人　工　知　能　100－4
（１９９５．　５．　２４）

有機的エージェントアーキテクチャ

中島秀之, 大澤一郎, 野田五十樹
電子技術総合研究所
通信知能研究室 & 協調アーキテクチャ計画室

概要

知的エージェントの構成手法を提案する．

知的エージェントは設計時に想定しなかったような新しい状況に適切に対処できる必要がある．Brooksの服属アーキテクチャはそのための一つの手法であるが，対話などを含む高度に知的な作業を行なう記号処理のモデルとしては適切ではない．ここではより高度な作業のために服属アーキテクチャを包含する新しいモデルとして有機的プログラミングを提案する．

1　はじめに

人工知能研究の一つの側面は複雑なシステムの振舞いをいかに柔軟に記述するかにある．求められるソフトウェアやその動作環境が複雑化するに従い，プログラマがその動作を完全に予測しプログラムを構成することは困難になって来た．つまり，プログラムが遭遇するであろう事態とその対処法を事前にすべて書き尽くすことは不可能である．かと言って，何も書いていないのでは対処できない．個々の場面を記述しておけば，その複合体が適当な動作を示してくれることが望まれる．このような，個々の場面記述を統合し，新しい状況に対応できるシステムの構築手法として有機的プログラミングの考え方を提唱する．

環境に柔軟に適応するシステムの構築手法としてはBrooksの服属アーキテクチャ [Brooks 91] があるが，これは反応型エージェントの構築には向いているが言語処理や対話などを含む高度に知的な作業を行なう記号処理のモデルとしては適切ではない．有機的プログラミングはより高度な作業のための記号処理やプラン生成を含んだ一般的モデルであり，服属アーキテクチャを包含する新しい考え方である．特に有機的システムの以下のような特性を取り込むことを目指している：

- 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．

これを別の観点から眺めると，個々の部品に書かれた情報を，環境に応じて柔軟に統合することにより，その集合体としては部品の和以上の仕事をしてくれるような構成法の提案である．環境や他の部品との相互作用により自分の構造を動的に変化させ環境の変化に対応するのである．

これを実現するにはまず各部品を状況依存のかたちに構成し，状況に応じた部品の組み立てを行なうのがよい．本論文ではまずこのような状況依存表現について述べる．その後これを一般化した有機的プログラミングの概要について述べ，それが服属アーキテクチャなどの複雑な動作記述に応用できることを示す．

2　状況依存表現と状況遷移

2.1　状況階層

我々の別の論文 [大澤一郎 95] で定式化したような＜状況－行為＞ペアによる行為の記述を考える．これはプロダクションシステムなどに用いられている＜条件－行為＞ペアの記述とは

- グローバルな条件を記述する必要がない（状況依存でよい）

- 条件を陽にチェックする必要がない（特定の状況にいることを知っていればよい）

という点で異なっている．

概念が階層構造に整理されるがごとく，状況も階層構造に整理できる．この場合，概念階層における属性継承と同様に状況－行為ペアの継承も起こる．一般的状況における行為の結果と特殊状況におけるそれが相違した場合に特殊な方が優先されるのも属性継承と同じである．

さて，このような状況依存表現をとった場合，エージェントが適切に振舞うには，いまどういう状況であるのかを知ることが肝要である．これには以下に述べるように受動的方法と能動的方法がある．

2.2 受動的状況遷移

エージェントは外界の観察により（センサーから入ってくる情報を時々刻々モニターすることにより）現在の状況を知ることができる．しかしながら，状況タイプはセンサーの値には直接対応していないことが多いので様々な計算が必要である．したがって，これをそのまま行なったのでは効率が悪い．更に，自分の行為の結果など，外部センサーからは得られない情報を加味する必要がある（例えば顔の向きを変えると網膜に入る映像は大きく変化するが，これを補正する必要がある）．

プロダクションシステムなどでは黒板を用いることによりこれを解決しているが，黒板がグローバルでは計算量が膨大になるため，黒板を階層的するなどの工夫が行なわれている．しかしながら，黒板の階層性はエージェント側の機能に対応したものであり，外界の状況に対応したものではない．[大澤一郎 95] では状況階層を用いることによりこれを解決している．

2.3 能動的状況遷移

ここでは状況遷移をエージェントの思い込みで能動的に行なう手法を提案する．もちろん，外界の観察によりこれを修正する必要があるので，実際のシステムは受動・能動両遷移を備える必要があるが，膨大な環境からの情報のどの部分に注目するのか，また膨大な内部知識のどの部分を使うのかを決定し，エージェントの行動様式を変更する必要がある場合には能動的手法が有効である．

これを説明するために例をマニュアル車の運転にとる．エンジン回転が上がってくるとシフトアップをするし，回転がある程度以下に落ちるとシフトダウンする．しかし，トップギアで走っている際にはそれ以上シフトアップできないからそのままで走行を続ける必要があるし，ローギアで走っている場合に回転が下がればクラッチを使う必要がある．

これを行なうのにエンジン回転とシフトレバーの位置を入力として，操作を出力とする反応回路をつくることは可能である．しかし，問題は回転が変化したときにシフト操作をする前にレバー位置を観測する必要がある点である．車の運転の場合には目あるいは手でレバー位置を確認する操作が必要である．信号発進の加速中にはこのような確認はいちいち行わないだけでなく，この方式には致命的欠陥がある．もし，トップで回転がある程度以上上がった場合にはレバー位置の確認を続けなければならない．それよりはレバー位置を「覚えておく」方が実用的である．もっと良いのは自分でモードを切り換えることである．

3 有機的プログラミングの概要

3.1 セル

複雑なシステムを構築するにはまずその部品を用意する必要がある．この部品がプログラムモジュールであるが，有機的プログラミングにおいては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．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．

セルは状況の断片に対応する．特定の状況断片にのみ使われるプログラムが各々のセル

に格納されることになる．実際に直面する環境はこのような状況断片の順序つき集合になると考える．

3.2 環境

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1）．

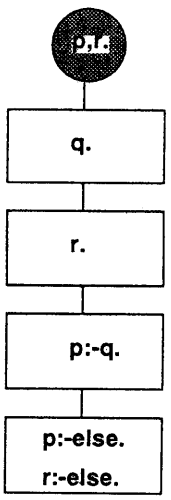

図 1: セルのスタック

エージェントはプロセスとその環境を統合したものに相当する（図2）．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．

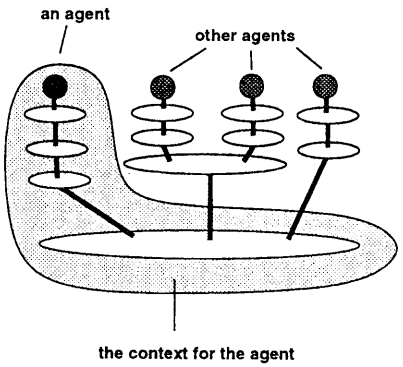

図 2: エージェントと外界

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．具体的にはセルに対して以下の操作を用意する：

1. push (cell)

   現在の環境のトップに cell を積む．下にある環境中の情報は新しいセルで明示的に否定されない限りそのまま継承される（後述の意味論参照）．従って，知識表現でよく使われる IS-A 階層と同等のものが動的に構成される [Nakashima 84] と考えてよい．

2. pop

現在の環境のトップのセルを環境からはずす.

3. use (cell)

   現在使っているセルを cell と入れ換える（モード切替えに使う）.

4. current-context (context)

   現在の環境のリストを context に得る.

5. set-context (context)

   現在の環境を新しいもの (context) に変更する.

3.3 情子

情子とは情報の単位のことである（概念としては状況理論で用いられる infon に近いが，以下の定義は異なる）．我々が設計している有機的プログラミング言語 Gaea[中島秀之 94] では，情子はロールと値のペアの集合として表現される [Nakashima *et al.* 91]. [1]

```
((relation time) (hour 4)
  (minutes 30) (zone jst))
```

通常，少なくとも relation だけは陽に書かれることが多いので，ロールを省略し

```
(time (hour 4) (minutes 30)
  (zone jst))
```

と書くことができる.

また，一部の情報（例えば (zone jst) が環境から明らかな場合（3.4参照）には，それを省略して

```
(time (hour 4) (minutes 30))
```

のように書いても同じ情子と解釈される.

[1] 情子はリストとして表記するが，引数の順序は可変である.

3.4 制約

計算というのは情報を加工することである．つまり，環境の持つ情報と計算主体の情報の相互作用によって計算が進む．[橋田浩一 94]

複数の情報間（異なるセルに存在する場合もある）に制約がある場合，一方の情報が変化した場合にこの制約を満たすように他方も変更することはシステムの仕事である．ユーザはこの制約（これもプログラムとして与える）が守られていることを前提にプログラムすれば良い．システムが提供している制約には以下のようなものがある：

1. 情報間の関係

   (=> *infon1 infon2*)
   (<= *infon1 infon2*)

   => は前向きの推論を表しており，*infon1* が成立しているときには *infon2* も成立していることを表す．<= は Prolog と同じ後向き推論である．これらが通常のプログラムに対応する.

2. セルと情報

   あるセルとプログラムの関係を表すことができる．これも infon である.

   (in *cell infon*)

   で *cell* で *infon* が成立していることを表す.

3. 特殊化

   環境が特殊になればなる程，その特殊環境内では少ない情報操作で計算が行なえる．何故ならその特殊な環境ではより多くの制約が成立しており，それを利用することができるからである.

   具体的にはあるセルの情報の一部を固定することにより，そのセルに属するプログラムに部分計算のようなものを施して固定された情報に全く言及しない新しいプログラムが構成される．これをセルの特殊化 (specialization) と言う．あるセルを特殊化して別のセルを作った場合，両

者には情報の同一性という制約が課される．つまり，一般的なセルcにある情報(p a b)と，このうちaを固定して作った新しいセルc/aにある情報(p b c)[2]の同一性をシステムとして保証する（例えば一方から他方を導く）のである．

4 例

例としてマニュアル車の運転の記述を行なう．シフト時には同時にレバーの位置がわかるのでそれを利用してモード切替えを行うようなプログラムは以下のように書ける．なお，各モードはセルとして表現され，モード切替えはセルの取り替え（useで指示される）によって実現される．（なお，以下のプログラム例では紙面の節約と単純化のためロール名は省略する．）

```
(in normal
 (=> (high-rpm)
   (shift up)
   (if (gear top) (use top)))
 (=> (low-rpm)
   (shift down)
   (if (gear low)) (use low))
 (<= (high-rpm)
   (rev *x)(> *x 3000)))
(in top
 (=> (low-rpm)
   (shift down) (use normal)))
(in low
 (=> (high-rpm)
   (shift up) (use normal))
 (=> (low-rpm) (clutch)))

;;; driving セルには他の一般的
;;; 運転プログラムが書かれている
;;; とすると，top-level は
;;; 例えば以下のようになる

(push driving)
(push low)
(drive)
```

この方式では5速のギアの位置を常に覚えておく必要はなく，3つのモードでよい．更にどのモードか忘れた場合には通常モードに戻ればよく，その場合にはギア操作をしようとして失敗することでトップやローのモードに戻ることが可能である．

なお，ここでの議論はギア位置の確認が動作を伴う高価なものであるという前提にたっている．そうでない場合（例えばKnight-9000のようにエージェントが車と一体化しておりギア位置センサからの情報が常時利用可能である場合）には受動的状況遷移が使える．しかしながら，そのような利用は回路が固定されてしまい，新たな要求仕様への追従は困難である．例えば，実際の運転では上記の条件以外に坂の登り下りなど，様々な別の条件が加味される．有機的プログラミングにおいては，これらの新しい条件も，別のセルに記述することにより容易に統合可能である．例えばシフトのタイミングを遅くしたいときには以下のように高回転の定義を変更したセルを上に（プロセスに違い側に）追加することによりnormalセルのhigh-rpmの定義を上書きできる．

```
(in down-hill
  (<= (high-rpm)
    (rev *x)(> *x 5000)))
```

5 積層アーキテクチャ

これまで見てきたような仕組みで複雑なシステムを構築することを考えよう．これにはBrooksの提案した服属(subsumption)アーキテクチャ[Brooks 91]が有効である．我々はこの考え方を更に拡張し，以下の意味で積層アーキテクチャと呼ぶ：

1. 機能単位の並列重ね合わせとして全体が構成される．[3]

2. 上位機能は下位機能を使って構成される．

[2]従来の，引数の位置による同定法では，このような引数の除去に対応できないので，各引数には役割名を付し，それで区別する．

[3]Brooksは行動(behavior)による分割と呼んでいるが，行動とは外部に現れたもの（環境との相互作用の結果）を指すのが正しいと考える．

3. 上位機能は下位機能の一部を置き換えることができる.

4. 上位機能の発動に失敗したときには下位機能が有効になる.

Brooksの服属アーキテクチャにおいては下位機能を司る回路に直接信号を流して,下位機能を抑止(subsume)する.が,積層アーキテクチャにおいては上位層で下位の機能をフィルタリングする点が異なる.下位機能はそのままの形で残されるので4のようなことができる.これは上位機能が全く使われなかった場合のみではなく,使われて失敗した後にもバックトラックするように設定することが可能である.

先の例で<坂道運転>は<運転>の上位機能に当たる.ここではhigh-rpmの定義を置き換えることにより下位機能に変更を加えている.

6 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.

また,小さな記述単位(セル)を動的にまとめることにより複雑なシステムの構築が容易になることを示した.

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ならびにIPAの諸氏に感謝する.

参考文献

[Brooks 91] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, Vol. 47, pp.139–160, 1991. (柴田 正良 訳. 表象なしの知能, 現代思想, **18** (3), 85–105).

[Nakashima 84] Hideyuki Nakashima. Knowledge representation in Prolog/KR. In *Proc. of 1984 International Symposium on Logic Programming*. IEEE, 1984.

[Nakashima *et al.* 91] Hideyuki Nakashima, Ichiro Ohsawa, and Yoshiki Kinoshita. Inference with mental situations. TR-91-7, ETL, 1991.

[橋田浩一 94] 橋田浩一, 松原仁. 知能の設計原理に関する試論－部分性・散層・フレーム問題－. 日本認知科学会年報「認知科学の発展」, Vol. 7, pp.159–201, 1994.

[大澤一郎 95] 大澤一郎, 中島秀之. 因果関係と状況に基づく動作効果の動的計算. 人工知能学会誌, Vol. 10, No. 2, pp.58–67, 1995.

[中島秀之 94] 中島秀之. 有機プログラミング事始め. プログラミングシンポジウム報告集, pp. 161–167, 1994.