ソフトウェア工学 112－1
（1996．11．15）

進化するプログラム

野村　章
あさひ銀総合システム株式会社
〒107　東京都港区南青山3-10-43（勤務先）　☎ 03-3403-3131
/E-mail VZF03342@niftyserve.or.jp
〒194　町田市南大谷521（自宅）　☎ 0427-29-5190

あらまし　本論文では、プログラムを動的に進化させる方法を論じる。通常、コンピュータを用いて新規にプログラムを自動的あるいは半自動的に作りだすことを「プログラム合成」という。運用中のプログラムが様々なトリガーによって自らの機能を高めたり、環境に適応して変化したりする「ゆらぎ」の構造をもつと便利である。これを「進化」と呼ぶ。進化するプログラムは、再利用する際に手作りで埋めなければならない「差分」をミニマムにする。Prologのもつ機能を用いてプロトタイプによる実験をおこない、その実現可能性と応用の限界を考察した。

A Study of Self-evolving Programs

Akira Nomura
Asahigin Systems Co.,Ltd.
Minami-Aoyama 3-10-43,
Minato-ku, Tokyo, Japan
+81-3-3403-3131
/E-mail VZF03342@niftyserve.or.jp

Abstract: This paper describes a method to evolve programs during execution ,accumulate the newly born functions and keep them alive at next run time. Two prototype programs were built and used to confirm the possibilities of 'self-evolving programs' and predict their real world availability in future, for instance, in Case-Based Reasoning, 'Software Construction from Parts' paradigm and so on. Some problems to be resolved in future work are discussed.

第一章　はじめに

本論文は、進化するプログラム(Self-evolving Program)の構造について述べる。仕様からプログラムを合成するには、自動化の程度の差はあれ人手でを介して合成をおこなう。しかし、一度完成されたプログラムを、運用中に必要に応じて自動的にあるいは人間との対話を介して進化させることができると、環境変化や利用者のニーズに柔軟に適応可能(adaptable) なソフトウエアが期待できる。

[Neumann 66]が遺伝情報を用いた29状態モデルにより「自己増殖機械」の構成可能性を示した。[菅田 85]は遺伝情報を使わないで自分の体を調べながら自己複製をおこなった。一方過去の設計事例を材料に、これを加工して要求仕様を充足するように改良する枠組みを「事例ベース推論（ＣＢＲ)」という。すなわち、ＣＢＲでは、蓄積した事例データベースから、問題事例に類似した設計事例を検索し、類比あるいは類推機能により新しいルールを作り出す。このルールをプログラムと読み替えると「プログラム合成」になる。通常ＣＢＲでは、発見したルールを問題解決に利用した後、この新しく発見されたルールを含む問題事例を事例データベースに蓄積して、別の問題解決に役立てることになる。この場合、発見されたルールを人間が理解して問題事例に組み込む（スロットやスクリプトを追加する）ことが行われる。本論文に述べる「進化するプログラム」の枠組みを用いると、問題事例に対応したプログラムを自動的に改良したり新しく発見したルールを人手を用いないでシステムに組み込むことが期待できる。

Prologは述語論理に基づく言語である。要求仕様の記述や各種推論を得意とし、バックトラックによる検索機能を内蔵している。帰納推論を論理プログラムに適用してプログラムの作成・検証をおこなう「モデル推論」とＣＢＲを結び付けて新しいルールを発見する研究がある[Shapiro 81][Shapiro 83]。また、Prologには「メタ言語」の機能があり、これを用いてプログラムの編集機能やオペレーティング・システムが作られている。「反転可能なインタプリタ」を用いて、例からプログラムを合成する研究がある[沼尾、志村 91]。本論文では、MacProlog32 (LPA 社の商標) を用いてプログラムに改良機能や追加機能を累積的に盛り込むメカニズムを提案し、問題解決に必要な差分的仕様を検出して自動的に機能を補間する手法の実現可能性を考察する。

第二章　メタ言語機能

この章では、Prologの持つ「メタ言語」機能について述べる。

プログラム自身をデータとして扱う述語を「メタ述語 (meta predicates)」という。DEC-10 Prolog系のMacProlog32には、retract, assert, setof, clause, …などがある[LPA 94]。「メタ述語」を用いてインタプリタを作ることができる。

図1は、「進化するPrologプログラム(Self-evolving Programming)」の枠組みを示すため、MacProlog32のメモリーの使用方法を図示し、外部ファイルとプログラムおよびメモリー間のインタラクションを表現したものである。

本論文で扱うメタ機能は次の五つである。

(1) ワークスペースに対する述語やデータの動的な割り付けと消しこみ

図1のMeta Codes 領域でメタ述語 assertx/2, retractx/2 を用いて、Work Space 領域に実行可能な述語を書きこんだり、取り消したりできる。

例えば、メタ述語 assertx(love(jack,betty),1) によって、Work Space 領域の頭部に love(jack,betty) という述語を割り付けることができる。

また、メタ述語 retractx(love/2,1) によって、Work Space 領域の頭部の述語 love(jack,betty) を消しこむことができる。

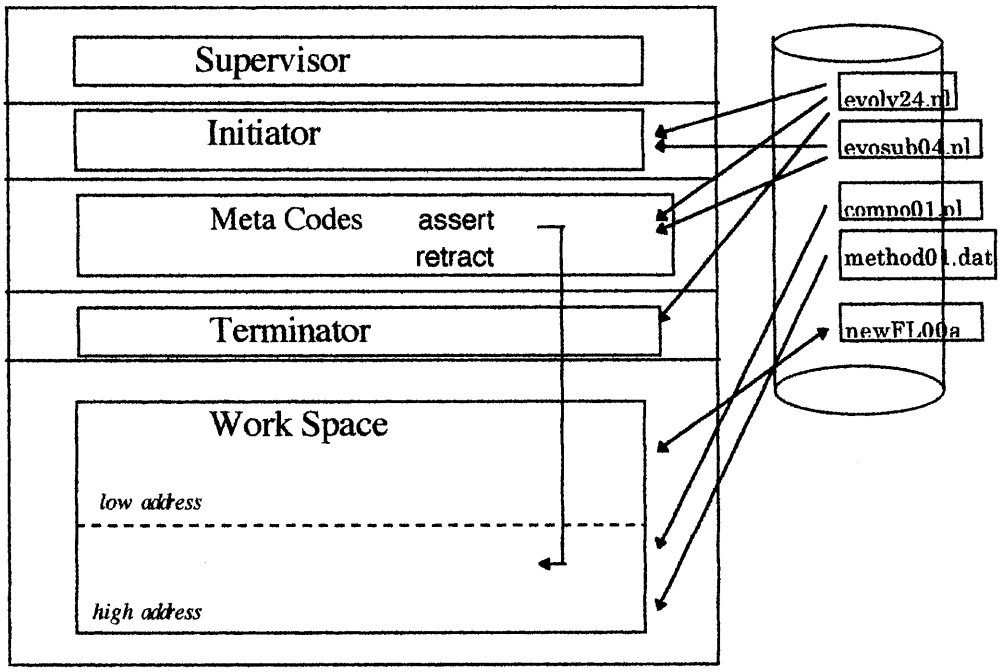

図1 メモリー使用法と外部ファイルの関係

(2) 樹状データベースに対する検索

ツリー形データベースのバックトラックによる検索は、Prolog の特徴である。

(3) 外部ファイル上のコンポーネントの動的な取り込みと実行

図1のWork Space 領域にある述語を外部記憶装置上のファイル compo01.pl に追記するには、次のような指定をすればよい。

fopen('compo01.pl'), output('compo01.pl'), listing([love/2,…]),
fclose('compo01.pl').

また、次のように compo01.pl から、Work Space 領域に述語を読み込む。

load_files('compo01.pl',[if(true),load_type(compile),all_dynamic(true)]).

(4) プログラム内での述語の動的な形成と実行

図1のMeta Codes 領域で動的に新しい述語を形成して、Work Space 領域に割り付けるには、次のように指定する。

atom_string('love(X,Y)',String1), read(P)<~ String1, assertx(P,999).

(5) ワークスペースの外部ファイルへの書き出しと再ロード

次の指定により、図1のWork Space にある述語やデータを外部記憶装上のファイル、たとえば 'newFL00a' に書き出す。

fopen('newFL00a',1), output('newFL00a',1), listing([love/2,…]),
fclose('newFL00a',1),…

また、外部記憶装置上のファイル 'newFL00a' に書き出したWork Space 上のプログラムをWork Space に戻すには、次のように指定をする。

load_files('newFL00a',[if(true),load_type(compile),all_dynamic(true)]).

プログラムをスタートするときに、直前のWork Space の状態に外部記憶装置上のファイル 、たとえば 'newFL00a' から書き戻し、プログラムを終了するときに、その時点のWork Space の状態を同じファイルに書き出しておけば、プログラムは常に過去の状態より進化した最新の状態を保持することができる。

第三章　プロトタイプ

「進化するプログラム」の実現可能性を実証するため、次のようなミニプログラムを作成した。

(1) プログラム1：述語 write_1/n のアリティ n を1から5まで順次増加して述語write_1 の機能を進化させる。

(2) プログラム2：ツリー形データベース（図1のWorkSpace 上にロードされている）のノードに格納されている文字列で表わされた述語を会話形式で指定して、Meta Codes 領域に読みこんで実行可能な述語に変換したうえで、Work Space にセットする。以後この新しい述語を保持して、新しい機能を追加するようにする。ノードに格納されている文字列の例を次に示す。

```
% link(node1,node2)
link(x,rt). link(r,a). link(r;b). link(r,e). link(a,c). link(a,d).
% link4(noden,upperclass,[body of method1 = predicates])
link4(r,[], %This is root class.[]).
link4(a,r, %This is a supper_class.
 ['sin_X(X,A):-A is sin(X).']).
link4(b,r, %This is a supper_class.
 ['cos_X(X,A):-A is cos(X).']).
link4(e,r, %This is a supper_class.
 ['component_X(A):-component1(A).','component1(''compo01.pl'').']).
link4(c,a, %This is a sub_class.
 ['tan_X(X,A):-A is tan(X).']).
link4(d,a, %This is a sub_class.
 ['powr_X(X,Y,A):-A is X ^ Y.']).
```

第四章　実験結果

次に実験の結果を示す。

(1)　プログラム１では、述語 write_l/n のアリティが+1づつ増加するかたちで指定すると順調に述語のアリティに巾ができて機能が増加する。しかし、述語write_l/n が次のように積み上げ形の構造をしているので、アリティを+2以上いっきに増加するかたちで指定すると、アリティを増やそうとした述語の創出は失敗する。

```
% write_l/ (n+1)
write_l (A,B,…) :-write_l/n,write_l/1,nl.
```

プログラム１は、いったん述語を登録すると、プログラムを修了したのち、再度実行しても再登録することなく有効に働き、プログラムの機能が以前よりも進化しているのが確認された。

(2)　プログラム２では、method01.dat 上のツリー形データベースのノードを外部から指定することにより、Prolog に検索させた。ノードに記録された文字列表現の述語を実行可能な形式に変換して Work Space にセットし、実行することができた。いったん取り込まれた述語は、プログラムの実行が修了したのちも、プログラムの内部に保持された。一方、プログラムの実行途中で外部記憶装置上のファイル 'compo01.pl' から Prolog のソースコードを入力して実行し、以後これらのソースコードはプログラムの一部として機能し続けることも確認した。

以下にテスト出力を示す。

```
% pArea/1          Work Space の状態
pArea (a).
Key in Start,End (Goal).
 as like 'ra' or 'rd'
rd was entered.
From  r to d  (Goal)
% pArea/1          Work Space の状態
pArea (a).
pArea (powr_X).
Key in a Command.
as like 'sin_X (23,ANS)' or' powr_X (23,3,ANS)' or' component1 ('compo01.pl')'
'powr_X (23,3,ANS)' was entered.
ANS= 12167   <= Result
% powr_X/3          Work Space の状態
powr_X (A, B, C) :-C is A^B.
% pArea/1
pArea (a).
pArea (powr_X).
[Process End]

Reconstruct The Workspace.
```

```
Key in Start,End(Goal).
 as like 'ra' or 'rd'
re was entered.
From  r to e  (Goal)
% pArea/1          Work Space の状態
pArea(a).
pArea(powr_X).
pArea(component_X).
Key in a Command.
as like 'sin_X(23,ANS)'or'powr_X(23,3,ANS)'or'component1('compo01.pl')'
'component1('compo01.pl')' was entered.
[ This is the output from compo01.pl.
  You are able to understand from this examle
  that some kind of program can evolve by itsself.
  And also you will find any commponent could be
  attached to it dynamically from commponent files. ]
% powr_X/3          Work Space の状態
powr_X(A, B, C) :-C is A^B.
% run/0
run :-nl,nl,
  write('[ This is the output from compo01.pl.'),nl,
  write('  You are able to understand from this examle'),nl,
  write('  that some kind of program can evolve by itsself.'),nl,
  write('  And also you will find any commponent could be'),nl,
  write('  attached to it dynamically from commponent files. ]'),nl.
% pArea/1
pArea(a).
pArea(powr_X).
pArea(component_X).
[Process End]
```

第五章　考察

前の二章で、prolog を用いて継続的に機能を進化させるプログラムの構造を示した。これまでに種々の自動合成法や再利用法が提案されてきたが、継続的に機能を進化させるプログラムの構造をもつものはなかった。

プログラムの合成を一般的におこなおうとすると、いわゆる「組み合わせ的爆発」を起こす。「進化するプログラム」についても、設計を行うとき環境条件や入出力条件をある粒度で指定しないと「組み合わせ的爆発」をおこすので、本質的にはプログラム合成と同様の限界がある。

したがって、進化するプログラムを設計する場合、対象領域を適切に絞り、特定目的に限定するとともに、独立した機能に分割できなければならない。

プログラム進化のトリガーとして、次のようなものがある。

① 外部からの指示
② 与えられたデータの含む条件
③ プログラム自体のプロセスから生じるニーズ

これらのトリガーから生じる進化のニーズを「差分」として正確に把握し、問題解決の方法を確定して必要なプログラム・コンポーネントを検索したり合成する手法の検討が必要である。そのためには静的なオブジェクト図や動的な状態遷移図などをリアルタイムで検索したり、メンテナンスしたりする技術が重要となる。

また、メタ機能を活用するためには、プログラムのソースコードの現状をデータとして常に把握できる仕組が必要である。すなわち、プログラム自身が自らの姿を映す鏡がなければならない。たとえば、Work Space の何行目にどの述語を挿入するかを判断したり、ソースコードのどこをどのように修正するかを決める必要があるからである。

プログラムを進化させるとき、誤った内容を作りこむ危険性もある。限定された領域において、証明可能なものやあらかじめよく検討されたシナリオに基づくものだけが許容されるようになっていなければならない。

近年ソフトウエアの再利用を目標としてオブジェクト指向に基づくソフトウエア開発がおこなわれている。オブジェクト指向においては、各オブジェクトのもつ機能（振る舞い）が外部から見て確定している必要がある。あるオブジェクトが他のオブジェクトに処理を依頼したり自らに依頼された仕事を他のオブジェクトに委託するとき、相手のオブジェクトの守備範囲(responsibilities of object)や機能が見えている必要があるからである[Wirfs-Brock 90]。したがって、プログラムの進化もオブジェクトとしてのプログラムの守備範囲や求められている機能からはずれない程度に制約を加える必要がある。そうしないと「オブジェクト相互の関係（associations between objects)」が不明確となり、システム全体の振る舞いが予測できないものになってしまう。オブジェクトは、属性（attributes)、他のオブジェクトとの関係(associations)、機能（functions)、目的（purposes）によって、特徴づけられる。オブジェクト指向でプログラム進化を考えるとき、個々のドメイン・システムのオブジェクトについて、目的に応じてどの程度機能の自由度を認めるかをあらかじめ検討しておく必要がある。

以上のような仕組を使ってプログラムを進化させるとき、生物の自己保存と対比して、ドメインにおけるプログラムの単体としての独自性をどこまで、どのようにして保持するかも検討されなければならない[Maturana & Valera 73]。

プログラム合成の方法として、次のようなものがある[原口 90]。

① 仕様が所与のものとして、プログラムを形式的変換により合成する方法
② 入出力例から帰納推論により、プログラムを合成する方法
③ 超高水準言語により仕様を記述し、コンパイラでプログラムを合成する方法[石川 96]
④ プログラム部品を用いて、プログラムを合成する方法
⑤ データベースを利用した知識処理により、プログラムを合成する方法

このうち、②については前述の[沼尾、志村91]の研究、③についてはC++のメタ言語を用いた[石川 96]の研究、⑤についてはprologの検索機能、述語論理およびメタ言語機能を用いてプログラム機能の拡張を行った研究がある[野村・寺野 96]。

第六章　まとめ

進化するプログラムをprologの「メタ言語」を用いて実現する基本的な枠組みを述べた。ダイアグラムなどを用いて差分を効率的に検出する手法の検討が必要である。「ゆらぎ」を作り出す方法はいくつか考えられるが、それらをドメイン知識を用いてインプリメントし、実証するのが今後の第一の課題である。第二の課題は、進化するプログラムがプログラムの再利用においてどの程度有効かを評価することである。

参考文献

1. [Neumann 66] von Neumann, J. :Theory of Self-Reproducing Automata, edited and completed by Burks, A. W., University of Illinois Press (1966)
2. [菅田 85] 菅田、森田、岩村、三井:記述（遺伝子）なしの自己増殖セル・オートマトンについて、数理科学、No. 265, pp. 7-16 (1985. 7)
3. [Shapiro 81] Shapiro, E. :Inductive Inference of Theories from Facts, Yale DCS TR-192 (1981)
4. [Shapiro 83] Shapiro, E. :Algorithmic Program Debugging, MIT Press (1983)
5. [沼尾、志村91] メタインタプリタによる帰納的プログラム合成規則の学習、人工知能学会誌、Vol. 6 No. 6 pp. 920-927 (1991)
6. [LPA 94] Logic Programming Associates Ltd. :LPA-PROLOG Technical Reference, Logic Programming Associates Ltd. (1994)
7. [Maturana & Valera 73] H. R. Maturana and F. J. Varela: オートポイエーシス　生命の有機構成、国文社、pp. 45-159, (1996)
8. [石川 96] 石川裕：適応可能言語システム、オブジェクト指向最前線、朝倉書店、pp. 207-208 (1996)
9. [Wirfs-Brock 90] Wirfs-Brock, R. and B. Wilkerson: Designing Object-Oriented Softwaae, Prentice Hall, Englewood Cliffs, NJ (1990)
10. [原口 90] 原口誠：自動プログラミング, 人工知能学会編「人工知能ハンドブック」1. 基礎編6章 pp. 109-113, オーム社 (1990)
11. [野村・寺野 96] 野村章・寺野隆雄：ドメイン知識の獲得と再利用のための類似オブジェクト探索システム、信学技法AI95-54, pp. 95-102 (1996)