ソフトウェア工学 89－7
（1992. 12. 14）

対話型ソフトウェアの動的な部品間接続の方式について

佐藤豊　大蒔和仁

電子技術総合研究所

対話型ソフトウェアは、比較的粗い粒の疎に結合された並行動作する部品の統合体ととらえることができる。本論文ではこのような部品＝エージェント間を柔軟に結合する方式を提案し、その試作について述べる。ここでは互いに異なる言語で記述されたエージェント間での共通の通信プロトコル記述に並列計算モデルの一つであるπカリキュラスを用いる。プロトコルの記述はそのまま実行されて実際の通信が実行される。これにより、πカリキュラスの高い記述力による柔軟な通信制御が実現でき、また外部からエージェント内部の構造モデルを知り、実行状態を監視し制御することが可能となった。

A Flexible Inter-Agent Connection Scheme for Interactive Software

Yutaka Sato　Kazuhito Ohmaki [1]

Electrotechnical Laboratory
1-1-4 Umezono, Tsukuba, Ibaraki 305, Japan

This paper presents a scheme for inter-agent connection, where the agents are software modules of coarse granularity, communicate without central memory, and are written in diverse languages. A process algebra named π- calculus is chosen as the base of the common description of inter-agent communication protocol among agents written in different languages. Expressions written in π-calculus are executed directly to carry out communication, as well as static descriptions. Thus high expressive power of π-calculus makes control of communication protocol flexible, and internal framework and status of each agent manageable from outside of the agent.

[1]E-mail: ysato@etl.go.jp, ohmaki@etl.go.jp

1 はじめに

コンピュータネットワークの発達を基礎としてクライアント／サーバモデルによる粗い粒のソフトウェアの部品化は成功を収めてきた。例えば、ウィンドウサーバ、かな漢字変換サーバ、ニュースサーバやメイルサーバなどである。これらの部品は複数の言語環境へ共通の機能を提供することに成功している。中でもインターネット上の各種のプロトコルであるFTP,SMTP,NNTP,POPなど、文字列ベースのプロトコルは複数の言語で記述されたクライアントからの共有利用を容易にしている。

対話型ソフトウェアの実現技術は、このように提供された部品群、ユーザインターフェイス部品と応用処理部品を、対話制御アルゴリズムのもとで連係させながら繋ぎ合わせる技術ととらえることができる。対話型ソフトウェアの開発支援システムとして研究が進められているユーザインターフェイス管理システム (UIMS) においても、インターフェイス部品と応用部品の間を分離し結合する方式と、そのための部品のモデル化は重要な課題となっている。

ここで、インターフェイス部品と応用部品を密接に統合するにはクライアント／サーバモデルでは不十分である。必要なものは、互いに対等な立場で協調動作する複数のエージェント間の通信モデルである。部品化再利用の観点からは、特定の言語の制御構造やデータ構造から独立した通信モデルが必要である。

筆者らは、UIMSの実行時アーキテクチャの基礎として、VIABUSを開発した[1]。VIABUSは文字列ベースのプロトコルであり、ネットワークワイドな通信媒体である。VIABUSによる文字列ベースのプロトコルは多様な言語で書かれた部品の接続を容易にしているが、このプロトコルを生で用いると、複雑な通信の制御が必要な場合にその記述を各エージェントで行なう必要があり、またエージェント内部の構造や状態を外部から (UIMSから) 把握し制御することができない。

任意の言語によるエージェントの記述を許しつつ、各エージェントの枠組のモデルと現在の状態を把握する必要がある。この目的のためには、各エージェントのモデルを、プロセス代数などの形式的な並列計算モデルを利用して記述することが、一つの解であると考えられる。Milnerらによるπカリキュラス[2]は、我々の目的に充分な記述力を持つとともに、その簡潔な実行モデルは実際のシステム実行時の通信メカニズムとしても利用可能と考えられたので、その実現を試みることにした。

本論文では以下、2でUIMSの立場からの部品接続方式への要求を述べる。3ではこれらの要求を満たすために、πカリキュラスをベースにした通信モデルを検討し、4で処理系の試作とVIABUSへの接続について述べる。

2 UIMSのためのソフトウェア部品接続方式への要求

UIMSはユーザインターフェイスの開発と運用を支援するシステムであり、これは、特定の目的を設定したソフトウェア開発支援環境と言える。我々は、特定のインターフェイス装置や対話モデル、応用の種類に依存しない、多目的の柔軟なUIMSの実現を目標としている。そのためには、部品化・再利用、仕様記述、プログラム合成などのソフトウェア工学全般の技術を総合的に扱うことが必要となる。

ただし、個々の部品の詳細な記述を支援することはUIMSの守備範囲外であり、比較的粒の大きな部品の組を与えられて、それらの間をいかに柔軟に結ぶかが、UIMSの主要な課題である。ここで、各部品は内部状態を持って並行動作しながら協調するエージェントである。したがって、

接続方式 = 通信モデル
＋ 部品インターフェイスの記述形式
＋ 部品の内部構造モデルの記述形式

をいかに実現するかがUIMSの重要な基礎であり、これらを統合的に記述できる方式が望まれる。以下に、要求をより具体的にして列挙する。

複合通信モデル: Hartsonらは、制御の中心をインターフェイス側、応用側、対話制御部のいずれに置くかにより、内部制御、外部制御、均衡制御に分類した[3]。対話ソフトウェアの性質により、このいずれが向いているかは異なり、いずれも支援することが望ましい。クライアントサーバモデルのみによる通信では不足であり、複数の通信モデルのサポートが必要である。

複合接続方式: Coutazは応用部品とインターフェイス部品との間の接続を、必要に応じて静的接続、動的接続、1対N接続、N対Nの接続で実現できることの必要性を述べている[5]。複数のエージェントとの通信を個々のエージェントで管理するのではなく、共通の通信メカニズムが提供されることが望ましい。

複合実行方式: ManheimerらはLUISの実現を通じて、部品間の接続のインターフェイスを変えずに、性能上の必要に応じて密結合／疎結合で実現できることの有用性を示した[4]。

言語独立性: 部品の種類によって、その記述に最適な言語は異なる。特定の言語の制御構造やデータ構造に依存しない共通な通信方式であることが必要である。また、既存の部品を大幅に書き換えることなく接続できることも重要である。

P	::=	0	終了状態のエージェント (*inaction*)
		$\bar{y}.x.P$	y という名前のポートから x を送り出して P となる (*negative prefix*)
		$y(x).P$	y という名前のポートから受け取りそれを x に束縛して $P\{x'/x\}$ となる (*positive prefix*)
		$\tau.P$	なにもせずに P となる (*silent prefix*)
		$[x{=}y]P$	x と y が等しいなら P に、そうでないなら 0 になる (*match*)
		$(x)P$	x という名前の通用範囲を P の中に限定する (*restriction*)
		$P_1 \mid P_2$	P_1 と P_2 を並列に実行し、両者が 0 になった時 0 になる (*composition*)
		$P_1 + P_2$	P_1 と P_2 を並列に実行し、いずれかが 0 になった時 0 になる (*summation*)
		$A\ (y_1, ..., y_2)$	対応する引数を束縛して、定義されたエージェント A になる (*defined agent*)

図 1: π カリキュラスの概略 (a summary of π -calculus)

モデル化機能: エージェントの記述言語を特定しないことにより、エージェントの外部 (UIMS) からそれらを接続し、管理をすることが難しくなる。しかし、UIMS が管理すべきエージェントの内部構造モデルや現在の状態は、共通の形で表現されてエージェントの外側から把握し制御できることが必要である。

処理の委任: 応用とインターフェイスを分離して**対話独立性**を実現することが UIMS の基本概念である。すると、応用とインターフェイスの間の通信が非常に重くなるため、インターフェイス部で反射的に応答できる処理については応用部から委任することが必要とされている。このために、エージェントの移住の機能が必要であろう。

この他に当然の要求として、**機種独立性**や**分散透明性**などがある。また、応用対象がグループウェアである場合などは、**通信のスコープの制御機能**なども必要になる。これらの要求を、個々のエージェントの記述になるべく負担をかけずに実現したい。

従来用いられてきた方式の中で、リモートプロシージャコールは比較的言語独立性が高く、広く用いられている通信モデルであるが、上述のような対等な複数のエージェントの相互通信には向かない。多様な通信モデルを記述できるものとしては、Linda がある [6]。Linda では通信はタプルスペースと呼ばれる広域的なテーブルに対してタプルの追加と読み出しを介して間接的に行なわれる。このため 1 対 N の通信を含め多様な通信モデルが表現できる。ただ、我々の要求に対して適合しない点として、データの書き込みと読み出しに関する同期のメカニズムが固定されていて制御ができない、またデータの可視性を制限できないなどの点がある。

我々が実現した VIABUS は、送出されたメッセージに対して、受信者がメッセージのヘッダ部に対して正規表現によるマッチングを行って取り込むという通信モデルである。これによって、受信者数が 0 の場合を含む相手を特定しないマルチキャストができ、複数の対等なエージェント間の双方向通信ができる。しかし、この上でのエージェント間の通信の制御は、個々のエージェントに任されていた。また、各エージェントの内部の状態は、外部から制御することが不可能である。このため、これを共通の形て制御できる上位レイヤのプロトコルが必要であった。

3 π カリキュラスに基づく通信方式

エージェント間の複雑な通信の制御や、エージェントの制御構造のモデル、実行の状態などを表現するために、プロセス代数を用いることができる。記述にとどまらず、これを実行時の通信方式と考えると、強力な通信メカニズムを手にすることができる。そこで以下では π カリキュラスを用いた通信方式を提案する。

3.1 π カリキュラス

π カリキュラスは、Milner らによって設計された動的に生成消滅しながら通信し合うエージェントを記述するための並列計算モデルである [2]。π カリキュラスの式は一つのエージェントに対応し、それはまた並列型および逐次的型エージェントで階層的に構成される。エージェント間は、名前付けされたポートを介して通信し合い、ポート自体を受渡しすることができる。ポートの名前の通用範囲は制限することができ、局所的なポート名の受渡しに伴う通用範囲の自動的な拡大が起こる。

図 1に π カリキュラスの概略を示す。ここで P は任意の式でありエージェントである。例として、$\bar{y}x.P_1 \mid y(z).P_2$ は、2 つのエージェントが y を介して通信を行ないその結果 $P_1 \mid P_2\{x/z\}$ となる。ここで $P\{x/z\}$ は、P 中の z が x に束縛されていることを表す。一般の言語のような変数名とデータのような区別はなく、例えば $y(x).\bar{x}z.P$ は、y から x を受けとって、これをポート名として、x

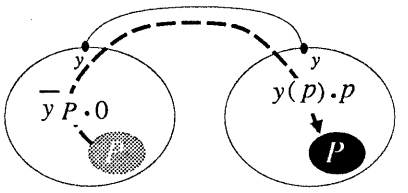

図 2: エージェントの送信 (agent passing)

から z を送出する。これは Linda における、以下のものと同等である。

in("y",&x); out(x,z);

名前の局所化の例として、$(y)(\overline{y}x.P_1 \mid y(x).P_2)$ では、この 2 つの並列エージェントは外部のエージェントに知られていないポートである y を介して通信する。局所的な名前と同一の名前が外部から受けとられた時は、局所的な名前を全て別の名前で置き換える (*name intrusion*)。また、局所的な名前を外部に送出する時、その通用範囲は自動的に受信者と送信者の間に広がる (*name extrusion*)。

3.2 ポートを介した π エージェントの送受信

π カリキュラスには、エージェントをポートを介して送る機能はない。すなわち、

$$(y)(\ \overline{y}P.0 \mid y(p).p)$$

のような式 (図 2のように動作する) を書くことはできない。Milner らはこの点について「π カリキュラスにおいては、エージェントの送信と等価なことが、ポートの送信で実現できる。もし π カリキュラスでエージェントの送信を行なうなら、その一つの解釈は、プロセスのテキスト (文字列表現) を送ることと同様な効果となる。これは、McCarthy による LISP の function parameter の扱いと同様であり、動的リンキングと呼ばれるものである。しかしながら、我々は静的なリンキングを採用したいのだ」と述べている [2]。

しかしながら、物理的に分散したエージェント間での通信の実現においては、π の式そのものをポートを介して受渡しできることには、大きなメリットがある。すなわち、物理的に分散したエージェント群を、通信に関する性能的な要求を満たしながら柔軟に協調動作させるには、物理的に分散したエージェント間でのエージェントの送受信が、強力な手段となるからである。

別のアプローチとして、処理系の「隠された」階層によって、分散したエージェント群を仮想的に単一空間上のエージェント群として処理することも考えられる。しかしその場合、物理的な分散構成を反映して通信の制御を行なうことを、π カリキュラスで記述することはできなくなる。我々はまさにそれを記述したいのだ。リンクの受渡しによる処理を基本としつつも、場合によりエージェントの受渡しができることが必要である。

物理的に分離したエージェント間でエージェントを交換できることは、遠隔のエージェントにアルゴリズムを送り込むことができることを意味する。これは、極めて柔軟な通信の方式の実現を可能にする。例えば、遠隔エージェント内で検索を行なって返送する、あるいは、複数のエージェントの間を渡り歩いて情報収集して返送する、さらには、遠隔エージェントに新たな機能を追加する、などなどである。実際、4 で述べるように、分散した VIA エージェント上での π エージェント同士の通信は、受信者側から送信者側に対して「送信者の環境で受信を行ないこれを受信者に送り出す π エージェント」を送出する形で実現されている。

3.3 外部仕様の共通な記述と実行

任意の記述言語で部品が書かれている状況で、各部品の枠組の把握と現在の状態の読み出しを行ないたい。そのために、(1) 全ての部品はその仕様レベルの記述を π カリキュラスで共通に行ない、実現レベルは言語は自由とする。部品の枠組が π で記述されているために、その部品全体を外部からは π のエージェントとして見なして扱うことができる。例として簡単なファイルサーバ部品の記述を示す。

```
SV ≡ fileserv(command).(
     [command=get]argument(arg).GET(arg,done)
   + [command=put]argument(arg).PUT(arg,done)
)
SERVER ≡ (done)(SV | done(x).SERVER)
```

GET および PUT は、処理の終了を done ポートに知らせるように定義されているものとする。またその実体は、部品本体の実現言語で記述されているものとする。この記述から、このファイルサーバ部品が fileserv というポートからコマンドを受け取り、get または put という command を一つづつ処理することが分かる (ここでは簡単のために、必要な molecular action などは省略している)。あるいは、

```
SERVER ≡ (done)(SV |
      (done(x).SERVER + cancell(x).SERVER))
```

ならば、done を待たずに、cancell して次のサービスに移れることがわかる。次に (2) 各部品の実行状態を任意の時点で読み出せることとする。読み出した状態も

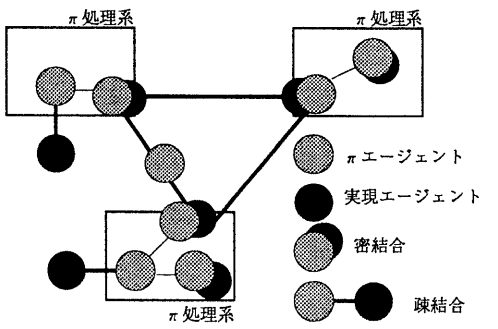

図 3: 多様なエージェント結合形態

πカリキュラスの式で表す。このために全ての部品は要求に応じて、その部品に含まれるπエージェント全体をテキスト表現にして送り出すポートを持つこととする。例えば、読み出した結果が

```
(argument(arg).GET(arg,done)|done(x).SERVER)
```

ならば、現在このファイルサーバ部品がgetコマンドのargument待ち状態であることがわかる。ここでエージェントの構造を読み出して文字列にして送出するメカニズムは、上述のエージェントの送受信と共通に実現できる。ただし、こちらの場合には送出されたエージェントが消滅することはない。

3.4 仕様と実現の結合方式

上述の例において、GETおよびPUTというエージェントの実現は、この部品の本体の記述言語により記述されている。πで記述された仕様と部品の実現との間の結合の方式としては、幾つかの方法が選択できるべきである。

密結合方式: これは、πの処理系と同一空間（プロセス）上に実現のコードをリンクする場合であり、基本的にはπ処理系と実現コードが同一の言語で記述されている必要がある。まず、実現エージェントを関数として実現し、πの処理系を制御の中心とする方法がある。また、πの構文を拡張して実現言語を組み込める形とし、組み込み構文を作る方法がある。実現のコードとπの処理系を静的リンクまたは動的リンクのどちらで結合するかについても選択がある。

疎結合方式: 実現エージェントは外部のプロセスとして動作する。π処理系のプロセスと実現のプロセスの間は、プロセス間通信メカニズム (pipe,socket, VIABUSなど) で結び、その間での通信のためのプロトコルを与える。このプロトコルは、実現エージェントの起動時と終了時に、{ 束縛 / 名前 } の組を受渡しするものとなる。

PIE ::= 0
| y¯x.PIE — $\overline{y}x.PIE$ の意
| y(x).PIE
| −.PIE — $\tau.PIE$ の意
| [x=y]PIE
| (x)PIE
| $PIE \mid PIE$
| $PIE + PIE$
| A(y_1,...y_n)
| A(y_1,...y_n) = PIE エージェントの定義
| (PIE)
| y¯PIE PIEの文字列表現をyに送出して0に
| x! xの文字列をエージェントとして実行
| PIE_1; PIE_2 PIE_1を実行してPIE_2へ

A ::= [A-Z][A-Za-z0-9_]*
x ::= y
y ::= [a-z][A-Za-z0-9_]*
| [a-z][A-Za-z0-9_]*@[A-Za-z0-9_]*
| <.*> <と>で囲まれた任意の文字列

図 4: PIEの構文 (the summary of PIE)

複数の部品の間の接続に関しては、全ての部品が同一の空間内に静的に結合されている場合や、仕様が集中されて各部品が疎結合になっている場合、部品と仕様が密に結合されて部品同士は疎に結合されている場合、などの場合が考えられる (図3)。重要なことは、これらのいずれの物理的な結合方式をとっても、結合された部品群が等価に動作することである。

別々の言語で実現された複数のπ処理系があるとき、それらの間はπエージェントの交換で通信する。

4 実現

πカリキュラスの評価器としてPIE(PI calculas Engine)を試作した。またPIEの外界とのインターフェースとして、PIEにC言語の実現コードを差し込んで使用するためのPIERCEライブラリ、VIABUSに接続して外部との通信を代行するためのPIECEを実現した。

4.1 PIE: πカリキュラスエンジン

PIE処理系は、与えられた入力からPIEスクリプトと呼ぶ式の列を読み、処理し、評価結果の式を出力するものである。図4にPIEスクリプトの構文を示す。ここに示すように、エージェントの送信のための y¯P、およびポートから受信したエージェントの実行のための x! が追加さ

```
% pie -d2 << EOF
 # SIMPLE CLIENT SERVER MODEL
 SV = info(r).r~_info_.SV;
 # info から r を受け取り r に info を返すループ
 CL = (r)info~r.r(i).putl~i.0;
 # info に送った r から i に結果を受けて putl に出力
 CL | SV;
EOF
[1] CL|SV
[2] (r)info~r.r(i).putl~i.0|info(r).r~_info_.SV
[3] (r)(r(i).putl~i.0|r~_info_.SV)
[4] (r=_info_)(r(i).putl~i.0|SV)
[5] (r)(putl~i.0{_info_/i}|info(r).r~_info_.SV)
 info_
[6] (r)info(r).r~_info_.SV
(r)info(r).r~_info_.SV
%
```

図 5: PIE スクリプトの記述例と実行例

れている。図 5に、PIE スクリプトで記述した簡単なクライアントサーバモデルの例と、これを PIE でトレースしながら実行した例を示す。

PIE はプロセス木を評価するエンジン部と、パーサおよび逆パーサから構成されている。エージェントの文字列表現を PIE 処理系の間で交換することなどのために、プロセス木としては構文解析木を直接用いている。PIE の基本構文は非常に単純であるが、実用の言語として使うための機能拡張を考えてパーサは yacc を用いて実現した。

π カリキュラスをベースとしながら実用の言語を実現するために、PIE は処理系の外部にあるポートおよびエージェントとの通信機能を提供している。それは、4.2 に述べるような関数を使用して定義できる。外部ポートおよびエージェントの追加は随時行なうことができ、これは PIE プロセス間で π エージェントを交換する場合などにも使用される。

名前のスコープはエージェントの木構造で表現されているが、ここで問題となるのは、外部エージェントや外部ポートに局所的な名前を渡す場合である。この場合、外部エージェントとの局所的な名前の共有は、通常の name extrusion のような木構造的関係で表現できない。そこで、この場合には局所的な名前 x に対する広域的一意名 x' を作り出し、元の x への参照をその x' に付け変え、外部ポートや外部エージェントにはこの x' を送出する。内部エージェントと外部エージェントの間を $P_{\text{internal}} \parallel P_{\text{external}}$ のように区切って表わすと、外部ポートへのエージェントの送出は

(x)(y~P|Q) ‖ y(p).p! → Q{x′/x} ‖ p!{x′/x}

のように動作する。

PIE プロセスはその外部からは仮想的に、固有のポート名を持った π エージェントとして見えるようになっている。そのポート名は inport@$PIEuniqueID$ の形で、このポートに π の式 (エージェント) を送り込むと、それが外部エージェントとして実行される。これを基礎として、PIE プロセス間の通信が実現されている。

4.2 PIERCE ライブラリ

C プログラムからは、PIERCE ライブラリを用いて、外部ポートや外部エージェントの定義を行なうことができる。

(1) **外部ポートの定義**

```
int pierce_port(io,portname,portfunc)
     int io;
     char *portname;
     int (*portfunc)();
```

pierce_port 関数は、portname という外部ポートへの入 (出) 力が起きたときに、portfunc を呼び出すことを定義する (portname~obj または、portname(obj) という prefix から呼ばれる)。io 引数は、このポートが入力ポートであるか出力ポートであるかを指定する。以下に "getline" という名前のポート – これは、標準入力から一行読み込んで返すポートである – の定義例を示す。

```
#include <stdio.h>
#include <pie.h>
int Getline(io, subj, obj)
     int iog; char *subj,*obj;
{
     fgets(obj,PIE_NAME_SIZE,stdin);
     return PIE_PORT_READY;
}
...
   pierce_port(PIE_IN, "getline", Getline);
```

portname に '*' が含まれている場合、(他に exact match がない場合に) そのポート名を正規表現として (* は長さ 0 以上の任意の文字列) 適合した時に、このポートが呼び出される。例えば、

```
piece_port(PIE_IN,"*",ViaIn);
```

とすれば、広域的なポート名からの入力を全て自動的に ViaIn 関数に振り向けるようにできる。そして、ViaIn を VIABUS からの入力ポートとして実現すれば、VIABUS に接続されたエージェント間での通信がトランスペアレントに実現される。

(2) **外部エージェントの定義**

```
char *
pierce_agent(agentname,agentfunc,p1, ···,pn,0)
     char *agentname,*p1, ···,*pn;
     int (*agentfunc)();
```

pierce_agent 関数は、agentname という名前のエージェントを定義するための π 式を返す。p1,...,pn は、エージェントの引数名である。返された π 式を以下の pierce() 関数で評価することで、実際に定義される。

(3) PIE 式の評価

```
char *pierce(alien,pexp)
     int alien;
     char *pexp;
```

pexp で与えられた π 式を評価して、その結果の π 式を返す。alien 引数は、この pexp を外部エージェントとするか、内部エージェントとするかを指定する。以下に、"SYSTEM"という名前のエージェントの記述例を示す。これは、system() 関数によりシステムコマンドを実行し、終了を done ポートに報告するエージェントである。

```
#include <pie.h>
System(command,done)
   char *comand,*done;
{
   system(command);
   pierce_out(done,"ok");
}
...
   pierce(PIE_ALIEN,
          pierce_agent("SYSTEM",System,
                       "command","done",0));
```

ここで pierce_out(char *port,char *string) は、広域名である port に string を送出する関数である。このように定義されたポートとエージェントを、以下のように利用することができる。これは、標準入力から 1 行入力しては、それをシステムコマンドとして実行するエージェントである。

```
SYS = getline(c).SYSTEM(c,done)|done(x).SYS
```

実現エージェントと PIE とを疎に結合する場合には、この SYSTEM と同様に外部プロセスとして実現エージェントを起動し、PIE との間を pipe などにより双方向に結合すればよい。

4.3 VIABUS 上の通信を提供する PIECE

複数の PIE 処理系を並列動作させ、ネットワークワイドに通信させるためのエージェントとして PIECE を作成した。PIECE は、外部エージェントとして VIABUS 経由での通信を内部エージェントのために代行する。PIECE 同士は VIABUS 経由で π の式（エージェントのテキスト表現）を交換し合いながら通信を実現する。

PIECE は export と inport の 2 つの外部ポートを提供する。内部エージェントは PIECE に対して export$^-$*PIE*

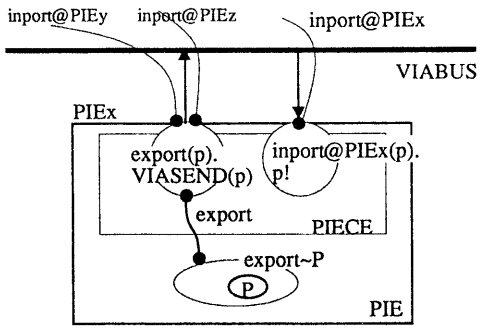

図 6: PIECE の構成 (the framework of PIECE)

により PIE 式を送り、PIECE がこれを VIABUS に流す。一方、inport は他の PIE 上の PIECE エージェントに inport@*PIEuniqueID* の形で知られており、ここに向けて export で受けた式が送られる。PIECE は VIABUS に流れて来た *PIE* 式を取み、inport(p).p! を実行する (図 6)。

inport と export を用いて、遠隔の PIE の上にある *remote-port* ポートからの受信は以下のような式で実現できる。

(resp)export$^-$(*remote-port*(x).export $^-$(resp$^-$x.0))

これは、「遠隔の PIE で受信を行なって、その結果を元の PIE に返信する」 PIE 式を、export に送出することを表している。この実現では、ポートへの送信者側のエージェントは、同一のインターフェイスで外部／内部エージェントと通信している。図 7に、他の PIE 上のポートからの受信時のエージェントの動作例を示す。ここで、‖ の右辺は、外部エージェントである。

PIE 処理系の中で未知の *rport* ポートからの受信を、自動的に他の PIE からの受信とするためには、

rport(x)
→ (r)(export$^-$(*rport*(x).export$^-$(r$^-$x.0)) ‖ r(x).0)

という書換えを自動的に行なえば良い。4.2 で述べたように '*' を含む外部ポート名を定義してその処理を上述のようにすることで、これが実現できる。

4.4 実現上の問題点

実用的な通信媒体として使用するとき、全てのデータを PIE のポート経由で送るのは得策でない。特にサイズの大きなデータの送信は効率上問題があり、さらに π カリキュラスでは名前とデータの区別がないため処理系の異常動作を招く恐れがある。これは、4.2 の例で示した "getline"の実現のように、外部ポートから受信する場合に問題となる。ポート名として使用することを禁止する名前

```
EX = export(p).VIASEND(p)                          # single export action
IN = inport(p).p!                                  # single inport action
R0 = (r)(export~(rport(x).export~(r~x.0)) | r(x).R) || EX|IN  # receiver agent
          S0 = rport~x.0 || IN|EX                              # sender agent
# now the receiver is invoked
R1 = (r)(export~(rport(x).export~(r~x.0)) | r(x).0) || export(p).VIASEND(p)|IN
              # r is renamed to unique name "r@R" to be passed to external port
R2 = (0 | r@R(x).0) || VIASEND(rport(x).export~(r@R~x.0))|IN
R3 = (0 | r@R(x).0) || 0 | IN
               # the exported agent intrudes into the sender's environment
            S1 = rport~z.0 || rport(x).export~(r@R~x.0) | EX
            S2 = 0 || export~(r@R~z.0) | export(p).VIASEND(p)
            S3 = 0 || VIASEND(r@R~z.0)
            S4 = 0 || 0
# the exported agent returns with received object
R4 = r@R(x).R || r@R~z.0
R5 = R{z/x} || 0     # the receiver gets the returned object
```

図 7: VIABUS 上でのπエージェントの移動 (an example of agent passing)

の形式の導入、および効率的な通信のためにデータ構造を導入する必要があると考えられる。

また、分散したエージェント間でのエージェントの交換において、エージェントの物理的な宛先の判断を効率化するために、名前を構造化する必要があると思われる。(例えば広域ネットワークで用いられているドメイン記法を採用すれば、広域ネットワーク上での PIE の分散を支援できる可能性がある)

密結合により実現エージェントを PIE に結合する場合に、物理的に並列な実行を必要とするエージェントのために、コルーチン型の実行制御を導入する必要がある。

5 むすび

プロセス代数を直接的に通信メカニズムとして使用する試みについて報告した。通信の性能などを含めての実用性については現在のところ明確でないが、その柔軟な記述力は極めて魅力的であり、応用範囲は広いと考えている。

UIMS への応用としては、PIE で対話制御アルゴリズムを記述したり、利用者のモデルや固有のカスタマイズを記述したりすることが考えられる。そのようにして、UIMS 全体をπカリキュラスという共通のベースの上で動作させることで、対話型ソフトウェアの構成部品の実現や接続に関する柔軟性を保ちながら統合する UIMS が実現できるであろう。

ただ、現在の PIE 言語は書き易さや読み易さの面で問題がある。そこで PIE 言語に必要な抽象化機能の検討と、構造エディタなどによる対話的なインターフェイスの提供を当面の課題と考えている。

謝辞

本研究の機会を与えていただいている、弓場敏嗣情報アーキテクチャ部長に感謝します。また、Linda に関してご教授いただいたシャープの増井俊之氏、πカリキュラスの応用に示唆を与えてくれた情報ベース研究室の小方一郎氏他の諸氏と言語システム研究室の高橋孝一氏に感謝します。

参考文献

[1] 佐藤豊, 真野芳久: UIMS の試作とそのニュースリーダへの応用, 情報処理学会 ソフトウェア工学研究会 SE77-14, pp.81-88 (1991).

[2] Milner,R. Parrow,J. and Walker,D.: A Calculus of Mobile Processes, LFCS Report Series ECS-LFCS-89-85 (1989).

[3] Hartson,H.R. and Hix,D.: Human-Computer Interface Development: Concepts and Systems for Its Management, ACM Comp. Surv. Vol.21, No.1 (1989).

[4] Manheimer,J.M., Burnett,R.C. and Wallers,J.A.: A Case Study of User Interface Management System Development and Application, CHI'89 Proceedings (1989), pp.127-132.

[5] Coutaz,J.: Architecture Models for Interactive Software: Failures and Trends, Proc. of the IFIP TC2/WG2.7 Working Conf. on Engineering for Human-COmputer Interaction (1989).

[6] Carriero,N. and Gelernter,D.: How to Write Parallel Programs: A Guide to the Perplexed, ACM Computing Surveys, Vol.21, No.3, pp.324-357 (1989).