E-09

自動関数定義を用いた遺伝的プログラミングによる排他制御プログラムの生成

Generation of Concurrency Control Program by Using Genetic Programming with Automatically Defined Functions

田村 真司†　　宝珍 輝尚†　　野宮 浩揮†
Shinji Tamura　Teruhisa Hochin　Hiroki Nomiya

1．はじめに

現在，様々なコンピュータが環境に合わせて発達し，それと同時にデータベースが多種多様な場面で必要となっている．しかし，様々な分野の要求に応えるには，単一の汎用的なデータベース管理システム(DataBase Management System, DBMS)では対応できない部分が出てくる．このことから，各分野に特化した機能を持つ DBMS が望まれる[1]が，その構築は難しい．また，実現できたとしても特定の場面でしか使わない不要な機能が多くなる恐れがある．つまり，各分野に特化した DBMS が必要となるのだが，その構築には多大なコストと時間が必要となる．このことから，DBMS の拡張性が強く求められている[2].

本研究では，DBMS の性能や信頼性にかかわる最も重要な機構の一つである排他制御機構の生成について検討する．データベースを利用する際には，分野によって異なるトランザクションが発生すると考えられる．それは例えば，データを読む操作がほぼ全てを占めていたり，ある一つのデータに対して操作が集中したり，といったように様々な特徴があり，ある特徴に対しては適切な排他制御機構も別の特徴を持つ分野で運用すると著しく効率が落ちる可能性がある．発生するトランザクションの特徴に応じて最適な排他制御機構を生成できるならば，利用分野に最適な DBMS の構築に繋がる．

一方，人工知能の分野では，生物が環境に適応して進化していく過程を模倣した遺伝的プログラミングという手法が開発されてきている．遺伝的プログラミングでは，プログラムを一定のアルゴリズムに従って進化させることで，環境に適したプログラムが得られる．

これまでに筆者らは，トランザクションの特徴に応じた排他制御プログラムの自動生成を目的として，遺伝的プログラミングにより，様々な排他制御プログラムの生成を可能とする手法を提案してきた[3]．トランザクションの特徴に応じた排他制御プログラムが生成できる可能性を示してきたものの，同一のプログラム部分を使用すべき所で同一の処理を生成する確率は非常に低く，効率の良いプログラム生成は困難であった．また，排他制御プログラム生成のもととなるスケジュールは，ごく基本的なものであり，現実のスケジュールからはかなり乖離したものであった．

そこで本論文では，より効率良く，より現実的な排他制御プログラムの自動生成を目的として，排他制御プログラム生成の際に関数(サブルーチン)も自動生成させることで，生成するプログラムの効率を向上させ，より有益なプログラムの生成を試みる．提案する手法は，トランザクションの特徴に適合するように，自動関数定義を用いた遺伝的プログラミングで様々な排他制御プログラムを生成可能とする手法である．また，より現実的なスケジュールとして，トランザクション処理性能評議会(TPC)が策定したベンチマーク[5]で使用するスケジュールをもとにする．

以降，2．で遺伝的プログラミングや自動関数定義について概説する．次に，3．で排他制御プログラム自動生成システムについて述べ，4．で改善点について述べる．最後に 5．でまとめる．

† 京都工芸繊維大学大学院、 Kyoto Institute of Technology

2．遺伝的プログラミング

2.1　遺伝的プログラミング

遺伝的アルゴリズム(Genetic Algorithm, GA)とは進化論的な考え方に基づいてデータを操作し，最適化の問題や学習，推論を扱う手法である[4]．GA のアルゴリズムを以下に示す．

(1)現世代の集団を生成する．
(2)現世代の集団内の各個体に対して適合度を計算する．
(3)適合度をもとに現世代の集団から個体を選択する．
(4)(3)で選択した個体に対して，突然変異・交叉などの操作を行い，次世代を生成する．
(5)(2)～(4)を必要数繰り返し，全ての世代の中で最高の適合度を持つ個体を解とする．

遺伝的プログラミング(Genetic Programming, GP)は GA を構造的な表現ができるように拡張したものである[4]．GP における個体を木構造で表わし，突然変異・交叉などの遺伝的操作を部分木の変更によって実現させる．また，木構造は Lisp で使われる S 式で表わすことができる．プログラムを表わす木構造(プログラム木表現)と S 式の対応関係を図 1 に示す．　図 1 の F1，F2 は非終端記号，T1，T2 は終端記号を表わしている．

2.2　自動関数定義

人間のプログラマが作成したプログラムと GP で生成したプログラムの違いとして，関数(サブルーチン)の有無が挙げられる．関数を使用しない場合は，同じ処理をする部分がプログラム中に何度も登場してしまい，そのプログラムを格納するために，大量のリソースが必要となる．これに対して，関数を使用することにより，冗長性が低下し，結果として探索が効率化できる可能性がある．

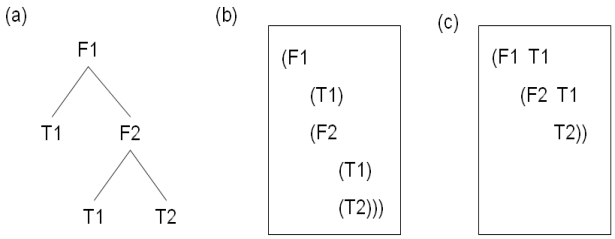

図 1. プログラムの(a)木構造表現(b)S 式表現と(c) S 式の省略表現

また，自動関数化手法は GP の探索原理の根拠を示している，という見方もある．元々，GA の探索原理の根拠としてスキーマ定理と積み木仮説があり，この仮説では各集団のもつスキーマと呼ばれる最適解の部分解を交叉・突然変異によって組み合わせて最適解を構成していくと説明されている．このスキーマが GP においては部分木であると考えられている．つまり，自動関数定義(ADF)とは，有効な部分木を組み合わせることで，プログラムの短縮化による探索効率の向上や理論的な裏付けが実現できる手法である[4].

ADF を用いた GP を ADFGP と呼び，図 2 のように通常の生成プログラムを右側に，もう一方の左側に DEFUN を用いて定義した ADF を存在させる．このとき，VALUES 以下の ADF0_body に ADF0 のプログラム本体が存在する．また，左側で定義した ADF0 は非終端記号名 ADF0 として右側の生成プログラムで自由に使用することができる．

また，GP の実行に際して，左側の ADF0 の定義部分にも右側の生成プログラムと同様に交叉や突然変異を行う．ただし，ADF0 の定義部分は ADF0 の定義部分同士で，右側の生成プログラムの定義部分は生成プログラムの定義部分同士で交叉を行う．

システムの仕様として ADF の数，引数の数，記号の種別などを決める必要がある．

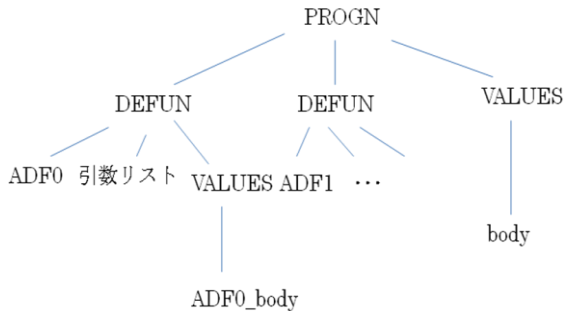

図 2. ADFGP のデータ構造

3. 排他制御プログラム生成システム

3.1 排他制御プログラム生成システムの概要

本システムは大きく分けて二つのシステムから構成されている．大まかな流れとしては，まず「スケジュール生成システム」によりスケジュールを生成し，そのスケジュールを用いて「排他制御プログラム生成システム」で排他制御プログラムを生成する．

「スケジュール生成システム」は初期設定ファイルでトランザクションの数や操作するデータの数，含まれる書き込み(WRITE)命令の割合等を指定し，初期設定に基づいたスケジュールを生成する．

次に，「排他制御プログラム生成システム」の概要を図 3 に示す．このシステムは以下の順で実行される．

(1) 遺伝的操作により排他制御プログラムを得る．
(2) 排他制御プログラムで得たスケジュールを制御して適合度を得る．
(3) 最終世代に到っていなければ(2)で得た適合度をもとに(1)の操作に戻る．最終世代なら終了して最良個体を得る．

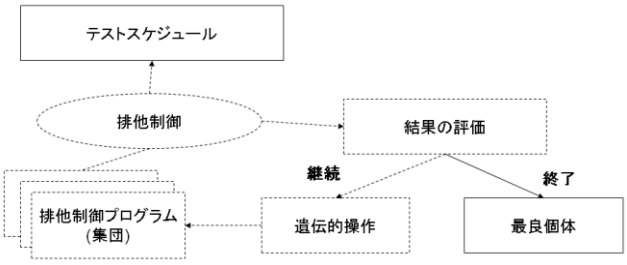

図 3. 排他制御プログラム生成システムの概要

3.2 排他制御プログラム生成システムの設定

GP によるプログラム生成のための設定も必要である．以下の設定項目がある．

- s1：最大世代数
- s2：一世代における個体数
- s3：初期世代になるプログラムの指定の有無
- s4：プログラムの最大深さ
- s5：プログラムの成長方法
- s6：非終端記号における交叉確率
- s7：終端記号における交叉確率
- s8：コピーのみを行う確率
- s9：評価に用いるスケジュールの数
- s10：小さいプログラムを重視する割合

s3 において初期世代のプログラムがあるとする場合，プログラム（S 式表現）を記述した外部ファイルが必要である．初期世代になるプログラムを指定すると，初期世代の集団の半分程度が指定したプログラムとなり，以降の進化に影響を及ぼすこととなる．s5 では，プログラムをランダムに成長させるか，各枝を必ず s4 の最大深さまで成長させるかを設定できる．s6，s7，s8 は遺伝的操作の頻度の設定である．突然変異が起こるのは，交叉もコピーもしない場合である．s9 および s10 は適合度計算に関係する．

3.3 ノード

二相ロック法や時刻印順序法などの代表的な排他制御を実現するプログラムを分解して得たノードではなく，より汎用的な排他制御プログラム作成システムとするために，ノードをより一般的な基本機能である変数の値の操作と定めた[3].

具体的には，データやトランザクションが個別に持つ変数を宣言・操作可能とし，この変数を不定変数と呼ぶことにした．また，排他制御ではデータとそれを操作するトランザクションを関係づけて管理することが多いため，データとトランザクション間で持つ特別な変数も宣言・操作可能とし，この変数を共有不定変数と呼ぶことにした．

提案した 2 つの変数に対する処理を行うノードを代表的な排他制御法である二相ロック法や時刻印順序法を表すことができるように設定し，それらのノードを組み合わせることで新しい排他制御法を得られることを期待している．既存のノードについては最後に付録として掲載する．

3.4 適合度の計算

適合度の計算に関わるものは大きく分けて 6 つあり，以下の順で計算する．

（1） 直列可能性の検証

直列可能性の検証とは，複数の処理を並行して行った時に，それらの処理を順番に行った時と同じ結果が得られるかどうかを検証することである．

制御済みスケジュールにおいて，直列可能性が保証される場合のみ以降の適合度計算を行う．直列可能性が保証されない場合は適合度を最低に設定する．ここでは，直列可能性の検証法として先行グラフを採用し，競合直列可能性を調べる．

（2） 同時実行性の検証

この計算を行う時は，直列可能性が保証されているので，単純に一度に実行できるトランザクションの数が多ければ多いほど効率的になると考えてよい．

直列スケジュールの長さを l_0，制御後スケジュールにおける同時実行オペレーションの最大同時実行数を $c_{\max}$，同時実行を考慮した総ステップ数をスケジュールの長さで割ったものを平均同時実行数 c_{ave} として，式(1)により得られる値を同時実行性の評価値とする．

$$E_1 = l_0 \times c_{\max} \times c_{ave} \qquad (1)$$

（3） トランザクションの応答時間

トランザクションの集合$\{T_1,T_2,\cdots,T_n\}$があるとき，T_n の直列スケジュールにおける応答時間を ts_n，制御済みスケジュールにおける応答時間を tc_n とする．また，n_{abort} はアボートしたトランザクションの数，ts は直列スケジュールにおけるスケジュール開始から終了までの応答時間，tc は制御済みスケジュールにおけるスケジュール開始から終了までの応答時間とする．式（2）の計算を行い，トランザクションの応答時間の評価値とする．

$$E_2 = \frac{1}{2n} \cdot \{(n - n_{abort}) \cdot \frac{ts}{tc} + \sum_{k=1}^{n} \frac{ts_k}{tc_k}\} \quad (2)$$

ここでの応答時間は，オペレーション READ，WRITE の時間とそのスケジュールを制御するときの不定変数を宣言する時間を足したものとする．実機を用いて評価を行い，READ・WRITE・不定変数の宣言を 10：20：1 とすることにした．

（4） トランザクションのアボート率

全トランザクションのうちアボートされた割合 p_{abort} を求め，式(3)の計算を行い，トランザクションのアボート率の評価値とする．

アボート率が 0.5 以上の時，性能が悪いことが容易に分かるため，2 乗している．

$$E_3 = \begin{cases} 1 - p_{abort} & if \quad p_{abort} < 0.5 \\ (1 - p_{abort})^2 & otherwise \end{cases} \qquad (3)$$

（5） プログラムサイズによる補正

プログラムサイズを考慮する計算を行う．

具体的には S 式プログラムの記号(非終端記号と終端記号)を数えて N とし，プログラム起動前に設定しておいた，どれだけプログラムの大きさを重視するかの値 α を乗算して得られた値をプログラムサイズの評価値とする．

$$E_4 = N \times \alpha \qquad (4)$$

（6） 全テストスケジュールに対する結果の評価

(1)～(5)までは各テストスケジュールに対しての評価であったが，排他制御プログラムで競合等価な制御済みスケジュールが得られないという結果があっては問題なので，全てのテストスケジュールに対して制御済みスケジュールが競合等価であれば適合度を大きくすることとして，式（5）を採用した．

$$E_5 = \begin{cases} 1.5 & if \quad p_{not\_conflict\_equal} = 0 \\ 1 - p_{not\_conflict\_equal} & else \quad if \quad p_{not\_conflict\_equalt} < 0.5 \\ (1 - p_{not\_conflict\_equal})^2 & otherwise \end{cases} \qquad (5)$$

ここで，$p_{not_conflict_equal}$はテストスケジュールを操作して得た制御済みスケジュールが競合等価ではない割合である．

（7） 最終的な適合度

（1）～（6）までの計算を行い，最終的な適合度 E を求める．今までの結果をまとめると式（6）となる．ここで max_schedule はテストスケジュール数，$E_1(s)$はテストスケジュール s に対する E_1 である．

$$E = \left(\frac{\left(\sum_{s=0}^{\max\_schedule - 1} E_1(s) \times E_2(s) \times E_3(s) \right)}{\max\_schedule} - E_4 \right) \times E_5 \qquad (6)$$

4. 排他制御プログラム生成システムの改良

ここでは，排他制御プログラム生成システムの改良として，遺伝的プログラミングにおける ADF の導入，ならびに，スケジュール生成システムの改良について述べる．

4.1 ADF 導入のための初期設定

ADF の初期設定として各 ADF に対して以下の設定項目がある．ここで，ADF の数を n として，各 ADF を ADF_0, ADF_1，･･･，ADF_i，･･･，ADF_{n-1}と表わす．

- ADF_iの新規生成時の最大深さ
- ADF_iの交叉時の最大深さ
- ADF_iの突然変異時の最大深さ
- ADF_iの最低深さ
- ADF_iの成長方法

4.2 スケジュール生成システムの改良

ここでは，TPC-C を基にしたスケジュールを生成するように改良する．TPC-C は TPC が策定したベンチマークの一つである．TPC-C は，従来から行われてきたオンライントランザクション処理のためのベンチマークで，想定している業務は受注販売である[5]．TPC-C では「新規注文」・「支払い」・「注文確認」・「配達」・「在庫確認」の 5 つのトランザクションから成り，各トランザクションにより指定されたデータに対してアクセスを行う．

ここで，スケジュールは READ 命令と WRITE 命令からだけ成るものとする．単純に WRITE 命令の発生確率等を設定し，スケジュールを生成しない理由としては，より現実に近い実践的なスケジュールを用いて実験を行い，そのスケジュールに対して既存の排他制御法よりも効率的に処理できるかどうかの実験を行うためである．

しかし，全てを再現することは難しいため，ここでは SELECT 命令を READ 命令一つ，UPDATE 命令を READ 命令と WRITE 命令一つずつとして作製した．また，GP で実験を行うことを考慮すると各トランザクションの長さが問題になったため，競合に関係ないデータ操作については省略するものとした．

初期設定として以下の項目を設定可能とする．

T1：倉庫数
T2：各トランザクションの発生確率
T3：1 スケジュールに含まれるトランザクション数
T4：トランザクションの開始時間差
T5：初期新規注文リレーションのタプル数
T6：生成スケジュール数

T1 が多ければ多いほど各データのタプル数が増え，競合する可能性が減り，オペレーションの数が増える．トランザクションは全て同時に発生するのではなく，ある程度の時間差で発生するため，T4 で 1 スケジュール中に発生するトランザクションの時間差を設定する．また，T5 は配達トランザクションを行う際に新規注文リレーションのタプル数によりオペレーション数が変化するため初期設定として変更可能とした．

4.3 適合度計算の改良

3.4 の(2)では直列スケジュールの長さl_0として使っていたが，TPC-C を基に作成したスケジュール生成プログラムではほぼ全て READ 命令の長大なスケジュールを生成する可能性があるため，次に示すように変更を行った．

直列スケジュールの WRITE 命令の数をl_{write}，制御後スケジュールにおける同時実行オペレーションの最大同時実行数を$c_{\max}$，同時実行を考慮した総ステップ数でスケジュールの長さを割ったものを平均同時実行数c_{ave}として，式(7)により得られる値を式(1)の代わりに同時実行性の評価値とする．

$$E_1 = l_{write}{}^2 \times c_{\max} \times c_{ave} \qquad (7)$$

5. おわりに

自動的関数定義を用いた排他制御プログラム生成システムの提案を行った．また，それに伴いスケジュールの生成方法や全体のシステムの見直し，設定項目の変化や評価方法の精錬を行った．

今後はこの仕様で実験を行い，その後単純な ADF ではなく MA(Module Acquisition)のライブラリの考え方を加えた COAST(Collective ADF for subroutine acquisition)[4]を実装してより環境に適応した排他制御プログラムの生成を目指していく予定である．

参考文献

[1] Stonebraker，M．and Cetintemel，U．：“One Size Fits All：An Idea Whose Time Has Come and Gone，”Proc. of the 21st ICDE，pp．2-11（2005）

[2] Seltzer，M．：“Beyond Relational Databases，”Commun．ACM，Vol．51，No．7，pp．52-58（2008）

[3] 田村真司，宝珍輝尚，野宮浩揮：“遺伝的プログラミングによる排他制御プログラムの生成”，DEIM Forum 2010 E8-3 (2010)

[4] 伊庭斉志：“遺伝的プログラミング”，東京電機大学出版(1996)

[5] 都司達夫，宝珍輝尚：“データベース技術教科書”，CQ 出版社(2003)

付録　遺伝的プログラミングで使用するノード

不定変数の操作に関するノードを表 1，共有不定変数の操作に関するノードを表 2 に，条件判定ノードを表 3，その他のノードについてを表 4 に示す.

表 1　不定変数操作ノード

ノード名	引数の数	動作	種別
set_trx_val	2	現オペレーションが属するトランザクションの id=『引数 1』の不定変数の value を『引数 2』の値に変更する．宣言されていなければ value=『引数 2』の値の変数として宣言する.	非終端記号
set_data_val	2	現オペレーションが操作するデータの id=『引数 1』の不定変数の value を『引数 2』の値に変更する．宣言されていなければ value=『引数 2』の値の変数として宣言する.	非終端記号
get_trx_val	1	現オペレーションが属するトランザクションの id=『引数 1』の不定変数の value を得る．存在しなければ 0 を得る.	非終端記号
get_data_val	1	現オペレーションが操作するデータの id=『引数 1』の不定変数の value を得る．存在しなければ 0 を得る.	非終端記号
increment_trx_val	1	現オペレーションが属するトランザクションの id=『引数 1』の不定変数の value を 1 増加する．宣言されていなければ value=1 の変数として宣言する.	非終端記号
increment_data_val	1	現オペレーションが操作するデータの id=『引数 1』の不定変数の value を 1 増加する．宣言されていなければ value=1 の変数として宣言する.	非終端記号
decrement_trx_val	1	現オペレーションが属するトランザクションの id=『引数 1』の不定変数の value を 1 減少する．宣言されていなければ value=-1 の変数として宣言する.	非終端記号
decrement_data_val	1	現オペレーションが操作するデータの id=『引数 1』の不定変数の value を 1 増加する．宣言されていなければ value=-1 の変数として宣言する.	非終端記号
Delete_trx_val	1	現オペレーションが属するトランザクションの id=『引数 1』の不定変数を消去する	非終端記号
Delete_data_val	1	現オペレーションが操作するデータの id=『引数 1』の不定変数を消去する	非終端記号
DeleteAll_val	0	現オペレーションが属するトランザクションの不定変数を全て消去する.	終端記号

表 2　共有不定変数操作ノード

ノード名	引数の数	動作	種別
set_Sh_val	1	現オペレーションが属するトランザクションと操作するデータ間の共有不定変数の value を『引数 1』の値に変更する．宣言されていなければ value=『引数 1』の変数として宣言する.	非終端記号
get_Sh_val	0	現オペレーションが属するトランザクションと操作するデータ間の共有不定変数の value を得る．宣言されていなければ 0 を得る.	終端記号
increment_Sh_val	0	現オペレーションが属するトランザクションと操作するデータ間の共有不定変数の value を 1 増やす．宣言されていなければ value=1 の変数として宣言する.	終端記号
decrement_Sh_val	0	現オペレーションが属するトランザクションと操作するデータ間の共有不定変数の value を 1 減らす．宣言されていなければ value=-1 の変数として宣言する.	終端記号
Delete_Sh	0	現オペレーションが属するトランザクションと操作するデータ間の共有不定変数を消去する.	週右端記号
DeleteAll_Sh	0	現オペレーションが属するトランザクションの共有不定変数を全て消去する.	終端記号

表 3　条件判定ノード

ノード名	引数の数	動作	種別
isWRITE	1	現オペレーションが WRITE であれば『引数 1』を実行する.	非終端記号
isREAD	1	現オペレーションが READ であれば『引数 1』を実行する.	非終端記号
isLastOpe	1	現オペレーションが属するトランザクションの最後であれば『引数 1』を実行する.	非終端記号
IfEqual	3	『引数 1』と『引数 2』の値が等しければ『引数 3』を実行する.	非終端記号
NotIfEqual	3	『引数 1』と『引数 2』の値が等しくなければ『引数 3』を実行する.	非終端記号
IfSmall	4	『引数 1』と『引数 2』の値を比べて小さければ『引数 3』を，小さくなければ『引数 4』を実行する.	非終端記号
Is_Sh_val	2	現オペレーションが属するトランザクションと操作するデータ間の共有不定変数が存在すれば『引数 1』を実行し，存在しなければ『引数 2』を実行する.	非終端記号
Is_deadlock	2	現オペレーションが操作するデータの共有不定変数の連なりを見て状態 Wait のトランザクションがあれば『引数 1』を，なければ『引数 2』を実行する.	非終端記号

表 4　その他のノード

ノード名	引数の数	動作	種別
ScheLoop	1	スケジュールの先頭から順に，全てに対して引数を実行する.	非終端記号
PROG2	2	第 1，第 2 引数を順に実行する.	非終端記号
PROG3	3	第 1，第 2，第 3 引数を順に実行する.	非終端記号
ZERO	0	数値 0 を得る.	終端記号
ONE	0	数値 1 を得る.	終端記号
TWO	0	数値 2 を得る.	終端記号
get_data_id	0	現オペレーションが操作するデータの id を得る.	終端記号
get_trx_id	0	現オペレーションが属するトランザクションの id を得る.	終端記号
trxAbort	0	現オペレーションが属するトランザクションの不定変数と共有不定変数を消去して中止する.	終端記号
Sh_val_Loop	1	現オペレーションが操作するデータの共有不定変数の連なりを持つトランザクション全てに対して『引数 1』を実行する.	非終端記号
Wait	0	現オペレーションが操作するデータの共有不定変数の連なりを見て状態 Wait のもの以外を全て現オペレーションよりも前に持ってくる．また，自トランザクションの状態を Wait に変更する.	終端記号