2006年9月1日

SPEA2+(ver. 1.1.5)仕様書

同志社大学　工学部　知識工学科
三木光範
廣安知之
金美和
鈴木和徳
吉井健吾

目 次

1　はじめに

本ドキュメントは，設計変数空間における多様性を維持するメカニズムと制約条件を扱うメカニズムを組み込んだ多目的遺伝的アルゴリズム SPEA2+の使用方法と仕様についてまとめたものである．

2　プログラムの使用方法

2.1　ファイルの構成

フォルダ SPEA2+内は以下のような構成になっている．

Makefile

multiobjectiveProblem 多目的最適化のテスト問題が含まれる

- Function.h
- DEB.h
- BPF.h

spea2plus SPEA2+のプログラム

- Archive.h
- Archive.cpp
- Crossover.h
- Crossover.cpp
- Evaluation.h
- Evaluation.cpp
- GAInitializer.h
- GAInitializer.cpp
- GrayCoder.h
- GrayCoder.cpp
- Individual.h
- Individual.cpp
- MatingSelection.h
- MatingSelection.cpp
- mtrand.h
- mtrand.cpp
- Mutation.h
- Mutation.cpp
- OffspringGenerator.h
- OffspringGenerator.cpp
- OutputUnit.h
- OutputUnit.cpp

- Parameter.h
- Parameter.cpp
- Penalty.h
- Penalty.cpp
- Population.h
- Population.cpp
- RandomGenerator.h
- RandomGenerator.cpp
- Repair.h
- Repair.cpp
- SPEA2ranking.h
- SPEA2ranking.cpp
- Truncation.h
- Truncation.cpp

objarchiveData 各世代での目的関数アーカイブのデータを保存する

vararchiveData 各世代での設計変数アーカイブのデータを保存する

2.2 コンパイルとプログラムの実行

フォルダ SPEA2+の下に Makefile があるので実行する．

```
$    make
```

コンパイル後，実行ファイル SPEA2+が生成されるので実行する

```
$    ./SPEA2+
```

パラメータの設定に関しては SPEA2+/spea2plus/Parameter.cpp を編集する．パラメータについて次節で詳しく述べる．

2.3 構築パラメータ

本プログラムでは以下のパラメータを設定する必要がある．

numObjective 対象とする問題の目的関数の数

numDesign 対象とする問題の設計変数の数

populationSize 母集団サイズ（アーカイブサイズと同じ）

designLength 1 設計変数を表現するのに使用するビット数

geneLength 染色体長

numCrossover 交叉点数

maxGeneration 終了世代数

mutationRate 突然変異率

crossoverRate 交叉率

func 対象問題

finitial 初期個体の生成方法

SPEA2+/spea2plus/Parameter.cpp を編集しこれらのパラメータを指定する．設定後，make を行い実行ファイルを作成し，実行する．対象問題に関しては対象問題クラスを作成する．作成方法については次節に示す．

2.4 対象問題の設定方法

対象問題を設定するには以下の手順に従って対象問題クラスを作成する．以下の制約条件付テスト関数 DEB を例として用いる．

$$\begin{aligned} &Minimize \;\; f_1(x) = x_1 \\ &Minimize \;\; f_2(x) = (1 + x_2)/x_1 \\ &Subject\ to \;\; g_1(x) = -9x_1 - x_2 + 6 \leq 0 \\ &\qquad\qquad g_2(x) = -9x_1 + x_2 + 1 \leq 0 \\ &\qquad\qquad \begin{cases} 0.1 \leq x_1 \leq 1.0 \\ 0 \leq x_2 \leq 5 \end{cases} \end{aligned} \tag{2.1}$$

ファイルの作成

フォルダ multiobjectiveProblem 内に Function クラスの子クラスとして対象問題クラスのファイル（DEB.h）を作る．

```
#include "Function.h"
class DEB:public Function{
                ・
                ・
                ・
};
```

また Function.h に以下の文を追加する．

```
#include "DEB.h"
```

対象問題のパラメータ指定

作成したクラスのコンストラクタで親クラスである Function クラスの以下のパラメータを初期化する．

- max[]（各設計変数の上限値）
- min[]（各設計変数の下限値）
- designNum（設計変数の数）
- objectiveNum（目的関数の数）
- constraintNum（制約条件式の数）

式 (2.1) の例では以下のようになる．

```
#include "Function.h"
class DEB:public Function{
public:
        DEB(int number_design){
                designNum = 2;
                objectiveNum = 2;
                constraintNum = 2;
                max = new double[designNum];
                max[0] = 1.0;
                max[1] = 5.0;
                min = new double[designNum];
                min[0] = 0.1;
                min[1] = 0.0;
        }
} ;
```

目的関数評価メソッドの作成

親クラス Function 内の抽象メソッド $void\ \ evaluate(double*\ design, double*\ f)$ を記述する．この関数は設計変数配列 design[] を受け取り，目的関数配列 f[] を更新するものである．

式 (2.1) の例では以下のようになる．

```
        void evaluate(double* design,double* f){
                f[0] = design[0] ;
                f[1] = (1.0+design[1])/design[0] ;
        }
```

制約条件評価メソッドの作成

親クラス Function 内の抽象メソッド $void\ \ evaluateConstraint(double*\ design, double*\ g)$ を記述する．この関数は設計変数配列 design[] を受け取り，制約条件の値配列 g[] を更新するものである．制約条件のない問題の場合は "対象問題のパラメータ指定" の際に変数 constraintNum を 0 にセットし，本メソッドの中身を記述しない．

式 (2.1) の例では以下のようになる．

```
        void evaluateConstraint(double* design,double* g){
                g[0] = -9.0*design[0]-design[1]+6.0 ;
                g[1] = -9.0*design[0]+design[1]+1.0 ;
        }
```

これらのメソッドを加えて完成したものが以下の DEB.h となる．

```
#include ”Function.h”
class DEB:public Function{
public:
	DEB(int number_design){
		designNum = 2;
		objectiveNum = 2;
		constraintNum = 2;
		max = new double[designNum];
		max[0] = 1.0;
		max[1] = 5.0;
		min = new double[designNum];
		min[0] = 0.1;
		min[1] = 0.0;
	}
	void evaluate(double* design,double* f){
		f[0] = design[0] ;
		f[1] = (1.0+design[1])/design[0] ;
	}
	void evaluateConstraint(double* design,double* g){
		g[0] = -9.0*design[0]-design[1]+6.0 ;
		g[1] = -9.0*design[0]+design[1]+1.0 ;
	}
} ;
```

2.5 出力ファイルの見方

目的関数アーカイブのデータは objarchiveData フォルダに，設計変数アーカイブのデータは vararchiveData フォルダにそれぞれ世代ごとに csv として保存される．一列目から順に目的関数１，２…，設計変数１，２…のように記録され，一行につき一個体の情報を示している．Fig. 2.1 に目的関数の数が２，設計変数の数が２，個体数が５である場合の出力例を示す.

個体１	0.891738	0.204803	0.05063	0.349486
個体２	0.997081	0.005829	0.433684	0.332168
個体３	0.817927	0.330996	0.271625	0.281146
個体４	0.871927	0.239744	0.488297	0.066545
個体５	0.988317	0.02323	0.864391	0.129317
	目的関数１	目的関数２	設計変数１	設計変数２

Fig. 2.1　出力ファイルの例

2.6 入力ファイルを用いた初期化

全ての手法において設定ファイルを用いて初期個体を指定することができる．初期個体設定ファイル名は，「design_variable.ini」である (変更したい場合には，Archive.h ファイルにおける「INITIAL POPULATION

FILE」の定義を書き直す必要がある)．また Parameter.cpp で変数 finitial を 1 にする必要がある．

設定ファイル (design_variable.ini) の例を以下に示す．

```
RunCounter x1 x2 x3 x4 x5
1 -7.501370015 -8.850767853 -6.988393869 4.17963018 6.309546484
2 6.654656179 1.392214229 1.444718664 -3.603129042 3.797540378
3 3.550406736 -1.118594598 2.584324095 -3.938642765 -0.30946518
4 -0.538060596 6.945126972 -7.551823214 4.643517926 4.635494333
5 9.292352094 -6.869138526 2.147970777 7.330347315 1.373406009
6 10 10 10 10 10
```

上記の例のように，列が各変数を、行が各評価ケースを表し、１行で１つの個体初期化表現する．また，第一行目および，第一列目には必ず行名札と列名札を書く必要がある．本プログラムでは第一行目および，第一列目は入力値として読み込まない仕組みとなっている．

実際の利用の場面では、個体数とファイルのデータ行数は必ずしも一致しない場合も考えられる．本プログラムでは，その場合には

１）初期化ファイル中のデータの方が多い→ ファイル中データの先頭ｎ行を使用

２）個体数の方が多い→ファイルのデータでｎ個体目まで初期化した後、n+1 個体以降は、乱数で初期化のように初期化を行う．

＃注意： ただし，設定ファイル (design variable.ini) に書いた設計変数の数と対象とした問題の設計変数の数は必ず統一しておく必要がある．プログラムの方から設定ファイルにおける設計変数の数と実際の対象としている問題の設計変数の数との整合性についてチェックは一切行っていない．

3　クラスの説明

3.1　Archive クラス

目的関数アーカイブと設計変数アーカイブを持ち，ランキング，ペナルティ，エリート保存，メイティング選択を行う．

主なメンバをに示す．

Table 3.1　Archive クラス

メンバ変数	vector⟨Individual⟩ objarchive	目的関数アーカイブ
	vector⟨Individual⟩ vararchive	設計変数アーカイブ
メンバ関数	Population getSearchPopulation()	アーカイブから次世代の探索母集団を生成
	void updateArchive(Population& pop)	アーカイブの更新
	void exeranking(Population& pop)	探索母集団とアーカイブにランキング
	void penalty(Population& pop)	探索母集団とアーカイブにペナルティを与える
	void preserveArchive(Population& pop)	エリート保存を行う
	void truncation(vector⟨Individual⟩& fitness0)	端切りを行う
	void matingSelection(Population& pop)	メイティング選択を行う

3.2 Crossover クラス

交叉を行うクラス．Crossover クラスは抽象クラス（virtual void crossover(..)）であり，実際には子クラスである nPointCrossover クラスが n 点交叉を行う．主なメンバを Table 3.2 に示す．

Table 3.2　nPointCrossover クラス

メンバ変数	double crossoverRate	交叉率
	int numCrossover	交叉点数
メンバ関数	void crossover(Individual a,Individual b)	個体 a と個体 b を交叉する
	static Crossover& getInstance()	Crossover インスタンスを返す

3.3 Evaluation クラス

評価を行うクラス．主なメンバを Table 3.3 に示す．

Table 3.3　Evaluation クラス

メンバ関数	void evaluateOne(Individual& indiv)	設計変数値から目的関数値を得る
	void evaluateDesignValOne(Individual& indiv)	染色体から設計変数値を得る
	static Evaluation& getInstance()	Evaluation インスタンスを返す

3.4 GAInitializer クラス

個体の初期化の際，染色体を作成するクラス．GAInitializer クラスは抽象クラス（virtual void createOne(..)）であり，実際には子クラス RandomInitializer が初期染色体をつくる．RandomInitializer は初期染色体をランダムに決定する．主なメンバを Table 3.4 に示す．

Table 3.4　RandomInitilizer クラス

メンバ関数	void createOne(int* gene,int length)	染色体長 length の染色体をつくる

3.5 GrayCoder クラス

グレイコードを扱うクラス．主なメンバを Table 3.5 に示す．

Table 3.5　GrayCoder クラス

メンバ関数	int graydecode(int* grayString,int length)	bit 長 length の染色体を受けてデコードする
	void grayencode(int n,int length,int* gene)	n を bit 長 length の染色体としてコード化
	static GrayCoder& getInstance()	GrayCoder のインスタンスを返す

3.6 Individual クラス

個体を表現するクラス．主なメンバを Table 3.6 に示す．

Table 3.6　Individual クラス

メンバ変数	double* objectiveValue	目的関数の値
	double* designValue	設計変数の値
	double* constraint	制約条件の値
	int* gene	染色体
	double fitness	適合度
	int rank	ランク
	bool isFeasible	実行可能領域内なら true
	Individual* parent	親個体の情報
メンバ関数	void mutate()	突然変異する
	void evaluation()	設計変数値，目的関数値を計算し，実行可能であるか判定
	void crossover(Individual& i)	個体 i と交叉する
	void repair()	親個体に向かって引き戻る

3.7　MatingSelection クラス

アーカイブから次世代の探索母集団を選択するメイティング選択を行うクラス．MatingSelection クラスは抽象クラス（virtual void select(..)）であり，実際には子クラスである CopySelection クラスによって，メイティング選択が行われる．CopySelection クラスは単純にアーカイブから探索母集団へ個体をコピーする方式を用いる．主なメンバを Table 3.7 に示す.

Table 3.7　CopySelection

メンバ関数	void select(Archive& archive,Population& pop)	どちらかのアーカイブから個体を母集団へコピー

3.8　MTRand クラス

乱数を発生するクラス．主なメンバを Table 3.8 に示す.

Table 3.8　MTRand クラス

メンバ関数	double genrand()	0.0〜1.0 の間で double 型乱数を発生する
	int genrand_int(int n)	0 から (n-1) の間で int 型乱数を発生する
	void sgenrand(unsigned long seed)	乱数の種をセットする

3.9　Mutation クラス

突然変異をするクラス．Mutation クラスは抽象クラス（virtual void mutateOne(..)）であり，実際には子クラスの NormalMutation クラスが突然変異を行う．主なメンバを Table 3.9 に示す.

Table 3.9　NormalMutation クラス

メンバ変数	double mutationRate	突然変異率
メンバ関数	void mutateOne(Individual& indiv)	indiv を突然変異させる
	static Mutation& getInstance()	Mutation インスタンスを返す

3.10 OffspringGenerator クラス

交叉を行う際の個体同士のペアを決めるために母集団のソートを行うクラス．OffspringGenerator クラスは抽象クラス（virtual void makePairs(..)）であり，実際には子クラスとして ObjectiveNeighborhood クラス，VariableNeighborhood クラスがあり，それぞれ，目的関数空間における近傍ソート，設計変数空間における近傍ソートを担当する．主なメンバを Table 3.10 に示す．

Table 3.10　OffspringGenerator クラス

メンバ関数	void makePairs(vector⟨Individual⟩& pop)	個体配列を隣同士が交叉するように並び替える

3.11 OutputUnit クラス

探索結果をアーカイブに保存するクラス．主なメンバを Table 3.11 に示す．

Table 3.11　OutputUnit クラス

メンバ関数	void outputObjArchive(vector⟨Individual⟩& archive,int generation)	目的関数アーカイブを出力
	void outputVarArchive(vector⟨Individual⟩& archive,int generation)	設計変数アーカイブを出力

3.12 Parameter クラス

パラメータを保持するクラス．主なメンバを Table 3.12 に示す．

Table 3.12　Parameter クラス

メンバ変数	int numObjective	目的関数の数
	int numDesign	設計変数の数
	int populationSize	母集団サイズ
	int designLength	１設計変数のビット長
	int geneLength	染色体長
	int numCrossover	交叉点数
	int maxGeneration	終了世代数
	double mutationRate	突然変異率
	double crossoverRate	交叉率
	Function* func	対象問題
	int finitial	初期個体の生成方法
メンバ関数	static Parameter& getInstance()	Parameter クラスのインスタンスを返す

3.13 Penalty クラス

ペナルティのセットを行うクラス．Penalty クラスは抽象クラス（virtual void penalty(..)）であり実際は子クラスの NSGAIIpenalty によりペナルティを与える．主なメンバを Table 3.13 に示す．

Table 3.13　NSGAIIpenalty クラス

メンバ関数	void penalty(vector⟨Individual⟩& poparc	個体群にペナルティを与える

3.14　Popualtion クラス

探索母集団を表現するクラス．交叉，突然変異，評価，引き戻し操作を行う．
主なメンバを Table 3.14 に示す．

Table 3.14　Population クラス

メンバ変数	vector⟨Individual⟩ population	探索母集団
メンバ関数	void mutate()	突然変異を行う
	void crossover()	交叉を行う
	void repair()	端切りを行う
	void evaluate()	探索母集団を評価する

3.15　RandomGenerator クラス

MTRand クラスを使って乱数を発生するクラス．主なメンバを Table 3.15 に示す．

Table 3.15　RandomGeneration クラス

メンバ関数	double randomDouble()	0.0〜1.0 の間で double 型乱数を発生する
	int randomInt(int min,int max)	min から (max-1) の間で int 型乱数を発生する
	static RandomGenerator& getInstance()	RandomGenerator のインスタンスを返す

3.16　Repair クラス

交叉，突然変異により実行可能領域を外れた個体に対して引き戻しを行う．各個体は親個体の情報を持っており，その親へ向かって引き戻しが行われる．主なメンバを Table 3.16 に示す．

Table 3.16　Repair クラス

メンバ関数	void repair(Individual& indiv)	実行可能領域を外れた個体を親へ引き戻す
	static Repair& getInstance()	Repair のインスタンスを返す

3.17　SPEA2ranking クラス

ランキングを行うクラス．

Table 3.17　SPEA2ranking クラス

メンバ変数	void spea2ranking(vector⟨Individual⟩& population, vector⟨Individual⟩& objarchive, vector⟨Individual⟩& vararchive)	探索母集団，目的関数アーカイブ，設計変数アーカイブをあわせてランキング

3.18 Truncation クラス

端切りを行うクラス．アーカイブサイズを超えた vector⟨Individual⟩ を渡すと，アーカイブサイズまで個体を削減して vector⟨Individual⟩ を返す．主なメンバを Table 3.18 に示す．

Table 3.18 Truncation クラス

メンバ変数	vector⟨Distance⟩ obj	全個体同士の目的関数空間距離を保持
	vector⟨Distance⟩ var	全個体同士の設計変数空間距離を保持
メンバ関数	objtruncation(vector⟨Distance⟩ over)	目的関数空間端切り
	vartruncation(vector⟨Distance⟩ over)	設計変数空間端切り

4　補足資料

Fig. 4.1 にクラス図を示す.

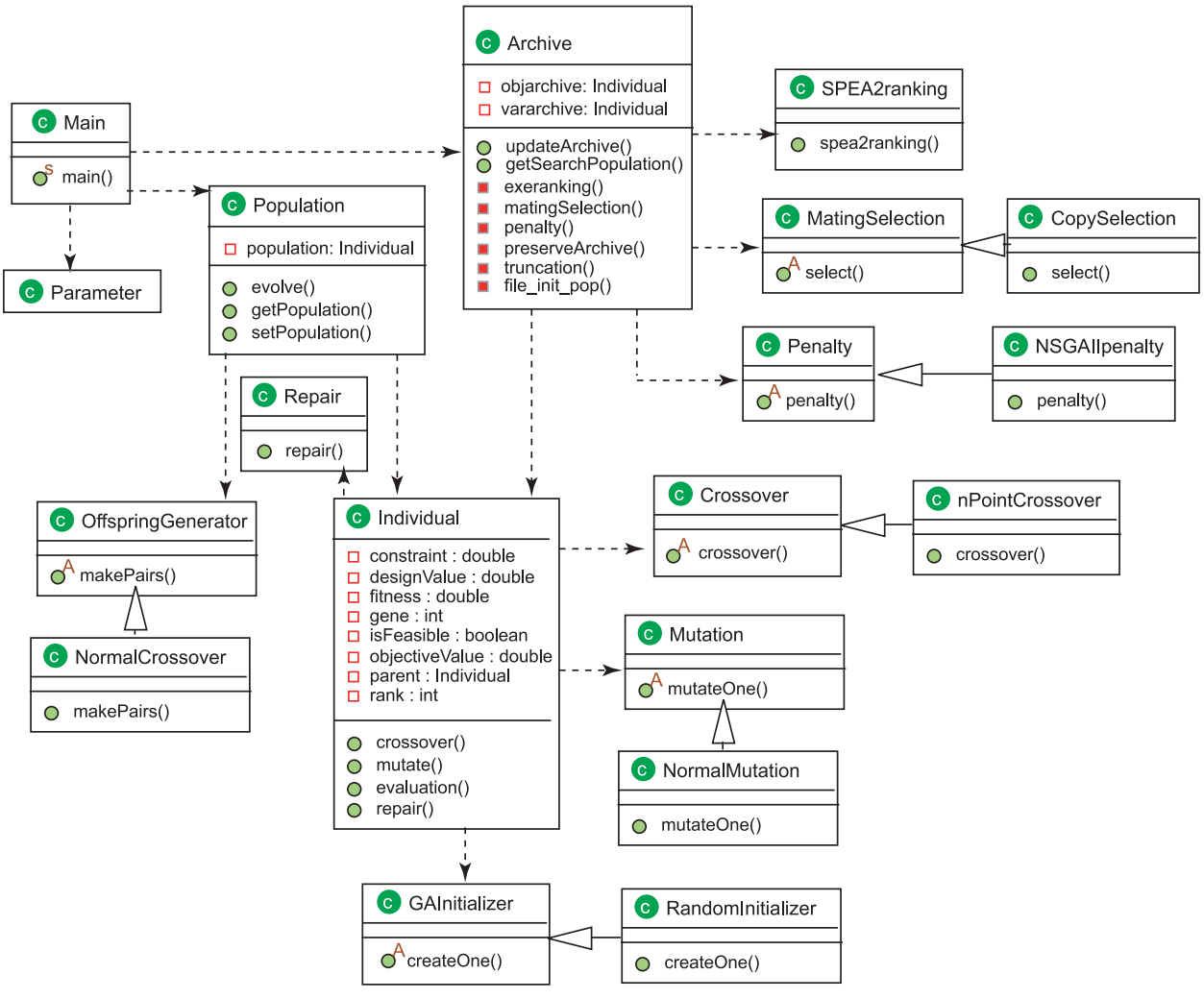

Fig. 4.1　クラス図

Fig. 4.2 にシーケンス図を示す.

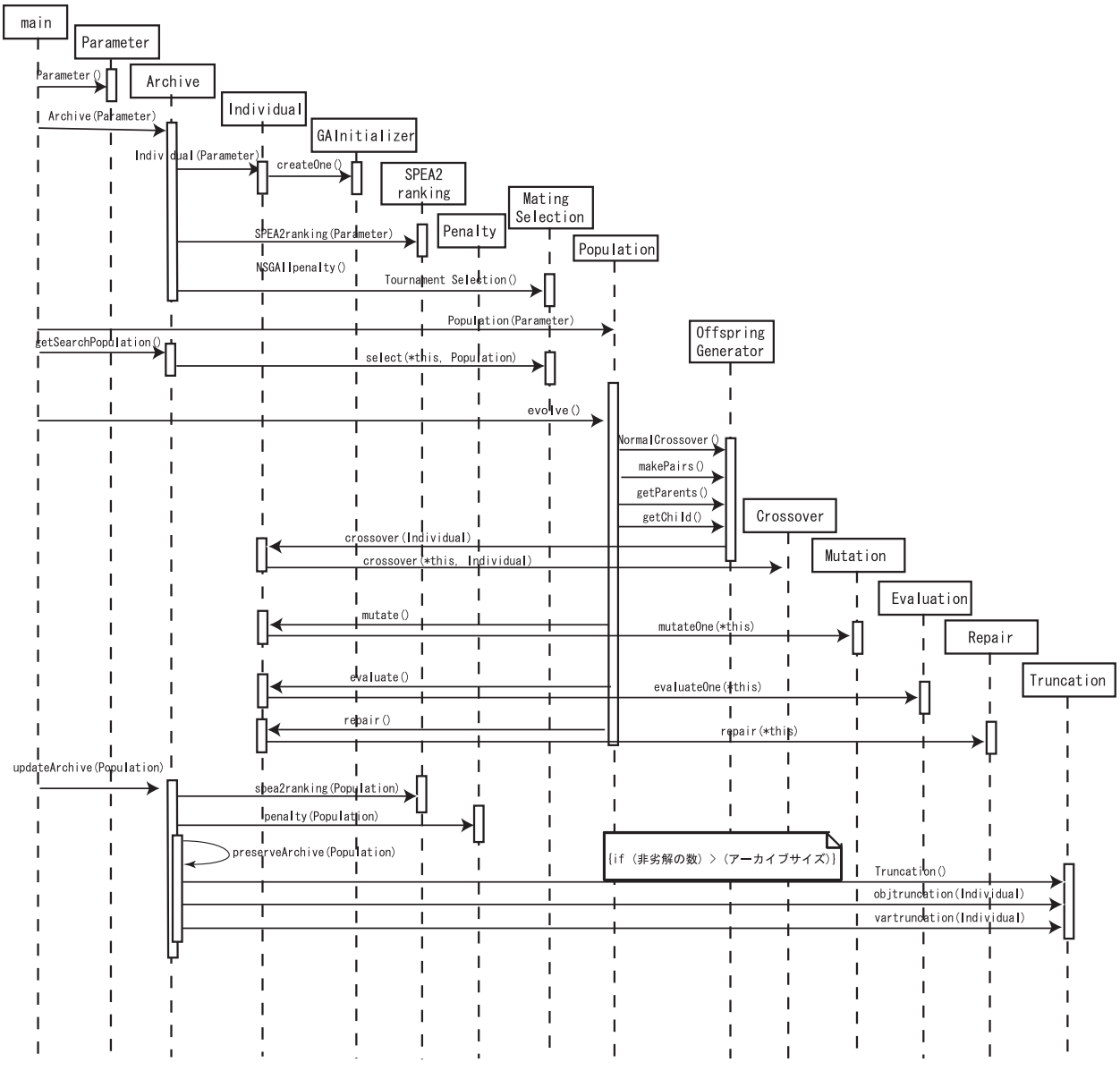

Fig. 4.2　シーケンス図

更新履歴

■ 2006.09.01 ver. 1.1.5 ■
クラス図の一部変更.

- Main.cpp : Archive クラスから getSearchPopulation() 関数により探索母集団を生成するように変更.
- Population.cpp : ga_operation() 関数を evolve() に変更.
- Archive.cpp : getSearchPopulation() 関数を作成. メイティング選択をこの関数内で使用するように変更.
- ControlCrossover.h : クラス ControlCrossover を OffspringGenerator に名称変更.
- OffspringGenerator.h : クラス NPointCrossover を NormalCrossover に名称変更

■ 2006.06.08 ver. 1.1.4 ■
引き戻しのバクを修正.
■ 2006.06.04 ver. 1.1.3 ■
初期個体をファイルから入力する機能の追加.
■ 2006.02.xx ver. 1.1.2 ■
クラス図の一部変更.
■ 2006.02.xx ver. 1.1.1 ■
クラス図の一部変更.
■ 2006.01.16 ver. 1.1.0 ■
端切りの処理における高速化.
■ 2005.04.01 ver. 1.0.0 ■
SPEA2+ 開発開始.

Copyright

■ SPEA2+:

Multi Objective Genetic Algorithm Team of Intelligent Systems Design Laboratory, Doshisha University.

■ **SPEA2+ manual:**