人　工　知　能 95−10
（1994．7．21）

有機的プログラミング言語 Gaea におけるエージェント間通信

中島秀之
電子技術総合研究所

概要

我々は“新ソフトウェア構造化モデル”プロジェクトにおいて，柔軟に協調的動作を行なうプログラムを構築する方法論を研究している．本論文ではこれらを満たすプログラミング像として，有機的プログラミングの考え方を提案し，その上のマルチ・エージェントシステムの実現に関して述べる．

Agent Communication in an Organic Programming Lanugage Gaea

Hideyuki Nakashima
Electrotechnical Laboratory

Abstract

We are developing a new software methodology for building large, complicated systems out of simple units. The emphasis is on the architecture (called cooperative architecture) which is used to combine the units, rather than on the intelligence of individual units.

We named the methodology “organic programming” after the flexibility of organic systems such as plants and animals. This paper describes how to implement “agents” and communication among agents on Gaea.

1 はじめに

> 【有機的】有機体のように，多くの部分が集まって一個の物を作り，その各部分の間に緊密な統一があって，部分と全体とが必然的関係を有しているさま．（広辞苑）

我々は“新ソフトウェア構造化モデル”プロジェクトにおいて，柔軟に協調的動作を行なうプログラムを構築する方法論を研究している．特に，

1. 柔軟性：動作中に環境や仕様の変更に追従すること
2. 協調性：集合体としては部品の和以上の仕事をすること

の二点を目標に，このような性質を持ったプログラムを構成するための手法を模索している．本論文ではこれらを満たすプログラミング像として，有機的プログラミングの考え方を提案し，その上のマルチ・エージェントシステムの実現に関して述べる．

有機的プログラミングは有機的システムの以下のような特性をプログラムに持ち込むことを考えて設計されている．

- ダイナミズム：構造が動的に変化すること．
- 相互作用：部分どうし，あるいは部分と全体との相互作用．（従来型のプログラミング言語では代入やサブルーチン呼び出しのような片方向作用は多く見られるが，相互作用はない．ユニフィケーションですらミクロには片方向の代入である．）

これらを満たすようなシステムは多数考えられるが，我々はその一つとして具体的には以下のような性質を持つシステムを構成するためのプログラミング方法論を提案する．

- マルチ・プロセス
- プロセス間の環境共有
- 記述の環境依存性
- 動作の環境依存性
- 環境の動的編成

以下ではまず我々が協調アーキテクチャ研究で目指しているものを述べ，その実現手段としての有機的プログラミングの概念を中心に述べた後，それに基づく言語 Gaea の仕様と，その上のマルチ・エージェントシステムの実現に関して述べる．

2 協調アーキテクチャの意味

まず，複雑なシステムとはどういうシステムかについて考えよう．そのためにスケールの厚み [1] という概念を導入する．スケールとはシステムを解釈・記述する際に用いる物指しの大きさのことである．例えば長さの物指しでいうと，キロメートルあるいはそれ以上の物指しで見ても人間は見えない．メートルというスケールで丁度人間が見える．更にセンチメートルのスケールでは人間全体は見えず，各器官が見えてくる．ミリメートルからミクロンのスケールでは細胞の構造が見え，オングストロームスケールになると分子あるいは原子構造が見えてくる．

又，時間の物差しで言うと，人間には秒単位の動きは見えるが時間あるいは日単位以上（例えば植物の成長），ミリ秒以下（例えば蜂の羽ばたき）の動きは見えない．このように，複雑なシステムを記述するにはそのスケールが重要である．これを間違うと見えるものが見えて来ない．

また，自然科学の分類でいえば，量子力学が対象とする現象のスケール，化学が対象とする現象のスケール，生物学が対象とする現象のスケールはそれぞれ異なっている．

更にスケールの厚み (scale thickness) という概念を導入する．これはどれ位のスケールの幅でそのシステムが (各々異なる) 意味を持っているかというものである．例えば三角形という概念はその大きさ程度のスケールの幅 (非常に狭い) でしか意味を持たない．一般に身のまわりの人工物はこういう性質を持つ．例えば紙クリップは数センチのスケールで見た形状だけが本質で，その材質が見えるレベルの

スケール（オングストローム）では意味を持たない．（図1）

これに対し集団という概念は集団としてのそれとその構成要素のそれとの少なくとも二つの異なるスケールにおいて意味を持つ．人間は多くのスケールで意味を持っている（図2）．フラクタルはこの厚みが（スケールの小さい方向に）半無限で，しかも，各スケールが相似である．このようにシステムの複雑さをスケールの厚みで計ることができる．

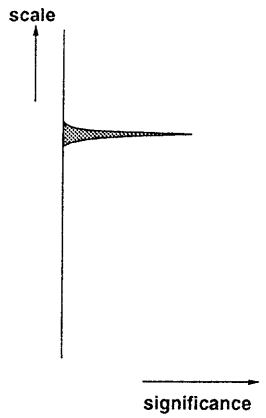

図 1: スケールに関して薄いシステム

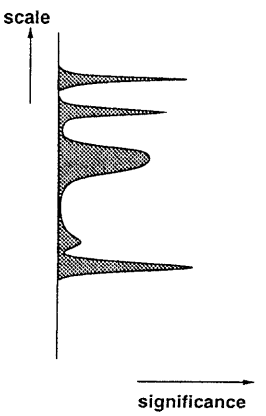

図 2: スケールに関して厚いシステム

さて，従来のプログラミング技法は，極論すると“分割統治”というやり方であった．これは複雑な問題をより単純な部分に分割しそれぞれを単独に解くというもので，部分問題がまだ手に負えないときは再分割を行なう．これを適当にくりかえすことにより，最終的には単純な問題に分割でき，それらを解けば最初の問題が解けたことになる．

分割統治方式は大量のデータが比較的単純な規則だけで結合されている場合には有効である．例えば大量の数値をその大きさの順に並べ直すソートのプログラムなどはこの手法で書かれ成功している例である．しかし，このような考え方はスケールに関して厚いシステムには使えない．分割によってどんどん小さな単位になるのだが，分割後の相対的に小さなスケールだけでは定義により問題がカバーしきれないからである．小さなスケールでの記述と大きなスケールでの記述が両方同時に存在する必要がある．

例えば蟻の社会を考えてみよう．少なくとも蟻全体と個々の蟻という二つのスケールが存在する．個々の蟻は，知的システムとして見た場合には比較的単純な存在である．しかし，蟻の社会は非常に複雑であることが知られている．そしてこの社会の性質は個々の蟻には還元できない．

協調アーキテクチャに求められているのは以下のような機能である．

- 対象システムを様々なスケールで記述できること
- スケール間の関係が記述できること

有機的プログラミングにおいては状況依存推論 [6] の考え方に基づいた，context reflection[5] の機能によりこれを実現している．

3 有機的プログラミングの概要

プログラムはモジュールより構成される．このモジュールをセルと呼ぶ．各セルにはプログラム片が格納され，セルを組み合わせることにより完全なプログラムとなる．このプログラムは走っているプロセスから実行時に参照される．つまり，実行前にプログラムが定

まっているわけではなく，実行時に変化するセルの組合せによってプログラムが決まる．

有機プログラミングにおける環境には二通りのものがある：

1. プロセスにとっての環境．

   マルチ・プロセスを前提としているが，あるプロセスから見た環境とはプロセスが参照する変数，プログラム，他のプロセスなどである．この意味で，セルの構造をプロセスにとっての環境と呼ぶ．

2. セルにとっての環境．

   セルにはプログラムの断片が格納されているが，そのプログラムが他のプログラムを参照している場合には，それが他のセルに存在する．

プロセスにとっての環境はセルの構造体として表現される．この構造はプログラム可能なものなら何でも良いが，最も単純なものとしてはスタックが考えられ，Gaea ではそれを採用している．つまり，プロセスが参照するデータはセルのスタックとして格納されており，必要に応じてスタックを探索すれば良い．例えばあるプログラム p の呼びだしが起こったときに，p の定義はスタック内のどれかのセルに格納されているはず（でなければエラー）だから，それを探して使うのである．この考え方はクラスからプログラム（メソッド）を継承するものに近いが，環境が動的に変化する点が異なっている．

あるセルに存在するプログラム p が別のプログラム q を参照している場合，これがどこに格納されているかは実行時の環境でしか決まらない（図 3）．

また，新しいセルをプッシュする場合，そのセルは名前で参照されるが，その名前はセルによって異なる．つまり，同じセルが別のセルからは異なる名前で参照されたり，同じ名前がセルによって異なるセルを参照していたりする[1]．これは一見混乱の元のような気もするが，人間同士が言葉による情報伝達を行なっている場合にもしばしば観察されることである．例えば“父”と“夫”という異なる名前で（娘と母が）同一の個体を参照したり，関西の“たぬき”は関東の“たぬき”とは別物で，関東では“きつねそば”であったりする．

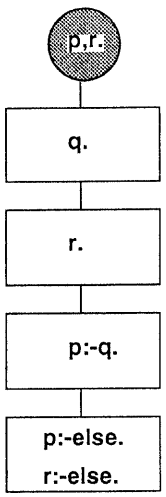

図 3: セルのスタック

このようなプログラムの実行時参照を通して“名前の柔軟な制御”が可能になる．ここでの名前とはプログラムとセルの両者の名前である．

環境をセルのスタックに限定した場合でも，マルチ・プロセスの間で外側の環境をシェアすることが可能であるから，一般的には環境は木構造を構成する（図 4）．

3.1 ダイナミズム

生命は動的平衡状態を保つことによって成立している．有機的プログラミングにおいても，プログラムの動的変化を重要視する．従って，従来のコンパイルの概念は否定される．[2]

具体的には環境（セルの構造）がプログラ

[1] この局所的名前付けの機能は後述の例では使われていない．

[2] コンパイルや部分計算が起こってもよいが，それらは永続的なものではなく，計算の必要性に応じて繰り返される必要がある．例えばコンパイルは環境の動的再編に応じて行なわれる必要があるが，再編は頻繁に起こるので効率を考えるとコンパイルしないか，あるいは漸時的に更新できるコードである必要がある．

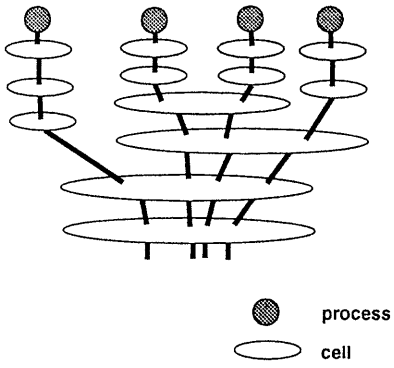

図 4: マルチ・プロセス間の環境

ムの操作対象となる．プログラム実行中にデータ構造を操作するように環境を操作する．これにより環境の動的変更が可能となる．

Object oriented プログラミング（OOP）では，クラス階層を設計時に固定してしまうので後の仕様変更に追従した変更が困難であった．特に上位クラスが勝手にオブジェクトを生成する場合，その生成されたオブジェクトのクラスは最初に固定されてしまうので，後の変更を取り込めない．

3.2 相互作用

各プロセスは環境を操作することができ，同時に新しい環境に影響される．これによりプロセスと環境の相互作用が実現される．プログラムは環境から継承するので環境を変えるとプログラムも影響を受けるからである．

セルにプログラムが格納され，セルの構造が変化することによってプロセスが参照するプログラムが変化するのであるから，これはある意味では自己反映計算の概念を押し進めたものである．自己反映計算においてはプログラムとそのモデルが存在するが，ここではプログラム自身が同時にモデルである．また，オブジェクトレベルとメタレベルの区別も存在していない．

3.3 エージェントの記述

有機的プログラミングにおいては“エージェント”という考え方を推進するわけではない．エージェントはプログラムされるべき存在で，アプリケーションに属する概念である．この際，エージェントをプロセスとその環境を統合したもの（図 5）と考えるのが自然である．ある特定のプロセスから見た場合，一番外側の環境は，プログラムが動作している実世界とのインターフェースである．実世界に存在する他のエージェントは別のプロセスに見える．概念的にはこれらのエージェントも同じ構造を持っているが，その内部は見えないので，実際には全く別の構造を持っているかも知れないがそれは問う必要がない．

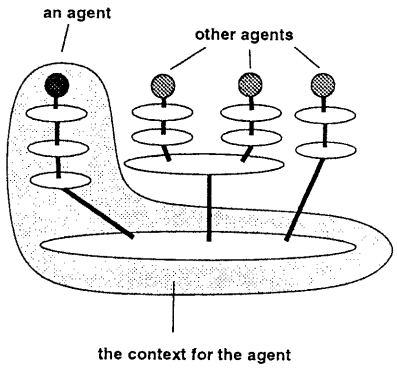

図 5: エージェントと外界

各プロセスの一番内側の環境は，プログラムの局所変数など．その意味で，環境とはプロセスをとりまくすべてのものであり，階層的に実世界へとつながっていると考える．例えばロボットなどを想定すると，どこかにセンサーやアクチュエータの層があり，それが外側の環境と内側の表現を結合している．

有機的プログラミングにおいては，これらの環境をすべて統一的に扱う．すなわち，セルと呼ばれる単位として環境の要素を表現し，それらを階層的に組み合わせることにより，上記の環境の層を実現する．

3.4 エージェント間通信

Gaeaにおいては通信は環境の共通部分にあるセルへの書き込みと，その読みだしによって行なう．従って，他のエージェントに強制的にメッセージを送る方法はなく，各エージェントが自律的にメッセージを読み出すことになる．この手法は久野の提案したサブジェクト間の通信 [2] に近い．

4 例

石川による交差点問題 [3] を例にとる．この問題は二方向から来たロボット2台が交差点で衝突せずに横断するための通信を並列オブジェクトオリエンテド言語で記述する際の問題点を示している．ここでの前提は以下の通りである：

1. negotiationするためには双方が共有する状態が必要となる．
2. 多くのプログラミング言語では言語上あるいはruntime時に共有する状態を実現するために，共有領域を仮定している．
3. この共有領域を仮定しないプログラミング言語の枠組＆最適実現方法があるか？

このような，共有領域のロックに頼らない通信では，各オブジェクトが内部で状態の整合性を保つ必要がある．すなわち，観測から行動までを原子行為とする必要がある（そうでないと，観測と行動の間に状態の変化が起こった場合に整合性のない行動となってしまう可能性がある）．また，negotiationにおいては通信の往復が必要であるが，その際に相手の返事が到着するまで行動を停止していたのではデッドロックに陥る可能性がある．そのため，返事が来なくてもなんらかの行動を続けるようにプログラムするが，その場合は自分が返事待ちであるという状態を維持する必要がある．

石川はこのような，内部状態を維持した状態でメッセージ交換を複数回起こすような並列オブジェクトの記述を従来方式で行なうと，サブクラスへの継承が不可能になることを指摘した．この解決として動的にメソッドスコープを変更する手法を提案している．

ここではセルの動的再編成により同様の機能が実現できることを示す．ただし，Gaeaにおいては原子行為を規定する機能がないので，動作の完全性は保証されない．すなわち，全く同じタイミングで2台のロボットが交差点に進入した場合は衝突の可能性がある．しかし，現実問題としてはタイミングのずれにより衝突は回避されるし，全く同じタイミングであったとしても衝突しないようにより詳細なプログラムをすることも可能である．いずれにしても衝突回避は本論文の主題ではないので例の簡略性を保つために通信とその記述に焦点を絞った．

また，Gaeaにおける通信は前述のように共有領域を用いているが，そこでのロック機構がないので，共有領域であることを本質的には利用していない．

プログラムは以下のように書ける．[3]

```
; test は初期化とプロセス分岐を行なう
(as (test)
    (new-cell *c) (name *c com)
    (push c)
    (new-cell *a) (name *a a)
    (fork (push a)
          (initialize a 12)
          (actloop)))
    (new-cell *b) (name *b b)
    (fork (push b)
          (initialize b 8)
          (actloop))))

; 各ロボット (*name) の初期状態を proceed に設定
; 位置（交差点までの距離）を *loc に設定
(as (initialize *name *loc)
    (set name *name)
    (push proceed)
    (set loc *loc))
```

[3] 現時点でGaeaはフル実装されていないので，実験に用いたのは並列版Uranus[4]である．本テーマの範囲では本質的な差はない．なお，プリミティブなどはGaeaに合うように変更してある．

```
; 状態遷移
(as (go *state) (pop) (push *state))

; 各プロセスはactloopのループ.
; 各状態でeを起動する.
; eの定義の動的変更により環境に追従.
(as (actloop)
    (act)
    (actloop))

; proceed状態の定義
(and (new-cell *p) (name *c proceed))
(push proceed)
  (as (act) (loc  0) (go want))
  (as (act) (decl loc) (loc *n))
(pop)

; want状態の定義
(and (new-cell *w) (name *w want))
(push want)
  (as (act) (find enter) (go stop))
  (as (act) (enter) (go enter))
(pop)

(as (enter) (message enter)
      ; 状態でなく各エージェントの
      ; ところに位置設定
    (name *n) (push *n)
    (set iloc 5)
    (pop))

; enter状態の定義
(and (new-cell *e) (name *e enter))
(push enter)
  ; 0になったら交差点から出る
  (as (act) (iloc 0)
          (message cleared)
          (go proceed))
  ; 進む
  (as (act) (decl iloc))
(pop)

; stop状態の定義
(and (new-cell *s) (name *s stop))
(push stop)
  (as (act) (find cleared)
          (go proceed))
  (as (act))  ; 何もしない
(pop)

; message送信
; comが通信チャネル
(as (message *m) (name *name)
    (push com)
    (as (message- *name *m))
    (pop))

; message受信
; comが通信チャネル
(as (find *m)
    (push com)
    (message- *n *m)
    (not (name *n))
       ; 自分の発信したメッセージ
       ; でないことを確認
    (retract (message- *n *m))
    (pop))

; 距離を減らす操作
(as (decl *c) (name *nm)
    (push *nm (*c *n) (- *n 1 *n1)
              (set *c *n1)))
```

以下簡単にプリミティブの説明を行なう. 基本的概念は論理型なのでPrologなどに準じている. *で始まるシンボルは変数である.

(as *p) assertの略. プログラムを定義する.

(set *p *v) (assert (*p *v))に同じ. 述語を変数として使うためのsyntax shugar.

(push *e) 環境*eを現在の環境の上に積む.

(pop) 最上位の環境をはずす.

(fork *p) プロセスを発生する.

なお, このプログラムではセルは図6のような構造になっている.

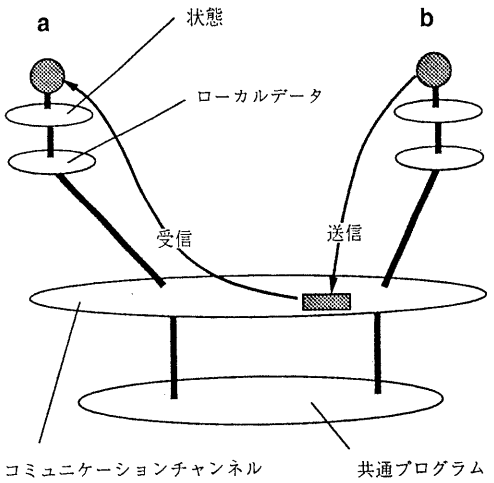

図 6: 交差点問題のセルの構造

さて，このプログラムの定義を変えずに新しい特殊な振舞いをするエージェントをプログラム継承により追加してみよう．エージェントcは基本的にはa, bと同じ振舞いをするが，他のロボットが交差点直前に居れば，自分は止まってしまうという点だけが異なる．これは以下のようにpushの定義の変更により特殊なセル(proceed-c)を余分にpushすることにより実現できる．

```
(push c)
  ; proceed に入る時
  (as (go proceed)
      (pop)
      (push proceed)
      (push proceed-c))
  ; 出る時
  (as (go *state)
      (pop) (pop)
      (push *state))
(pop)

(push proceed-c)
  (as (act) (push a) (loc 0) (pop)
          (go stop))
  (as (act) (push b) (loc 0) (pop)
          (go stop))
(pop)
```

なお，上記はpushとpopが分離した，非構造的プログラミングになっているために若干アドホックなプログラムになっているが，これは例えば

```
(as (in-cell *n *p)
    (push *n)
    *p
    (pop))
```

という定義により構造化可能である．

いずれにしても，pushやpopを操作するgoの再定義というメタレベルプログラミングによってセルの操作が任意に定義できる点が重要である．しかも，この例のように個別のセル(proceed)ではなく複数のセル(proceed-cとproceed)を同時に操作することにより，セルの集団に意味を持たせ，協調の節で述べたようにより上位の抽象レベルの記述を可能にしている点に注目されたい．

5 まとめ

有機的プログラミングの概念を提唱した．有機的プログラミングにおいてはセルの動的組合せと，セル間の（静的）制約記述によってシステムを構築する．これによってシステムの環境変化に対する動的追従の可能性，仕様変化に対する変更の容易性などが実現される．

有機的プログラミングの概念は，ある意味ではオブジェクトオリエンテドパラダイムの拡張になっている．また，プロセスと環境との相互作用は自己反映計算の新しい形と見ることができる．

謝辞

本研究は通産省，産業科学技術研究開発制度“新ソフトウェア構造化モデル”の一環として行なわれた．

有機的プログラミングのモデルは電総研協調アーキテクチャ計画室内での討論を基に産まれた．日頃討論いただく計画室の諸氏，特に石川裕，大澤一郎，木下佳樹，野田五十樹の各氏に感謝する．

参考文献

[1] Ivan M. Havel. Artificial thought and emergent mind. In *Proc. of IJCAI 93*, pages 758–766, 1993.

[2] Takumi Hisano and Motoi Suwa. Synchronization and communication in the 'subject'. In *Logic Programming '85*. Springer-Verlag, LNCS 221, 1986.

[3] Yutaka Ishikawa. Communication mechanism on autonomous objects. In *Proc. of OOPSLA'92*, 1992.

[4] Hideyuki Nakashima. Uranus reference manual. *Bulletin of Electrotechnical Laboratory*, 50, 1986.

[5] Hideyuki Nakashima. Context reflection. In *Proc. of IMSA 92 Workshop*, pages 172–177, 1992.

[6] 中島秀之. 状況を対象とした推論. 人工知能学会誌, 5(5):588–594, 1990.