

NueLinda Interpreter in NueLinda

——非均質システム NueLinda インタプリタの自己記述——

奥 乃 博[†] 明 石 修[†]
村 上 健 一 郎[†] 天 海 良 治[†]

分散システムでは、分散カーネルがすべてのプロセッサに配置される。プロセッサ、オペレーティングシステムなどが異なる非均質分散システムでは、分散カーネルの移植性が重要な課題となる。NueLindaは、非均質分散システムを対象に Linda モデルを多重タプル空間に拡張し、タプル演算をタプル空間のクラスタに拡張した計算モデルである。Linda モデルでのタプル空間は受動的な分散型データベースであるのに対して、NueLinda での個々のタプル空間は、タプル演算処理のために他のタプル空間と協調をするというアクティブな性質を持つ。アクティブタプル空間を実現するインタプリタは、各タプル空間に付随した仮想マシンの集合によって実現される。本稿では、NueLinda カーネルの高移植性を達成するために、NueLinda インタプリタを NueLinda 自身によって記述するという自己記述について報告する。また、NueLinda インタプリタ内での並列処理による高速化についても報告する。NueLinda の自己記述が可能になったのは NueLinda (と Linda) の構造の単純さによる。

NueLinda Interpreter in NueLinda

—Self-Description of Interpreter for Heterogeneous Distributed System NueLinda—

HIROSHI G. OKUNO,[†] OSAMU AKASHI,[†] KEN'ICHIRO MURAKAMI[†]
and YOSHIJI AMAGAI[†]

High portability is required for a kernel of a heterogeneous distributed system, since such a system consists of various kinds of hardwares and operating systems. *NueLinda* is an extended Linda computation model with multiple tuple spaces. *NueLinda* tuple operations are so extended as to work on clustered tuple spaces. While a tuple space in Linda is passive like distributed database, that in *NueLinda* is *active* in the sense that it cooperates with other tuple spaces to execute tuple operations. The *NueLinda* interpreter is composed of a set of virtual machines associated with each tuple space in *NueLinda*. In this paper, the *NueLinda* interpreter is described in *NueLinda* itself. This self-description is to attain high portability of the interpreter. It also exploits parallelism to avoid inefficiency caused by high-level description. All data the interpreter uses are tuples and thus it is very easy to construct meta-interpreter. This is possible due to the simplicity of *NueLinda* (and Linda) structure.

1. はじめに

人工知能研究では、人間のモデル化とその工学的な実現・応用が中心であるのに対して、最近注目されている分散人工知能では、人間の社会のモデル化とその工学的な実現と応用が中心課題である。すなわち、複数のエージェントが協力あるいは協調して問題を解決するための枠組の研究である。協調問題解決では、問題の分割とエージェントへの割り当てはあらかじめ決

まってはならず、動的に行われる。したがって、メッセージの送り先は動的に変化するので、並列オブジェクト指向のメッセージ伝達による実装は、協調問題解決には向いていない。

協調問題解決の計算モデルとしては黒板モデル^{10), 11)}が使用されることが多い。黒板モデルでは、知識源 (knowledge source) と呼ばれるエージェントが黒板 (knowledge base) というグローバルデータベース (あるいは計算の場) を介して問題解決を行う。黒板はすべてのエージェントで共有される論理的に均質なデータベースであり、エージェント間の通信は黒板に書き込むことによって、間接的に行われる。

[†] 日本電信電話(株)基礎研究所
NTT Basic Research Laboratories

共有空間を用いて、プロセス間の通信あるいはデータの交換を行う計算モデルとして、Linda モデルがある⁴⁾。Linda モデルでは、共有のデータベースをタプル空間と呼び、タプル空間にタプルを書き込み、それを必要なプロセスが読み込むという生成的通信 (generative communication) によって、協調的な計算が行われる。すなわち、タプル空間という仮想化された論理データ空間上で、演算を並列実行することによってシステム全体の計算の高速化を狙っている。Linda モデルでのタプル演算は6種類と極めてコンパクトであり、しかも、並列、並行処理での基本演算である通信・同期・プロセス生成がカバーされている。

Linda モデルはトランスパレンシーが優れているものの、以下のような問題点を挙げることができる。

- (1) データや通信の局所性が考慮されていない。
 - (2) グローバルなタプル空間だけでは、データアクセスのオーバーヘッドが大きい。
 - (3) コンピュータネットワーク上での通信プロトコルがない。
 - (4) 処理系カーネル構築のモデルがない。
- (1), (2) というスケラビリティの問題点を解決するために、我々は、クラスタ化タプル空間を導入した NueLinda モデルを提案してきた^{11), 8), 13)}。(3), (4) は、インタオペラビリティの問題であり、以下で議論する。

コンピュータネットワークの発達により、多種多様なハードウェア、種々のオペレーティングシステム、あるいはアプリケーションがネットワークを介して複合的なシステムとして使用されるようになってきた。このようなシステムの1つの特徴は、その構成要素の非均質性 (heterogeneity) にある。非均質システムは、次世代の計算機アーキテクチャの1つの側面としても注目を浴びている。従来から非均質性を吸収するために様々な仮想化技法あるいは通信プロトコルが提案されてきた。Linda モデルでは、各処理系が独自のプロトコルを使用し、TCP/IP 上での標準的なプロトコルはまだ提案されていない。これが上記の(3)の問題点である。これを解決するために、我々は Linda を拡張した NueLinda モデルのための通信プロトコル TOPS を提案してきた⁹⁾。TOPS は業界標準となりつつある TCP/IP 通信プロトコル群を用いている。

システムの普及を促進するためには、移植性の高い処理系カーネル (たとえば、インタプリタ) を構築す

るとともに、カーネルに拡張性を持たせることが必要である。これが Linda の問題点の(4)である。処理系カーネルの移植性を高めるためには、使用するハードウェアやオペレーティングシステムに依存しないようにシステムを記述することである。また、拡張性を高めるには、システムをインタプリタで作成し、そのインタプリタをメタインタプリタで制御する方法がある。我々は、Linda の4番目問題点を解決するために、NueLinda インタプリタを自分自身で記述するというアプローチをとる¹⁴⁾。自己記述されたシステムは移植性は高いが、その性能がハンドコーディングしたシステムよりも劣るという可能性が高い。したがって、自己記述するときにはシステム内での並列性が容易に引き出せるように記述すべきであり、自己記述レベルでの最適化も行う必要がある。

Linda モデルでのタプル空間は受動的な分散型データベースである。一方、NueLinda での個々のタプル空間は、タプル演算処理のために他のタプル空間と協調をするというアクティブな性質を持つ²⁾。したがって、アクティブタプル空間を管理するインタプリタは、各タプル空間に付随する仮想マシンの集合によって実現する⁹⁾。

本稿では、NueLinda インタプリタの NueLinda による自己記述という構成法について報告する。なお、NueLinda は Lisp 言語 TAO/ELIS で実装されているので、本稿での記述には TAO を用いるが、提案する処理モデルは記述言語には依存しない。以下、第2章で NueLinda について説明し、例とその実行過程を示す。第3章で処理モデルについて説明し、第4章で NueLinda 演算の実現方法について、第5章で多重タプル空間での演算の実現方法について述べる。第6章でメタインタプリタについて説明する。

2. NueLinda とは

2.1 Linda モデル

NueLinda のベースとなった Linda モデルでは、仮想的な単一空間であるタプル空間 (tuple space, TS) を持ち、それがすべてのプロセスで共有される。ユーザプロセスは、タプル空間を介し、オブジェクトであるタプルをやりとりすることによって、並列計算を実現する。タプル空間は、仮想化された共有メモリであるので、その実体は共有メモリであろうと、ネットワークを介して接続されているものでであろうとユーザは意識する必要がない。表1に Linda モデルにお

表 1 Linda プリミティブ
Table 1 Linda primitives.

演算	Operation
rd	タプルの値を TS から読む.
rdp	タプルの値を TS から読む. なければ戻る.
in	タプルを TS から読み, 取除く.
inp	タプルを TS から読み, 取除く. なければ戻る.
out	タプルを TS に出す.
eval	引数を別プロセス中で評価し, その結果をタプルとして TS に出す.

ける 6 個* のプリミティブを示す (これですべてである).

タプルは, “(論理名 データ…)” で表現される. タプルには rd, in を用いてパターン照合によりアクセスする. rd, in の引数として与えられた変数 (“?” が前につく) の具体的な値は, out で作成されたタプルの値とのパターンマッチによって決まる. out で変数を書いた場合には, あるいは, rd, in の引数として変数を与えると, その部分はどの値ともマッチし, そのマッチした値が変数に代入される. ただし, 変数と変数はマッチしない. また, パターンの論理名は具体的な値でなければならない. マッチするタプルが複数存在する場合には, そのなかから非決定的に 1 つのタプルが選ばれる. 逆に, マッチするタプルが存在しなかった場合は, 適当なタプルが TS に出されるまで待ち, 実行がブロックされる. rdp, inp は, rd, in とは異なり, マッチするタプルがなかったときにそのようなタプルを待つのではなく, 該当するタプルがないという情報を返す. タプルへのアクセス同期機構はプリミティブ自身を持つ.

Linda のプログラムとして食事をする哲学者 (dining philosophers) 問題を解く NueLinda のプログラムを付録 A に示す**. ここでは, 5 人の哲学者が 5 本ある箸のうちどれか 2 本を使用して食事をする. 箸 1 本に別々のタプルを用意し, また, デッドロック防止用のチケットとしてタプルを人数より 1 以上少ない個数だけ用意する (図 1).

2.2 クラスタ化された NueLinda モデル

NueLinda モデルは Linda モデルのスケラビリティ上の問題を解決するために提案してきた. Nue-

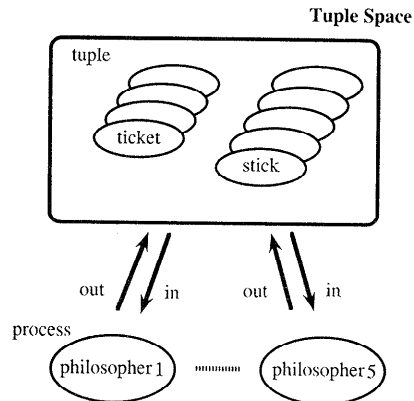


図 1 哲学者プログラムでのタプル空間
Fig. 1 Tuple space for dining-philosopher program.

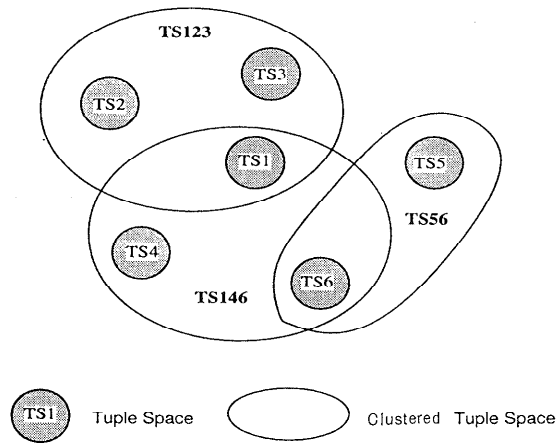


図 2 クラスタ化されたタプル空間
Fig. 2 Clustered tuple spaces.

Linda では, タプル空間は複数に分割され, さらにクラスタ化される. クラスタ化タプル空間を以下では単にクラスタと呼ぶ. 図 2 には TS1 から TS6 までの 6 個のタプル空間と, 3 つのクラスタ TS123, TS146, TS56 が示されている. クラスタ TS123 はタプル空間 TS1, TS2, TS3 から構成される. 図に示されるようにクラスタ同士はオーバラップしていてもよい. クラスタのオーバラップを利用して, 情報伝達を行うことができる. 今, タプル空間 TS3 がクラスタ TS123 に対してあるタプルを書き出したとしよう. クラスタ TS123 に対する書き出しはその構成要素であるタプル空間 TS1, TS2, TS3 で実行される. 次に, TS1 が今行われた変更に基づいて, さらなる計算を行い, 今度はクラスタ TS146 でタプルの書き出

* 文献 3) では, inp, rdp はなく, 4 種類であったが, 後にこの 2 種が追加された**.

** プログラム中では Lisp 関数 eval との名前の衝突を防ぐために eva を使用している**.

しをする。すると、その書き出しは、クラスタ TS146 に属するタプル空間 TS1, TS4, TS5 で実行される。

クラスタの定義は動的に与えることができる。クラスタが動的に変化することによって、情報の伝播が様々に変化しながら、全体としての処理が進んでいく。たとえば、TS123 で生じた変化が、TS1 経由で加工されて TS146 に伝わり、それがさらに加工されて TS6 経由で TS56 に伝わる。我々のクラスタ化の動機は、「オーバーラップする近傍系」として社会システムをモデル化するコネクティクス¹⁵⁾という新しい並列計算パラダイムの具現化の1つととらえている。クラスタ化の応用としては、動的な計算の伝播という見方だけではない。たとえば、タプル空間で複数のコピーを持つことによって信頼性向上を狙い、エラーが生じた時点でタプル空間を動的にクラスタ化 (view と呼ぶ) する研究もある¹⁸⁾。

NueLinda では、Linda の演算の一部 (out, eval) をクラスタ化タプル空間に拡張し、クラスタが指定できるようにした。(なお、in, rd のクラスタ化空間への拡張については第8章で触れる。) クラスタの指定方法は2通りある。

- (1) 基本演算 (out, eval) の呼出しで neighborhood というキーワード引数を用いてクラスタを指定する。
- (2) 関数 neighborhood でそれ以降の演算のデフォルトのクラスタを指定する。

ここで、(2)の方法を説明しよう。

```
(neighborhood '(("TS1" "TS2" "TS3")))
```

(a)

```
(neighborhood :all)
```

(b)

```
(neighborhood [:local nil])
```

(c)

(a)によって、図2の TS123 をデフォルトのクラスタと指定できる。従来の単一タプル空間は、全タプル空間を意味するので、(b)のように指定する。ローカルなタプル空間の指定は、(c)で行う。なお、デフォルトクラスタの初期値はローカルタプル空間である。また、nil のときはローカルタプル空間である。

食事をする哲学者のプログラムでクラスタの説明をしよう。前節で示したプログラムでは、どの箸もすべての哲学者からアクセスできるので、箸に番号をつけて自分の脇にあるものだけを取るようにした。クラス

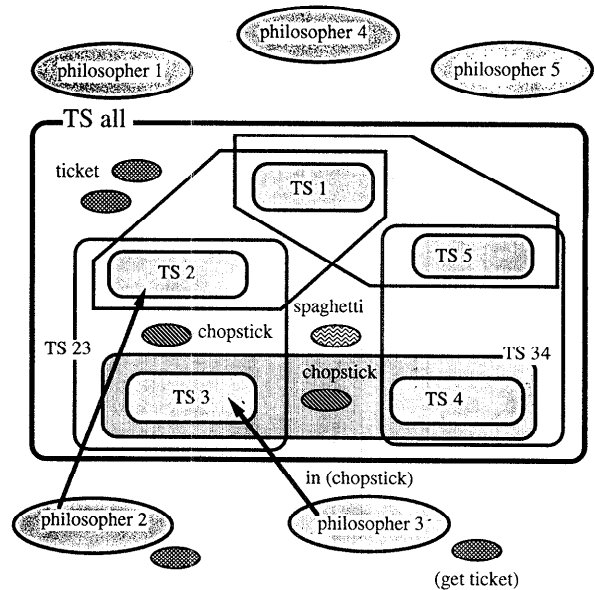


図3 哲学者プログラムでのクラスタ化タプル空間
Fig. 3 Clustered tuple spaces for dining-philosopher program.

タを使用すれば、箸には番号をつけなくても隣り合った2人の哲学者しか利用できないようにできる。それぞれの哲学者に対応してタプル空間を作成し、それを哲学者タプル空間と呼ぶ。箸を隣り合う2つの哲学者タプル空間からなるクラスタに置く、哲学者はそれぞれ箸を2回 in すればスパゲッティが食べられる。このプログラムを付録Bに示す。なお、スパゲッティは前節のプログラムと同様に5人の哲学者のタプル空間からなるクラスタに置かれているので、誰かが食べる時には排他制御が生じる。タプル空間の構成を図3に示す。

3. NueLinda 処理モデルとデータ構造

NueLinda では、個々のタプル空間に「仮想マシン」(virtual machine) が付随しており、その上でインタプリタが走る (図4)。仮想マシン間の通信はプロトコル群 TOPS⁹⁾ による。通信のモニタ機能は、プロトコル群を解釈することによって実現される。

NueLinda インタプリタは、それが使用するすべてのデータたとえば、タプル管理用データ、ブロックされたプロセス管理データといったタプル間の状態—を、ユーザ空間のデータから分離して管理するのではなく、タプル空間のタプルで表現する (図5参照)。また、デバッグ機能を提供するメタインタプリタは、

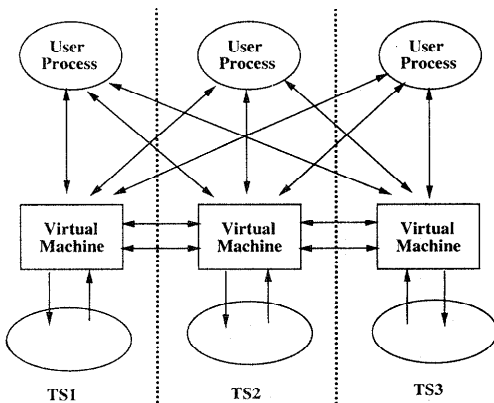


図 4 NueLinda 処理モデル
Fig. 4 NueLinda processing model.

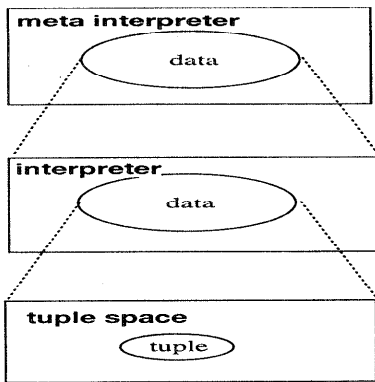


図 5 NueLinda インタプリタの階層構造
Fig. 5 Hierarchy of NueLinda interpreter.

インタプリタのデータを変更することによって、実現されている。メタインタプリタの使用するデータも、ユーザ空間、インタプリタのデータ空間から分離されているのではなく、タプル空間のタプルとして表現されている。

ユーザ、インタプリタ、メタインタプリタが扱うデータは同じタプル空間に格納されているが、その表現方法は以下に示すように異なっている。

- ユーザ指定タプル：ユーザ指定タプルには、接頭語 `sys:user-data` がつく。
(`sys:user-data` 論理名 データ …)
- マスタタプル：クラスタにあるタプルに `in`, `rd` 等のアクセス競合が生じたときに排他制御のために用いられるキーである。
(`sys:master` 論理名 データ …)
- キュータプル：待っているプロセスがあるタプルが登録される。

表 2 タプルの属性
Table 2 Tuple attributes.

Attribute	Contents
primary	primary タプル空間へのポインタ
neighborhood	コピーがあるタプル空間の集合
data	タプルデータ
derivation	タプル作成者
time-stamp	タプル作成時
time-to-live	寿命
trap	trap 条件
log	アクセス履歴

(`sys:waiting` リクエスト 論理名 データ …)

- ロックタプル：タプルへのアクセスに対する排他制御は、論理名単位で行い、全タプル空間のロックはしない。

(`sys:lock` 論理名)

タプルは、クラスタ化やデバッグのために、`primary`, `neighborhood`, `data`, `trap` 条件, ログ, 寿命などの属性を持っている。タプル属性を表 2 にまとめる。`primary` 属性は、タプルのコピーが存在するときに、マスタタプルの置かれているタプル空間を保持する。`neighborhood` 属性は、コピーの存在するタプル空間のリストを保持する。

4. NueLinda 演算の実現

以下では、まず、NueLinda インタプリタで用いられる内部関数を説明し、次にユーザが使用する NueLinda 演算を説明する。ユーザが使用する NueLinda 演算ではクラスが指定できるのに対して、内部関数ではクラスではなく、タプル空間しか指定できない。また、その名前には `basic:` という接頭辞が付く。

内部関数の処理は同期フェーズ、非同期フェーズに分かれている。同期フェーズは、処理要求プロセスと同期が必要な部分の処理を行い、非同期フェーズでは同期が不要な部分の処理を実行する。このように処理を同期フェーズと非同期フェーズに分離することにより、処理要求プロセスの待ち時間を減らし、処理プロセスとの並列性を高めている。

4.1 内部関数 `basic:out`

タプル空間に対する `out` である。第 2 引数はオプション引数であり、指定のない場合、あるいは、`nil` のときには、ローカルタプル空間となる (`neighborhood` と同じ)。

(`basic:out` タプル [タプル空間])

が与えられると以下のように実行される。

● **basic:out** の処理要求をしたプロセス側

- (1) 指定されたタプル空間に **basic:out** の実行を要求する。

この処理要求受理の返事 **ACK** はインタプリタで処理する。

- (2) 実行を継続する。

● **basic:out** の処理要求をされたタプル空間のインタプリタ

- (1) **basic:out** の要求があったタプルを作成する。

- (2) **basic:out** したプロセスに要求受理の返事 **ACK** を返す。

—以下の処理は非同期フェーズ—

- (3) 初使用の論理名をもつタプルに対しては、ロック・エントリに相当する

`'(sys:lock, 論理名)`

というロックタプルを作成する。

- (4) 作成されたタプルに対するマッチを非同期で起動する。

4.2 マッチの実行

マッチの実行は以下のように行われる。

- (1) `(basic:inp '(sys:waiting
?request, 論理名, パターン要素))`
を実行する。

- (2) マッチする要求がある場合には、その要求を再実行する。(再実行したときに、空振りに終わる可能性は残る。)

- (3) マッチする要求がない場合は何もしない。

4.3 内部関数 **basic:in**

タプル空間に対する **in** である。第2引数はオプション引数であり、指定のない場合、あるいは、**nil** のときには、ローカルタプル空間となる。

`(basic:in タプル [タプル空間])`

が与えられると以下のように実行される。

● **basic:in** の処理要求をしたプロセス側

- (1) 要求受理の返事 **ACK** が返ってくるまで実行を中断する。

- (2) 指定されたタプル空間に **basic:in** の実行を要求する。

- (3) **ACK** を受け取ったら実行を再開する。

● **basic:in** の処理要求をされたタプル空間のインタプリタ

- (1) **basic:in** 要求で指定されたタプルの論理名

を持つタプルの集合にロックをかける。

- (2) そのタプルにマッチするタプルがあるかを調べる。もしあれば、マッチするタプルを1つ適当に選び、そのタプルを要求元のプロセスに返すとともにタプル空間から削除し、ロックを解除する。

- (3) もしなければ、ロックを解除し、

`(basic:out '(sys:waiting
in, タプル))`

によってキュータプルを作成し、マッチするようなタプルが現れるまで (**out** されるまで) 待つ。

4.4 内部関数 **basic:inp**

basic:in と違うのは、マッチするタプルがないときには「見つからなかった」ことを示す **nil** という値を返す点である。タプルが見つかったときには、タプルが返される。その場合は **non-NIL** のリストであるので、**basic:inp** を起動したプログラムは所望のタプルが見つかったかどうかを簡単に判断することができる。

4.5 内部関数 **basic:rd, basic:rdp**

basic:in と違うのは、マッチするタプルを返すときに、それを削除しないことである。**basic:rdp** の実行は、**basic:inp** と **basic:rd** とから明らかであろう。

5. 多重タプル空間での演算の実現方法

クラスタに拡張した、**out**, **in**, **rd** 等の処理の実現方法を内部関数を用いて説明する。なお、**rd(p)** の処理は **in(p)** より明らかであるので、割愛する。これらの演算はユーザプログラムだけでなく、インタプリタでも使用される。

5.1 **out** 演算の実行

out は以下のように指定される。クラスタを指定する **:neighborhood** のキーワード引数はオプションであり、なければローカルタプル空間が取られる。なお、クラスタには、ローカルタプル空間が含まれていなければならない。もし、含まれていない場合には、クラスタに付加される。このローカルタプル空間を **primary** タプル空間と呼ぶ。

`(out タプル [:neighborhood タプル空間リスト])`

が与えられると、**out** の処理要求をしたタプル空間のインタプリタは以下のように実行する。

- (1) **out** を要求されたタプルに対するマスタタプル

を primary タプル空間に basic:out で作成する。

(2) 実行を継続する。

——以下の処理は非同期フェーズ——

(3) 指定されたクラスタに属する個々のタプル空間にタプルのコピーを basic:out で作成。

(4) それらのタプル空間からの ACK を待つ。

図 6 に (out “foo” :neighborhood (“TS1” “TS2” “TS3”)) をタプル空間 TS1 上で実行したときのタプル空間間での演算を図示する。

コピーが basic:out で作成されると、前節で説明したようにすぐにマッチが実行される。もし、そのタプルを待っているプロセスがクラスタに属するすべてのタプル空間にあるときには、basic:out が先に実行されるタプル空間で待っているプロセスが優先されて再開される可能性が高い。これを避けるためには、basic:out の実行順序を毎回変えるように拡張することもできる。

5.2 in 演算の実行

演算“(in タプル)”が与えられると以下のように実行される。

(1) ユーザプロセスから in 要求が来たら、ローカルなタプル空間にマッチするタプルがあるかを basic:rdp で調べる。

(2) もしマッチが成功すれば、マッチしたタプルの primary 属性から得られる primary タプル空間に対して次の演算でマスタタプルを取りに行く。

(basic:inp

‘(sys:master, タプル)

:neighborhood primary タプル空間)

これは、ローカルコピーへのアクセス権を獲得す

るためである。

●マスタタプルが取れず、アクセス権が確保できなければ、

(basic:out ‘(sys:waiting in , タプル))

でキュータプルを作成し、マッチするようなタプルが現れるまで待つ。

●マスタタプルが取れば、アクセス権が確保できたことになる。primary タプル空間からはマスタタプルが削除されていることに注意。ユーザプロセスにローカルコピーを返す。

——以下の処理は非同期フェーズ——

ローカルコピーをローカルタプル空間から削除する。また、マスタタプルの neighborhood 属性より、ローカルコピーが置かれているタプル空間が分かるので、それらのタプル空間に対して basic:in でそのコピーを削除する。

図 7 に (out “foo” :neighborhood (“TS1” “TS2” “TS3”)) が実行された後で、(in “foo”) をタプル空間 TS2 上で実行したときのタプル演算を示す。

rd 演算の実行方法は in 演算の実行方法から明らかであるので割愛する。

6. メタインタプリタ

インタプリタは、タプル演算を実行するだけでなく、(1) eval において、与えられた計算をプロセッサに割り当てる eval サーバとして機能、(2) デバッグのためのユーティリティの提供、(3) タプルの管理、などの役割を担う。表 3 にインタプリタが提供するアクティブ TS 機能を示す。

アクティブ TS 機能はメタインタプリタによって提

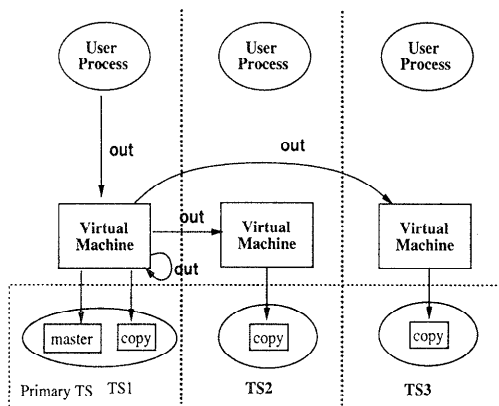


図 6 クラスタ化タプル空間での out 演算
Fig. 6 Out operation in clustered tuple space.

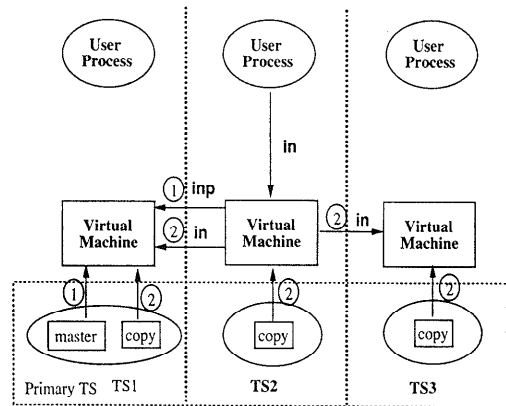


図 7 クラスタ化タプル空間での in 演算
Fig. 7 In operation in clustered tuple space.

表 3 アクティブ TS 機能
Table 3 Active tuple space primitives.

Primitive	Operation
eval-srv	eval サーバを指定
trap	演算の実行前に trap を設定
restart	中断されて演算を再開
log	実行のモニタ
freeze	タプルへのアクセスを中断
cancel	演算の実行を破棄
step	演算を 1 ステップずつ実行
expire	寿命が切れたタプルを回収

供されている。すなわち、メタインタプリタは、インタプリタの使用するデータや定義を変更することによってデバッグ機能やユーティリティを実現している。これらの機能の実現にあたってメタインタプリタは 2 つのプリミティブ、trap, multiple-wait を使用している²⁾。たとえば、タプルへのアクセスを中断する freeze と演算を 1 ステップずつ実行する step (表 3) は、trap を使用して次のように定義される。

```
(de freeze ()
  ; trap 条件
  (trap '(in inp rd rdp out eval)
    :before
    (process-interrupt) ))
(de step ()
  (trap '(in inp rd rdp out eval)
    (clear-before-trap)
    :after
    (freeze) ))
```

trap は、trap 条件を指示する第 1 引数に含まれるどれかの演算が起動されたら、:before か :after に従って、その前後で指定された処理を行う関数である。trap の第 1 引数で指定された演算が起動されるのを待つのは、multiple-wait で実現される。multiple-wait は、指定された演算 (これ自体タプル) に対して in を行おうとするプロセスを生成し、それらをブロックされたものとして登録する。次に、どれかの演算が起動されたら、対応するプロセスを起動し、残りの待ちプロセスはすべて破棄する。

インタプリタのすべてのプリミティブの定義は、タプルで格納されている。核の部分は、basic:in および trap, mutiple-wait 等の少数のプリミティブだけで記述され、残りの演算はすべてその上に作られている。したがって、メタインタプリタは、インタプリタが使用するデータを書き換えるだけでなく、プリミテ

ィブの定義であるタプルを書き換えることによって、インタプリタの行動を変更することも可能である。

7. 関 連 研 究

関連研究との比較を行う。

●並列協調処理モデル Cellula^{16), 17)}:

プロセスと場を一体化したものをセルと呼び、プロセス間の通信情報単位を Linda モデルのタプルで表現している。これは、場をタプル空間とみなすと、多重タプル空間が実現されていることになる。ただし、通信用のセルを介して通信が行われ、タプルに付随した属性が様々な通信形態を規定する¹⁶⁾。

実際の実験によって、通信用セルを保持するプロセッサへのアクセス競合が生ずることが判明し、最新版では「一時キャッシュ法」によって、1対1通信のオーバーヘッドを回避している¹⁷⁾。一時キャッシュ法とは、本稿で in/out したプロセッサにローカルコピーを持たせることによって、primary へのアクセスを少なくせよという方法である。このようなクラスタ化では、マスタタプルは不要となるので、演算の最適化が可能である。一時キャッシュ法はそのような最適化の方法の 1 つとして使用できる。逆に、Cellula では検討されていない 1 対多通信の最適化にクラスタを適用すれば、通信のオーバーヘッドは減らすことができると考えられる。

なお、処理系は、Common Lisp と TCP/IP のソケットを利用しているので、高級言語で記述されているので移植性はあるが、自己記述は考慮されていない。

●並列オブジェクト指向言語 TupleSpaceSmalltalk⁷⁾:

協調問題解決のためには、受け手が静的には決まらないので、メッセージ伝達による通信では、柔軟性がなく、不十分である。TupleSpaceSmalltalk ではメッセージ伝達ではなく、タプル空間を用いて通信を行う。また、タプルとタプル空間もオブジェクトとして扱われ、Concurrent Smalltalk で作成されている。高級言語で記述されているので移植性はあるが、自己記述はされていない。

●信頼性向上のために Linda に多重タプル空間を導入^{12), 18)}:

Xu はタプル空間で複数のコピーを持つことによって信頼性向上を狙っている。Linda カーネルは、処理プロトコルと view 変更アルゴリズムとから構成される。out ではすべてのタプル空間にコピーを作

成し, in ではどれか1つのタプル空間からマッチするタプルを取る。処理プロトコルは, 様々な最適化を行い, 処理遅延を減らすように工夫されている。たとえば, 連続する out はそれらのタプルを1まとめにして, すべてのタプル空間へ書き出したり, 同期が不要な部分の処理は同時的なバックグラウンドで行っている。もし, 障害が生じると, タプル空間がクラスタ化 (view と呼ぶ) されて, 障害の回復を図る。しかし, システムカーネルは, C言語で書かれており, 移植性や拡張性は考慮していない。

並列オブジェクト指向言語 LGO では, プロセッサの故障に対する故障回復機構を多重タプル空間を用いて実現している¹²⁾。プロセスの内部状態を複数のタプル空間に保持する。しかし, LGO でも, システムカーネルの移植性や拡張性には考慮していない。

● Linda 3⁶⁾:

従来の Linda が, 通信, 同期, プロセス生成の区別をなくすために設計されたように, Linda 3 はプログラム実行とファイルとの区別をなくすために多重タプル空間を導入している。新しく追加された演算は tsc (tuple-space-create) であり, タプルの属性として, ts (tuple-space) が追加されている。

多重タプル空間は木状に階層化され, たとえば, Unix 風のパスで名前が付けられる。自分の属する場所以外にタプルを書き出す場合には, name.out と名前 name を指定する。しかし, エージェントは自分の属する場所以外のタプルは直接には読めない。これについては, 適当なプロトコルを用いて, 相手から送ってもらう必要がある。場所間の通信にはメタタプル空間が使用される。

プロセスの実行とファイルとを同一視する方向は興味深い, あくまで, 使用するハードウェア, あるいは, オペレーティングシステム, 言語の実行時ルーティンが同じであるという仮定が必要である。すなわち, Linda 3 のような高次の抽象化は, 均質システムが前提であり, オリジナル Linda の非均質性が犠牲にされている。クラスタは考慮されていないが, 非均質システムの中で均質なものをクラスタ化することによって, Linda 3 の高次抽象化が生きてこよう。

なお, Linda 3 の処理系の自己記述は検討されていない。

●階層的な多重タプル空間 PoliSpaces⁵⁾:

多重タプル空間は木状に階層化されており, 個々のタプル空間が「場所」に, 複数の場所から「町」が構成される。場所は, 黑板モデルでの通信を中継する黑板として使用される。Linda 3 に従った考え方であり, ソフトウェアツールの設計に応用されている。これは, Linda 3 の実行中のプロセスとファイルの同一視による自然な応用である。Linda 3 の実装にはメタタプル空間を使用しているので, 種々の通信形態, あるいは, クラスタ化等が可能であろうが, 検討はされていない。

8. 考 察

本稿では, NueLinda のクラスタ化タプル空間のための処理インタプリタの移植性を高めるためにインタプリタを直接 NueLinda 自身で記述するアプローチについて報告した。インタプリタは高レベルでの記述でありがちな非効率を排除するために, 同期フェーズと非同期フェーズとに分けることによって, 並列処理の可能性を増すようにした。また, クラスタへのローカルコピーの作成と削除は非同期処理で行われるので, そのクラスタに属するどれかのタプル空間でローカルコピーが作成される前に, そのローカルコピーの削除が要求される場合がありうる。このような処理順序の逆転があっても, 本節で提案したインタプリタの処理方式では正しく処理が行われることが保障される。

インタプリタの自己記述は, 少数の基本関数しか使用していないので, 実装は容易に行えると期待している。とくに, 既存の Linda 処理系がある場合には, 既存処理系の通信部分およびタプル管理部分を再利用することが出来るので, クラスタ化し, NueLinda 処理系へ拡張することは容易であろう。TAO/ELIS システムでの NueLinda 処理系の開発は, 通常の Linda モデルをまず作成し, その後で提案した自己記述で, NueLinda モデルへと拡張する, という手法を採った。本稿で提案した処理モデルは, TAO/ELIS システムだけでなく, 他のシステムへも容易に適用できる。

今後の課題としては, 以下のようなものがある。

- (1) NueLinda インタプリタの実装: 現在, TAO/ELIS 上で実装している。稿を改めて, 実装法およびその性能評価を報告する。

また, UNIX 上での実装あるいは様々な言語に NueLinda を埋め込むことも検討する予定である。

- (2) クラスタを様々な応用に適用し、その有効性を検証する。
- (3) in, rd でのクラスタの指定: クラスタからマッチするタプルを取るには、幾つかのセマンティクスが考えられる。
- (a) どれか1つのタプル空間から1つだけマッチするタプルを読む。
(タプル空間へのアクセス順序に指定がある場合とない場合がある)
- (b) マッチするタプルがあるタプル空間からは1つずつマッチするタプルを読む。
- (c) すべての指定されたタプル空間から1つずつマッチするタプルを読む。

第1番目の拡張において、複数のタプル空間に対して同時に in を実行するという「先行実行」を行った場合には、(1)先行実行に対するリワインド機能が必要になること、および、(2)先行実行で発生した不要なプロセス待ちに対してキャンセルが必要となること、という2つの問題が生ずる。すなわち、指定されたすべてのタプル空間に対して、第5.2節で述べた in を同時に先行実行すると、マッチするタプルを取りすぎる場合があり、その時に不要なタプルを元あったタプル空間に戻す必要がある。また、マッチするタプルがないタプル空間にはそのタプルを待つプロセスが登録されるので、キャンセルしなければならない。現在、この戻し機構として検討しているのは、マスタタプルの位置を示す forwarder を使用する方法である。第5.2節で説明したように、マッチするタプルがあると、マスタタプルが持ってこられる。そこで、マスタタプルを primary タプル空間の所へ戻さず、持ってきたタプル空間に basic: out する。そして、元の primary タプル空間には、マスタタプルの新しいタプル空間を示す forwarder を置く。

不要な待ちプロセスのキャンセルは、容易である。待ちプロセスはキュータプルとして表現されているので、in で消去すればよい。

後者2つの拡張については、メタインタプリタを用いて簡単に実装できる。

9. おわりに

本稿では、非均質な分散システムでのカーネルの移植性を高める手法として、カーネルをそれ自身で記述

する自己記述という手法を提案した。具体的には、Linda をクラスタ化タプル空間に拡張した NueLinda のインタプリタを直接 NueLinda 自身で記述することによって、処理系が容易に作成できることを報告した。

謝辞 最後に、日頃ご指導をいただく NTT 基礎研究所情報科学部竹内郁雄リーダー、ご討論いただいた竹内グループの同僚、分散研究グループの同僚に感謝いたします。論文の構成等に貴重なコメントをいただいた査読者の方に感謝いたします。また、食事をする哲学者のプログラムの実行表示に Xsight のプログラムを使わせていただいた英国 University of Bath の Julian Padget 準教授に感謝いたします。

参 考 文 献

- 1) 明石 修, 佐島隆弘, 村上健一郎: Nue-Linda の実現, 日本ソフトウェア科学会第7回大会論文集, B5-2 (1990).
- 2) 明石 修: NueLinda における TS の制御とクラスタ化, 日本ソフトウェア科学会第8回大会論文集, F2-3 (1991).
- 3) Carriero, N. and Gelernter, D.: Linda in Context, *Comm. ACM*, Vol. 32, No. 10, pp. 444-458 (1989).
- 4) Carriero, N. and Gelernter, D.: *How to Write Parallel Programs. A First Course*, MIT Press (1990).
- 5) Ciancarini, P. and Gelernter, D.: A Distributed Programming Environment Based on Logic Tuple Spaces, *Proc. of Int. Conf. on Fifth Generation Computer Systems (FGCS-92)*, pp. 926-932, ICOT (1992).
- 6) Gelernter, D.: Multiple Tuple Spaces in Linda, *Proc. of PARLE '89*, Vol. 2, Lecture Notes in Computer Science, No. 366, pp. 20-27 (1989).
- 7) Matsuoka, S. and Kawai, S.: Using Tuple Space Communication in Distributed Object-Oriented Languages, *Proc. of OOPSLA '88*, ACM, pp. 276-284 (1988).
- 8) 村上健一郎, 明石 修, 天海良治, 奥乃 博: 計算モデル Linda の TAO への導入, 情報処理学会記号処理研究会資料, 57-4 (1990).
- 9) Murakami, K., Amagai, Y., Akashi, O. and Okuno, H.G.: TOPS: A Tuple Operation Protocol Suite for Nue-Linda Computation Model, *Proc. of 6th Joint Workshop on Computer Communications (JWCC-6)*, pp. 39-46, IPSJ (July 1991).
- 10) Nii, P.H.: Blackboard Systems, Part 1: The Blackboard Model of Problem Solving and Evolution of Blackboard Architectures, *AI*

Magazine, Vol. 7, No. 2, pp. 38-53 (1986).

- 11) Nii, P.H.: Blackboard Systems, Part 2: Blackboard Application Systems, Blackboard Systems from a Knowledge Engineering Perspective, *AI Magazine*, Vol. 7, No. 3, pp. 82-106 (1986).
- 12) 野里貴仁, 杉本 明, 阿部 茂: 並列オブジェクト指向言語 LGO の故障回復機構, WOOC '92, 日本ソフトウェア科学会 (1992).
- 13) 奥乃 博: 非均質システム NueLinda 上での ATMS の並列処理の検討, 「並列処理シンポジウム JSPP '91」報告集, pp. 377-384 (1991).
- 14) 奥乃 博, 明石 修, 村上健一郎, 天海良治: NueLinda Interpreter in NueLinda—非均質システム NueLinda インタプリタの自己記述—, 「並列処理シンポジウム JSPP '92」報告集, pp. 155-162 (1992).
- 15) 竹内郁雄, 後藤滋樹, 尾内理紀夫, 斎藤康己, 奥乃 博: コネクティクス構想, 日本ソフトウェア科学会第8回大会論文集, B3-1 (1991).
- 16) 吉田紀彦, 檜崎修二: 場と一体化したプロセスの概念に基づく並列協調処理モデル Cellula, 情報処理学会論文誌, Vol. 31, No. 7, pp. 1071-1079 (1990).
- 17) 吉田紀彦, 檜崎修二: 協調処理モデル Cellula の分散処理系, 情報処理学会論文誌, Vol. 32, No. 7, pp. 906-913 (1991).
- 18) Xu, A. S.: A Fault-Tolerant Network Kernel for Linda, MIT/LCS/TR-424 (Aug. 1988).

A 食事をする哲学者問題

```
(use-package "linda")
(defvar num-of-philos 5)           ;; 哲学者の数
(defconstant amount-of-spaghetti 20) ;; スパゲッティの量

(de do-philos (&aux info)
  (out "spaghetti" amount-of-spaghetti)
  (for i (index 1 num-of-philos)
    ; 哲学者のプロセスを作る
    (eva "philos"
      :load-files '("demo:philos-2.tao")
      (philos) )
    (out "chopstick" i)           ; 箸を置く
    (out "philosopher" i)         ; 哲学者の id 作成
    ; 部屋に入れる数を制限するチケット
    (if (< i num-of-philos)
      (out "room ticket" i))
    ; wait processes (eva の出した tuple)
    (for i (index 1 num-of-philos)
      (in "philos" ? info)         ; eva の結果: info = (ID eat-num)
      (format t "philos[~D] num=~D amount=~D ~%"
        (car info) (cadr info) (caddr info))
      (for i (index 1 num-of-philos) ; 終了時の掃除
        (if (< i num-of-philos)
          (in "room ticket" i)      ; チケットを捨てる
          (in "chopstick" i)        ; 箸を捨てる
          (in "spaghetti" i)        ; _ は wildcard
          ; _ は wildcard
          ))
      ))
    (de philos (&aux id left right) ;; 哲学者
      (in "philosopher" ? id)       ; id を取る
      (!left id)                    ; 右手
      (!right (1+ (mod id num-of-philos))) ; 左手
      (loop
        (&aux eat-num eat-amount try)
        (:init (!eat-num 0) (!eat-amount 0))
        (think id)
        (inc eat-num)
        (:while (> (!try (eat id eat-num left right)) 0)
          (list id eat-num eat-amount) ) ; 統計情報を返す
          (inc eat-amount try) ))
      ))
    (de eat (id num left right &aux amount try)
      (in "room ticket" i)          ; チケットを得る
      (in "chopstick" left)         ; 箸を取る (左手)
      (in "chopstick" right)        ; 箸を取る (右手)
      (in "spaghetti" ? amount)     ; スパゲッティを取る
      (!try (1+ (random 5)))         ; (排他制御)
      (if (> try amount)
        (!try amount) )
      (out "spaghetti" (- amount try))
      (format t "--- philos[~D] EATING -> ~D~%" id num)
      (sleep (1+ (random 5)))
      (out "chopstick" left)        ; 箸を置く (左手)
      (out "chopstick" right)       ; 箸を置く (右手)
      (out "room ticket" i)         ; チケットを返す
      try )
    )
  )

(de think (person)
  (sleep 0.5) )
```

B クラスタを用いた食事をする哲学者問題

```

(use-package "linda")
(defconstant num-of-philos 5)           ;; 哲学者の数
(defconstant amount-of-spaghetti 100)   ;; スパゲッティの量

;; ts-list は2人の哲学者が共有するタブル空間のリストを保持.
;; (ts-list '(TS1 ... TS5))

(de do-philos (&aux info)
  (neighborhood ts-list)                ;; default 近傍の設定
  (out "spaghetti" amount-of-spaghetti)
  (for i (index 1 num-of-philos)
    (eva "philos"                      ;; philosopher プロセス作成
      :load '(("demo:philos-1.tao")
        (philos) )
      (out "chopstick"                  ; 箸を個々の TS に置く
        :neighbor (list (nth (1- i) ts-list)
          (nth (mod i num-of-philos) ts-list) ))
      (out "philosopher" i)             ; 哲学者 id 作成
      (if (< i num-of-philos)
        (out "room ticket") ))         ; チケット作成
    (for i (index 1 num-of-philos)
      (in "philos" ? info)             ;; eva の結果を得る, info = (ID eat-num)
      (format t "philos[\"D\"] num= \"D\" amount = \"D\" \"%\"")
      (car info) (cadr info) (caddr info) )
    (for i (index 1 num-of-philos)      ; 終了時の掃除
      (if (< i num-of-philos)
        (in "room ticket") )           ; チケットを捨てる
        (in "chopstick") )             ; 箸を捨てる
      (in "spaghetti" _)               ; スパゲッティの皿を捨てる
    )
  )

;; execute as forked process
(de philo (&aux id my-ts left right)
  (neighborhood ts-list)                ;; default の近傍を設定
  (in "philosopher" ? id)              ;; ID を得る
  (!my-ts (list (nth (1- id) ts-list)))
  (!left (list (nth (if (= id 1) (1- num-of-philos) (- id 2)) ts-list)
    (car my-ts) ))
  (!right (list (car my-ts)
    (nth (mod id num-of-philos) ts-list) ))
  (loop
    (&aux eat-num eat-amount try)
    (:init (!eat-num 0) (!eat-amount 0))
    (think)
    (inc eat-num)
    (:while (> (!try (eat id eat-num my-ts left right))) 0)
    (list id eat-num eat-amount) )      ;; 統計情報を返す
    (inc eat-amount try) ) )

(de eat (id num my-ts left right &aux amount try)
  (in "room ticket")                   ; 部屋に入る
  (in "chopstick" :neighbor my-ts)      ; 箸を取る (右 / 左手)
  (in "chopstick" :neighbor my-ts)      ; 箸を取る (残った手)
  (in "spaghetti" ? amount) ; スパゲッティを取る
  (!try (1+ (random 5)))
  (if (> try amount)
    (!try amount) )
  (out "spaghetti" (< amount try))
  (format t "---- philos[\"D\"] EATING -> \"D\" \"%\" person num)
  (sleep (1+ (random 5)))
  (out "chopstick" :neighbor left)      ; 箸を置く (左手)
  (out "chopstick" :neighbor right)     ; 箸を置く (右手)
  (out "room ticket") )                ; チケットを返す

(de think ()
  (sleep (1+ (random 5))) )

```

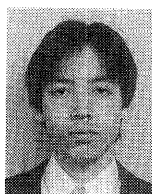
(平成4年9月21日受付)

(平成5年3月11日採録)



奥乃 博 (正会員)

1950 年生. 1972 年東京大学教養学部基礎科学科卒業. 同年日本電信電話公社 (現 NTT) 入社. 1986~1988 年スタンフォード大学コンピュータ科学知識システム研究所客員研究員. 1992 年より東京大学工学部電子工学科知能工学 (富士通) 寄附講座に客員助教授として出向中. AI の並列処理, 創発的計算モデルの研究に従事. 1990 年度人工知能学会論文賞受賞. 人工知能学会, 日本認知科学会, 日本ソフトウェア科学会, ACM, AAAI 各会員. 人工知能学会並列人工知能研究会幹事. 「知的プログラミング」(共著, オーム社, 新刊)



明石 修 (正会員)

1964 年生. 1987 年東京工業大学理学部情報科学科卒業. 1989 年同大学大学院修士課程修了. 同年日本電信電話株式会社入社. 以来, 主に分散システムの研究に従事. 現在, 基礎研究所情報科学部勤務. ACM, 日本ソフトウェア科学会各会員.



村上健一郎 (正会員)

1955 年生. 1979 年九州大学工学部情報工学科卒業. 1981 年同大学大学院修士課程修了. 同年日本電信電話公社 (現 NTT) 入社. オペレーティングシステム, コンピュータネットワークの研究に従事. 現在, NTT 基礎研究所主任研究員. 1991 年第 6 回元岡賞受賞. 電子情報通信学会, 日本ソフトウェア科学会, ACM, Internet Society 各会員. 「はやわかり TCP/IP」(共訳, 共立出版).



天海 良治 (正会員)

1959 年生. 1983 年電気通信大学電気通信学部計算機科学科卒業. 1985 年同大学大学院修士課程修了. 同年日本電信電話公社 (現 NTT) 入社. 以来, プログラミングパラダイム, 計算機アーキテクチャの研究に従事. 現在, NTT 基礎研究所主任研究員. 日本ソフトウェア科学会会員. 「GNU Emacs マニュアル」(共訳, 共立出版)