのオブジェクトとして append_node メソッドへ渡す.

環境：

環境は，名前（変数，チャネル名）に対する束縛を格納するものである．そのコードは次のようになる．

```
class Environment {
    Hashtable env = new Hashtable();
}
```

Environment クラスは，Hashtable クラスのオブジェクト env を宣言している．ハッシュテーブルとは，関連するキーを使って格納されたデータを検索可能なデータ構造のことを言い，Hashtable クラスによって実装されている．ここでは，名前をキーとすることで，それに対する束縛を格納する．

チャネルテーブル：

チャネルテーブルは，あるチャネル名に対応するチャネルキューを格納するものである．チャネルキューには，そのチャネルにおいて待機状態にあるプロセスと環境の組が格納される．そのコードは次のようになる．各メソッドの中身は省略し，後でまとめて説明する．

```
class ChannelTable {
    static Hashtable channeltable = new Hashtable();
    static void put_process(String name,Vector proc_env){}
    static Vector get_process(String name){}
    static Vector refer_process(String name){}
    static void channel_creation(String name){}
    static void queue_remove(String name){}
    static boolean null_queue(String name){}
}
```

ChannelTable クラスは，Hashtable クラスのオブジェクト channeltable を static に宣言している．channeltable にはチャネル名 (name) をキーとして，それに対するチャネルキュー (Queue オブジェクト) が格納される．チャネルテーブルには，以下のメソッドによってアクセスすることができる．

- void put_process(String name,Vector proc_env)
  チャネル名に対するチャネルキューにプロセスと環境の組 (Vector オブジェクト) を格納する．
- Vector get_process(String name)
  チャネル名に対するチャネルキューから先頭のプロセスと環境の組を取り出す．
- Vector refer_process(String name)
  チャネル名に対するチャネルキューの先頭のプロセスと環境の組を参照する．オブジェクトの取り出しは行わない．
- void channel_creation(String name)
  チャネル名に対するチャネルキューを新規生成し，チャネルテーブルに登録する．
- void queue_remove(String name)
  チャネル名に対するチャネルキューを削除する．
- boolean null_queue(String name)
  チャネル名に対するチャネルキューが空 (null) かどうか調べる．

解釈・実行部：

ここでは，2.3 節と 3.3 節で定義された仕様に基づいてプロセスの実行を行う．そのコードは次のようになる．各々のリダクションルールに対する処理の部分は省略し，それについては後で説明する．

```
class Interpreter extends Thread {
    RunQueue queue;
    String mylocation;
    static NetworkClient net;
    static DataInputStream net_input;
    static PrintStream net_output;

    public Interpreter(String location)
    {
        mylocation = location;
        queue = new RunQueue();
    }

    public void run()
    {
        while(true){
           if(queue.runqueue.first != null){
                // 実行キューの先頭にあるプロセスと環境を参照
                Vector rproc_env = (Vector)queue.runqueue.refer_front();
                ASTProcess rproc = (ASTProcess)rproc_env.elementAt(0);
                Environment renv = (Environment)rproc_env.elementAt(1);
                // リダクション Nil に対する処理
                if(rproc.jjtGetChild(rproc.order) instanceof ASTNil){}
                // リダクション Res に対する処理
                else if(rproc.jjtGetChild(rproc.order)
                          instanceof ASTChannelCreation){}

                else if(rproc.jjtGetChild(rproc.order) instanceof ASTOutput){
                    ChannelTable ct = new ChannelTable();
                    ASTOutput outproc =
                      (ASTOutput) rproc.jjtGetChild(rproc.order);
                    String cname = (String) renv.env.get(outproc.cname);
                    if(ct.null_queue(cname)){
                        ct.put_process(cname,rproc_env);
                        queue.runqueue.get_front();
                      }
                    else{
                    // 対応するチャネルキューの先頭にあるプロセスと環境を参照
                    Vector cproc_env = (Vector) ct.refer_process(cname);
                    ASTProcess cproc = (ASTProcess)cproc_env.elementAt(0);
                    Environment cenv = (Environment)cproc_env.elementAt(1);
                   // リダクション Out-R に対する処理
                   if(cproc.jjtGetChild(cproc.order) instanceof ASTInput){}
                   // リダクション Out-W に対する処理
                   else if(cproc.jjtGetChild(cproc.order)
                             instanceof ASTOutput){}
                   // リダクション Out-R*に対する処理
                    else if(cproc.jjtGetChild(cproc.order)
                             instanceof ASTReplication){}
                   }
                }

            else if(rproc.jjtGetChild(rproc.order) instanceof ASTInput){
            // リダクション Inp-W,Inp-R に対する処理
                 }
            else if(rproc.jjtGetChild(rproc.order) instanceof ASTReplication){
                   // リダクション Repl-W,Repl-R に対する処理
                 }
           // リダクション Prl に対する処理
           else if(rproc.jjtGetChild(rproc.order) instanceof ASTParallel){}
```

```
            // リダクション Send-Inp に対する処理
            else if(rproc.jjtGetChild(rproc.order) instanceof ASTRInput){}
            // リダクション Send-Out に対する処理
            else if(rproc.jjtGetChild(rproc.order) instanceof ASTROutput){}
            // リダクション Send-Repl に対する処理
            else if(rproc.jjtGetChild(rproc.order) instanceof ASTRReplication){}
            // リダクション Send-Ret に対する処理
            else if(rproc.jjtGetChild(rproc.order) instanceof ASTReturn){}
          }
        }
    }
    // ホスト名が location であるサイトにメッセージ (message) を送るメソッド
    public void send_message(String message,String location){}
    // サーバとの接続を終了するメソッド
    static void close_server(){}
}
```

Interpreter クラスは，Thread のサブクラスとして定義されている．つまり，解釈・実行部は一つのスレッドとして動作することになる．このサブクラスのスレッドが生成されると，そのスレッドは run() メソッドの中のコードを実行する．ここに，2.3 節と 3.3 節で定義された抽象機械の処理を記述する．run() メソッドの中では，while ループによって抽象機械のリダクションを永続的に行うことになる．各々のリダクションの処理は，実行キュー (queue) とチャネルテーブル (ct) から先頭にあるプロセスと環境を参照し，それらによって場合分けすることで行われる．また，メッセージの送信には send_message メソッドを用いる．次に，リダクションに対する処理について Out-R と Send-Out のコードを挙げて説明する．

- Out-R：

```
if(cproc.jjtGetChild(cproc.order) instanceof ASTInput){
   ASTInput inproc = (ASTInput) cproc.jjtGetChild(cproc.order);
   // 引数の束縛
   if(outproc.jjtGetChild(0) instanceof ASTName){
      Object element = renv.env.get(outproc.jjtGetChild(0).element());
      cenv.env.put(inproc.jjtGetChild(0).element(),element);}
   else if(outproc.jjtGetChild(0) instanceof ASTDigits){
      cenv.env.put(inproc.jjtGetChild(0).element(),
                   outproc.jjtGetChild(0).element());}
   // 実行キューのプロセスを一つ進める 次がないならば，実行キューから削除
   rproc.order++;
   if(rproc.order >= rproc.jjtGetNumChildren())
      queue.runqueue.get_front();

   // チャネルキューのプロセスを一つ進め，実行キューへ格納する
   Vector pe = new Vector();
   cproc.order++;
   if(cproc.order < cproc.jjtGetNumChildren()){
      pe.addElement(cproc);
      pe.addElement(cenv);
      queue.runqueue.append_node(pe);}
   // チャネルキューの先頭にあるプロセスを削除
   ct.get_process(cname);
}
```

上のコードは，実行キューに出力プロセス，チャネルキューに入力プロセスが現れた場合の処理を記述している．まず最初に，コミュニケーションの結果としてチャネルキューの入力プロセスが持つ環境 (cenv) に引数の束縛を格納する．次に実行キュー (queue) にあるプロセス (rproc) を一つ進める．これは，ASTProcess クラスが持つ変数 order をインクリメントして行う．これによって，Process ノードが持つ何番目の子ノードを参照すべきか分かるようになる．同様に，チャネルキューのプロセ

ス (cproc) も一つ進め，それを実行キューへ格納する．最後にチャネルキューの先頭にあるプロセス (cproc) を削除する．

- Send-Out：

```
else if(rproc.jjtGetChild(rproc.order) instanceof ASTROutput){
   // 新規チャネルを生成し，それをチャネルテーブルに登録する
   String channel;
   NameCreation nc = new NameCreation();
   ChannelTable ct = new ChannelTable();
   ASTROutput routp = (ASTROutput)rproc.jjtGetChild(rproc.order);
   channel = nc.name_creation();
   ct.channel_creation(channel);
   // 新規チャネルにおいて遠隔出力プロセスは待機状態になる
   ASTInput i_proc = new ASTInput(0);
   i_proc.cname = channel;
   i_proc.jjtAddChild(new ASTNull(0),0);
   rproc.jjtAddChild(i_proc,rproc.order);
   Vector pe = new Vector();
   pe.addElement(rproc);
   pe.addElement(renv);
   ct.put_process(channel,pe);
   // 実行キューの先頭のプロセスを削除
   queue.runqueue.get_front();
   // メッセージの送信
   String rch = routp.cname + "@" + routp.location;
   StringTokenizer strt =
                new StringTokenizer((String)renv.env.get(rch),"@");
   rch = strt.nextToken();
   String message = "remote-comm " + rch + " out ";

   if(!(routp.jjtGetChild(0) instanceof ASTNull)){
      Object element = routp.jjtGetChild(0).element();
      if(element instanceof String){
         Object e = renv.env.get(element);
         if(e instanceof String)
            message += " arg name " + (String)e;
         else
            message += " arg digit " + ((Integer)e).toString();
        }
      else
         message += " arg digit " + ((Integer)element).toString();
     }
   else
         message += " null ";

   message = message + " " + channel + " " + mylocation;
   send_message(message,routp.location);
}
```

上のコードは，実行キューに遠隔出力プロセスが現れた場合の処理を記述している．まず最初に，NameCreation クラスの name_creation() メソッドを用いて新規チャネル名を生成する．このメソッドはカウンタであり，整数値をインクリメントして重複のない名前としてそれを返す．これによって生成された名前 (channel) を引数として ChannelTable クラスの channel_creation(String name) メソッドを呼び出すことでチャネルテーブルへの登録が行われる．次に，jjtAddChild(Node n,int i) メソッドを用いて null で待機状態になる入力プロセスノード (i_proc) を生成し，それを実行キューの先頭のプロセス (rproc) において今参照している遠隔出力プロセスノードと置き換える．それから実行キューの先頭のプロセスはチャネルテーブル (ct) に格納され，実行キューから削除される．最後にメッセー

ジを生成し，それを引数として send_message(String message,String location) メソッドを呼び出すことで，メッセージの送信が行われる．生成されるメッセージの内容については次節で説明する．

4.2.3 Messenger

Messenger は，Interpreter によってリダクション Send-Inp,Send-Out,Send-Repl,Send-Ret の結果送られてきたメッセージを受信する．受信するメッセージは，識別子と値からなる一つの文字列であり，次のものがある．

- remote-comm “ c ” (out/inp/rep) (arg/null) (digit/name) “ x ” “ i ”“ l ”
  このメッセージは，プロセス $c(!/?/?*)(x/[]).i@l \Leftarrow (x/[])$ に対する情報を持つ．remote-comm は，識別子であり，このメッセージがリダクション Send-Inp,Send-Out,Send-Repl の結果送られてきたものであることを表す．“ c ” は，受け手のサイトにあるチャネルの名前である．(out/inp/rep) は，プロセスの種類を表す識別子である．(arg/null) は，プロセスの引数があるのかないのかを表す識別子である．arg のときは，その後に (digit/name) という識別子が続く．これは，次にくるものが数値なのか名前なのかを表す．“ x ” は，数値または名前である．“ i ” は，送り手のサイトのチャネル名である．“ l ” は，送り手のサイトのホスト名である．

- return (arg/null) (digit/name) “ x ”“ i ”
  このメッセージは，プロセス $i!x$ に対する情報を持つ．return は，識別子であり，このメッセージがリダクション Send-Ret の結果送られてきたものであることを表す．(arg/null) は，プロセスの引数があるのかないのかを表す識別子である．arg のときは，その後に (digit/name) という識別子が続く．これは，次にくるものが数値なのか名前なのかを表す．“ x ” は数値または名前である．“ i ” は受け手のサイトにあるチャネルの名前である．

Messenger のコードは，次のようになる．リダクション Rec-IOR,Rec-Ret に対する処理は，メッセージに対応する構文木の生成が主であり，これは jjtAddChild(Node n,int i) メソッドを用いて容易に実現できるので，ここでは省略する．

```
import java.io.*;
import sun.net.*;

class Messenger extends NetworkServer
{
    DataInputStream net_input;
    static NetworkClient net;
    static PrintStream net_output;
    static String mylocation;

    public Messenger(String hostname)
    {
        try {
             mylocation = hostname;
             startServer(1111);
            }
        catch (Exception e)
            {
                System.out.println("Unable to start server");
                return;
            }
    }
    // メッセージを受信したとき，呼び出されるメソッド
```

```
    public void serviceRequest()
    {
        net_input = new DataInputStream(clientInput);
        String message = read_net_input();
        StringTokenizer st = new StringTokenizer(message," ");
        String nt = st.nextToken();

        if(nt.equals("remote-comm")){// リダクション Rec-IOR に対する処理}
        else if(nt.equals("return")){// リダクション Rec-Ret に対する処理}
        System.out.println(message);
    }

    String read_net_input()
    {
        try {return net_input.readLine();}
        catch (IOException e){return null;}
    }
}
```

このプログラムでは，java.io パッケージと sun.net パッケージの 2 つをインポートしている．sun.net パッケージは NetworkServer クラスを含むパッケージであり，Messenger クラスは NetworkServer クラスを拡張して作成する．このクラスのメッセージ受信に対する処理は，serviceRequest() メソッド内に記述する．ここでは，受信したメッセージを read_net_input() メソッドによって読み込む．それから，読み込んだメッセージを StringTokenizer オブジェクトを使って読み進めていき，識別子によって場合分けしながら，リダクション Rec-IOR,Rec-Ret に対する処理を行う．

4.3 実行結果

ホスト名がそれぞれ is15e0dhc12 と is15e0u16 である 2 台のコンピュータで実行した結果を以下に示す．ここでの実行結果は，まず字句解析/構文解析が行わて生成された抽象構文木が階層的に表示され，次に受信したメッセージと print プリミティブによる出力が表示されている．

- **is15e0dhc12**

  [入力プログラム]

```
(new c1)((new c2)(X@is15e0u16!c1@is15e0dhc12 |
     c1?(n).X@is15e0u16!2.c2!n | c2?(m).print(m)))
```

  [実行結果]

```
[h-fukuda@is15e0dhc12 Pi]$ java PiAM sample4 is15e0dhc12
Interpreter for Pi-calculus Version 0.1:Reading from file sample4 . . .

Processes
 Parallel
  Process
   ChannelCreation
    Parallel
     Process
      ChannelCreation
       Parallel
        Process
         ROutput
          Name
        Process
         Input
```

```
          Name
         ROutput
          Digits
         Output
          Name
        Process
         Input
          Name
         Print
EOF

return null 2
remote-comm 0 out arg digit 1 1 is15e0u16
return null 3
1
```

- **is15e0u16**

  [入力プログラム]

```
X?(y@is15e0dhc12).y@is15e0dhc12!1.X?(n).print(n)
```

  [実行結果]

```
[h-fukuda@is15e0u16] 23 % java PiAM sample4 is15e0u16
Interpreter for Pi-calculus Version 0.1:Reading from file sample4 . . .

Processes
 Parallel
  Process
   Input
    Name
   ROutput
    Digits
   Input
    Name
   Print
EOF
remote-comm 0 out arg name 0 2 is15e0dhc12
return  null 1
remote-comm 0 out arg digit 2 3 is15e0dhc12
2
```

第5章　結論及び今後の課題

本研究では，Pi-calculus の意味論について考察し，それを分散環境へ実装する上でどのような問題が起こるかを明らかにし，その解決策を与えた．そして，分散環境へ実装可能な Pi-calculus の拡張モデルとそれに対する抽象機械を提案した．さらに，実際にその拡張モデルに基づく言語システムを分散環境へ実装し，動作の確認を行った．以下に，結論及び今後の課題について述べる．

5.1　結論

本研究によって得られた結論は以下の通りである．

- 同期通信のメカニズムを導入することによって，Pi-calculus の特性をそのまま持つ分散環境へ実装可能な計算モデル（以下，拡張 Pi-calculus と呼ぶ）を提案することができた．
- 拡張 Pi-calculus によって分散コミュニケーションを明確に形式化することができるようになった．
- Turner の抽象機械を拡張することで，拡張 Pi-calculus に対する実装を与えることができた．
- 拡張 Pi-calculus に基づくことによって，分散コミュニケーションを容易に，かつ柔軟に記述することができる言語システムを実装することができた．
- 実際に分散環境へ拡張 Pi-calculus に基づく言語システムを実装したことによって，今後の研究における実験のベースを与えることができた．

5.2　今後の課題

今後の課題として以下の事が挙げられる．

- 現在，Pi-calculus に対する型システムについてさかんに研究が行われている．型システムを導入することによって，型チェックや型推論を行うことができるようになり，プログラムの正当性や安全性を保証することができるようになる．そのため，実用レベルの言語システムを構築するために型システムを導入することは必須である．本研究で提案した計算モデルは，Pi-calculus の特性をそのまま持つものであり，これまでに提案されてきた型システムを統合・拡張して導入することが可能なはずである．
- 分散環境では，物理的なサイトの故障や停止によって，メッセージやプロセスが失われ，様々な予期されないエラーが発生する．そのため，この障害に対する機能を抽象機械に明確に定義しておく必要がある．
- 現在のモデルでは，チャネル名のスコープによってチャネルへのアクセス権限を行っている．しかし，チャネル名の漏洩によってプライバシーは侵害されてしまう．そこで，明確なチャネルへのアクセス

権限を行うことによって，例えチャネル名が漏洩してしまったとしてもプライバシーを保護することが求められる．

第6章　謝辞

本研究を進めるに際し，毎週の個人ゼミという形式で終始御指導くださった大堀淳先生には深く感謝致します．ならびに田島敬史先生，Rene Vestergaard 助手，そして計算機言語学講座の皆様と有意義に学生生活を送れたことに感謝したい．最後に，両親に感謝致します．

参考文献

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

[2] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department,Indiana University,1997.

[3] David N. Turner. The Polymorphic Pi-calculus: Theory and Implementation. PhD thesis,University of Edinburgh,1996.

[4] Pawel T. Wojciechowski and Peter Sewell. *Nomadic Pict: Language and infrastructure design for mobile agents.* In Proceedings of ASA/MA '99 (First International Symposium on Agent Systems and Applications /Third International Symposium on Mobile Agents), Palm Springs, CA, USA, October 1999.

[5] Cedric Fournet,Georges Gonthier, Jean-Jacques Levy,Luc Maranget, and Didier Remy. A calculus of mobile agents. In Proceedings of *CONCUR '96. LNCS 1119*, pages 406-421. Springer-Verlag,August,1996.

[6] L.Cardelli,A.D.Gordon. Mobile Ambients. in Foundations of Software Science and Computational Structures, Maurice Nivat(Ed.),Lecture Notes in Computer Science, Vol. 1378,Springer,140-155,1998.