特別研究報告

題目

遺伝的アルゴリズムに基づいた自動プログラム修正の進化過程の可視化

指導教員

楠本 真二 教授

報告者

富田 裕也

平成 31 年 2 月 12 日

大阪大学 基礎工学部 情報科学科

平成 30 年度 特別研究報告

遺伝的アルゴリズムに基づいた自動プログラム修正の進化過程の可視化

富田 裕也

内容梗概

ソフトウェア開発において，デバッグは開発工数の半分以上を占めると言われている．そのため，デバッグの工数削減を目的として，自動プログラム修正に関する研究が活発に行われている．自動プログラム修正は修正対象のプログラムに変更を加えることで，すべてのテストケースに成功するプログラムを生成する手法である．自動プログラム修正の手法として，遺伝的アルゴリズムに基づいて修正を行うものがある．この手法は修正対象のプログラムに対して，すべてのテストケースに成功するプログラムが得られるまでプログラム文の挿入，削除および置換を繰り返し行う．遺伝的アルゴリズムに基づいた自動プログラム修正では，探索空間が非常に大きいために，解を得るまでに非常に多くの変異プログラムを生成することが多い．そのため，解を得るまでの過程，すなわち進化過程の解析が難しい．本研究では，進化過程に対する解析を行うために，進化過程を可視化するツールを実装した．さらに，実装したツールが解析の助けになるかを評価するために，遺伝的アルゴリズムに基づいた自動プログラム修正ツールである kGenProg の進化過程に対し，被験者実験を行った．実験の結果，進化過程の可視化は自動プログラム修正ツールの開発者にとって有用であることを明らかにした．

主な用語

自動プログラム修正
デバッグ
可視化
遺伝的アルゴリズム

目次

図目次

表目次

1　はじめに

ソフトウェアの信頼性および安全性の向上のためにデバッグは必要な作業である．なぜならば，ソフトウェアにおいて信頼性および安全性は非常に重要な項目であり，これらが低いソフトウェアによって，社会に多大な損害を与える可能性があるためである．デバッグでは，ソフトウェアに含まれる故障を検出し，ソフトウェアのソースコードの中から故障の原因となった記述（以降，欠陥という）を含む箇所の特定と特定した欠陥を修正する作業を行う．

ソフトウェア開発においてデバッグは多大な労力を必要とする作業であり，開発工数の半数以上を占めると言われている [1, 2]．そのため，デバッグの支援はソフトウェア開発の効率化やソフトウェアの信頼性および安全性の向上に有益である．デバッグの労力を削減するために，デバッグの支援に関する研究が数多く行われている [3, 4, 5, 6]．

デバッグの支援に関する技術として，自動プログラム修正がある．自動プログラム修正は修正対象のプログラム，そのプログラムのテスト集合および欠陥の所在から全てのテストケースに成功するプログラムを生成する手法である．

自動プログラム修正の手法の 1 つに遺伝的アルゴリズム [7] に基づいた手法がある．この手法は，修正対象のプログラムに対して遺伝的アルゴリズムを適用することで欠陥の修正を試みる．遺伝的アルゴリズムに基づいた自動プログラム修正の手法を実装したツールの 1 つに GenProg[8] がある．

遺伝的アルゴリズムは，生物の進化過程を模した最適化アルゴリズムである．優れた個体を一定数取り出す“選択”，個体を変形して新たな個体を生成する“変異”，複数の個体を組合わせて新しい個体を生成する“交叉”を行いながら解を探索する．GenProg およびその関連手法では，プログラムを生物の個体と見立てて，テストケースの実行結果から生成したプログラムの評価や変形を行う．

GenProg およびその関連手法では遺伝的アルゴリズムを用いて修正を行うため，解を得る過程（以降，進化過程という）で多くのプログラムを生成する．そのため，進化過程の解析が難しい．この問題を解決するために本研究では，遺伝的アルゴリズムに基づいた自動プログラム修正の進化過程を可視化する手法を提案した．また，実際のソフトウェア開発の過程で生じた欠陥に対して自動プログラム修正を行ったときの進化過程を可視化し，進化過程の可視化が有用なのかを定性的に評価した．

以降，2 章では本研究の前提となる技術および用語の説明を行う．3 章では提案手法について説明し，4 章では実装したツールについて述べる．5 章と 6 章では実装したツールに対する被験者実験と性能評価実験について述べる．7 章では妥当性への脅威について述べる．8 章では関連研究について述べる．最後に 9 章で本研究のまとめと今後の課題について述べる．

2 準備

本章では本研究の前提となる技術および用語について説明する．

2.1 テストケースを用いたデバッグ

一般的に，デバッグとはソフトウェアに含まれる欠陥の発見から始まり，欠陥の箇所を特定して修正し，最後に修正の成否を確認するまでを指す．したがって，デバッグを行うためにはまず欠陥の箇所，すなわちソースコードの実装や機能的要求などの誤りを特定する必要がある．

最も簡単に欠陥を特定する方法はテストケースを用いることである．テストケースとは，入力値，期待値，実値の組を指し，ある入力値に対する出力値（実値）が要求される値（期待値）と等しいか否かを判定するものである．複数のテストケースを実行し，どの入力値を与えた場合に誤った動作になるかを把握することで，欠陥を特定しやすくなる．また，十分な数のテストケースがあり，全てのテストケースの実行に成功すれば，そのソフトウェアの信頼性は高いと言える．理想的なテストケースは，ソフトウェアの全動作パターンを確認するものであるが，そのようなテストケースの作成や実行は膨大な時間や労力を必要とするため現実的ではない．したがって，実践上では到達可能な実行パスを網羅する程度に留める場合が多い．しかし，その場合でも十分なテストケースを手動で用意するには多大な労力を要するため，テストケースを自動で生成する手法が提案されている [3, 9]．

2.2 欠陥限局

欠陥限局は欠陥を含むプログラムとテスト集合から，欠陥の所在を推測する技術である．1 つでも実行に失敗したテストケースが存在するプログラムを欠陥を含むプログラムと言う．欠陥限局には，スペクトルに基づく手法 [10] がある．この手法はテストケースの成否とテストケース毎に実行したプログラム文の集合から算出した疑惑値に基づいて，欠陥原因となっている文を推測する．疑惑値とは，欠陥を含む可能性を表す数値である．失敗したテストケースが実行する文の疑惑値は大きくなり，成功したテストケースが実行する文の疑惑値は小さくなる．

2.3 自動プログラム修正

自動プログラム修正は欠陥を含むプログラムとテスト集合を入力として，修正済みのプログラムを生成する手法である．修正済みのプログラムとは，テスト集合に含まれる全てのテストケースに成功するプログラムである．自動プログラム修正に関する研究は活発に行われており，多くの手法が提案されている [8, 11, 12, 13]．

2.4 遺伝的アルゴリズム

遺伝的アルゴリズム [7] とは生物の進化過程を模した探索アルゴリズムで，解の候補を生物の個体に見立てたものである．このアルゴリズムは生物の進化過程で行われる事象をモデル化した遺伝的操作と呼ばれるいくつかの操作を繰り返し行うことで，最適解に近い個体を生成することを目的とする．

遺伝的アルゴリズムでは，まず一定数の個体をランダムに生成する．生成された個体に対して，遺伝的操作を適用することで次世代の個体群を生成する．同様にして新たな世代の個体群の生成を繰り返す．この処理を最適解が得られるか，事前に設定した世代数に到達するまで繰り返す．

遺伝的アルゴリズムの主要な操作を以下に示す．

選択 各個体の適応度を計算し，その値に基づいて個体を一定数取り出す．適応度は解へ近ければ大きい値となり，この値が大きいほど優れた個体であることを示している．

変異 個体に何かしらの変化を加え，新たな個体を生成する．加える変化は解く問題によって異なる．

交叉 2 つの個体を混ぜ合わせることで，複数の個体を生成する．変異と同様に，個体の混ぜ合わせ方は解く問題によって異なる．

2.5 GenProg

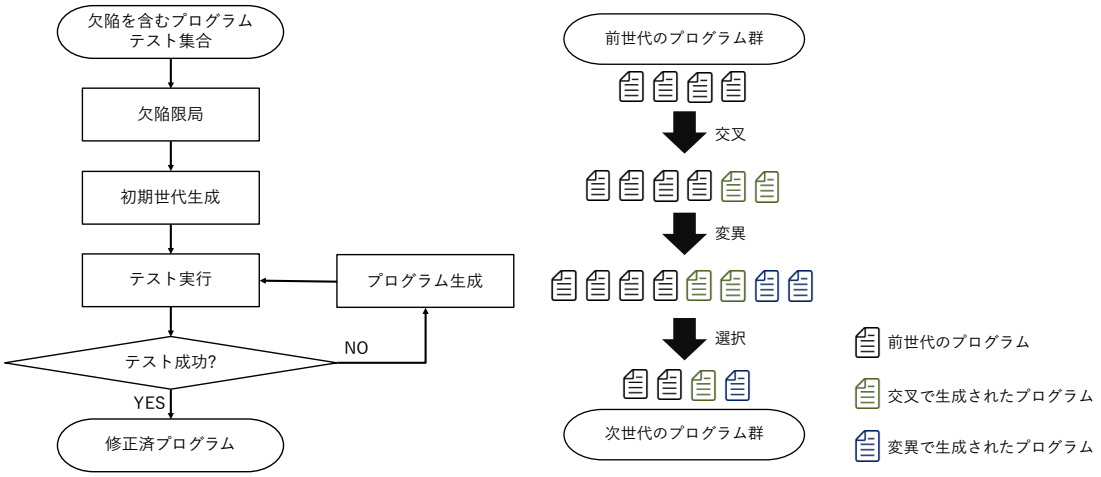
自動プログラム修正の手法の 1 つに遺伝的アルゴリズムに基づいた GenProg[8] がある．GenProg の流れを図 1（a）に示す．GenProg は欠陥限局によって欠陥箇所を推定した後，推定した欠陥箇所に変更を加えることでプログラムを生成し，修正済みのプログラムを出力する．GenProg における個体はプログラム文や文字列ではなく，変化を加えた位置と変化の対象となったプログラム文（以降，塩基という）の集合で表現される．GenProg は図 1（b）の流れでプログラムを生成し，世代を更新する．プログラムを生成するときに行われる操作の概要を以下に示す．

選択 生成されたプログラムに対してテストケースを実行する．生成されたプログラムから成功したテストケースが多いプログラムを一定数取り出す．

コピー 前世代のプログラムの一部を次世代のプログラムとして残す．

変異 前世代のプログラムの欠陥箇所に変更を加え，次世代のプログラムを生成する．加える変化はプログラム文の挿入・削除・置換のどれか 1 つである．挿入や置換で用いるプログラム文は修正対象のプログラムに含まれているプログラム文を用いる．

交叉 2 つの前世代のプログラムから，各々のプログラムに含まれる変異処理を持つプログラムを生成する．GenProg では，交叉の手法として図 2（a）に示す一点交叉を用いる．



(a) 全体の流れ　　(b) プログラム生成処理

図 1: GenProg の流れ

2.6 kGenProg

kGenProg[14] は GenProg の Java 実装である．GenProg と同様に kGenProg は図 1（a）の流れで動作する．

kGenProg では，図 2 に示す交叉の手法を使用できる．以下，交叉の手法について説明する．

一点交叉　図 2（a）のように，2 つの個体の塩基の集合をランダムに選んだ箇所で区切り，塩基を入れ換えて個体を生成する手法である．

一様交叉　図 2（b）のように，個体間で同じ位置にある塩基をランダムに選択して，個体を生成する手法である．

ランダム交叉　図 2（c）のように，個体の塩基をランダムに選択して，個体を生成する手法である．

また，交叉や変異によって偶然同じ個体を生成する可能性がある．探索済み空間の再探索を防ぐために，kGneProg では，このような個体を交叉や変異の対象にしない処理（以降，枝刈りという）を行う．

2.7 遺伝的アルゴリズムに基づいた自動プログラム修正ツールの問題点

遺伝的アルゴリズムに基づいた自動プログラム修正ツールには進化過程の解析が難しいという問題がある．これは，遺伝的アルゴリズムに基づいた自動プログラム修正ツールのアルゴリズムを改良するためには進化過程を解析する必要があるが，解を探索する過程で非常に多くの個体を生成することが多く，出力されるログが非常に大きくなりやすいためである．

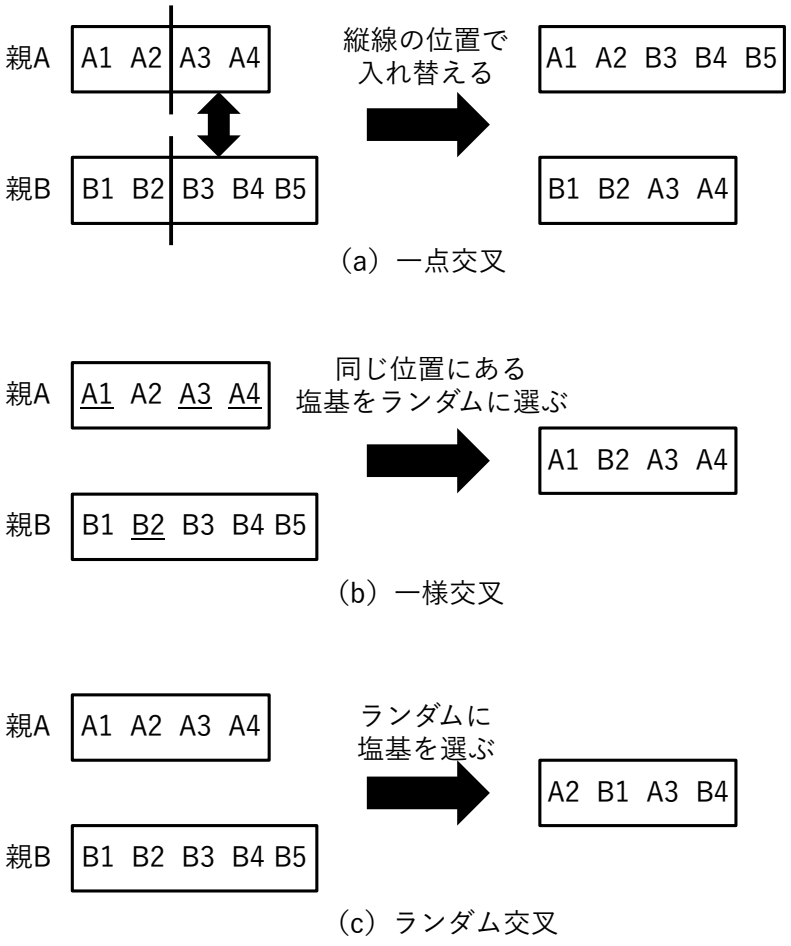


図 2: 交叉の手法

3 提案手法

3.1 概要

本研究の目的は，2.7 節で述べた進化過程の解析が難しいという問題を解決することである．本研究ではこの目的を達成するために，進化過程の可視化をする手法を提案する．可視化によって進化過程を一目で把握することができ，解析が容易にできる．

提案手法の流れを図 3 に示す．提案手法の入力は進化過程を記録したファイルである．出力は進化過程を可視化した画面である．進化過程を記録したファイルには，実行時の設定（最大世代数，各操作で生成する個体数，選択する個体の数）と進化過程で生成された全ての個体の情報が含まれている．各個体は以下の情報で構成されている．

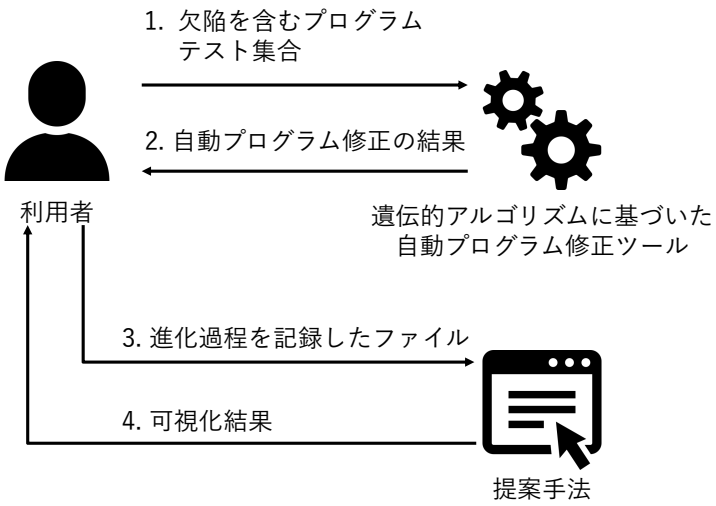
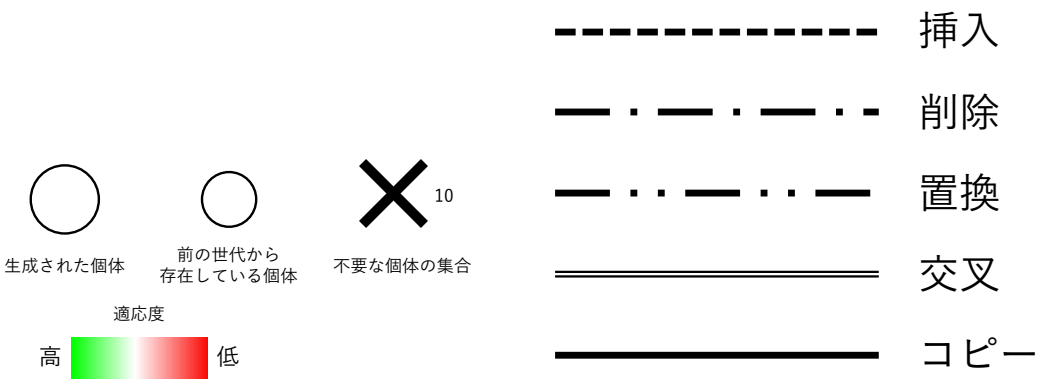


図 3: 提案手法の流れ

- ID
- 適応度
- 生成された世代
- 選択された回数
- コンパイルの成否
- 親に適用した操作の名前と親の ID
- 修正対象のプログラム（以降，初期個体という）との Unified 形式のソースコードの差分
- 実行したテストケースの数および成否

3.2 可視化の方針

提案手法では個体をノード，適用した操作をエッジに対応させた木構造で表現する．進化過程を木構造で表現しただけでは，可視化によって得られる情報量が少ないため，個体の情報をノードの色，形，大きさおよびエッジのパターンに対応させる．ノードとエッジの凡例を図 4（a），（b）に示す．円形のノードはその世代で生成された個体に対応している．円形のノードの内，一回り小さいノードは前の世代から存在している個体に対応している．円形のノードの色は適応度によって変化する．初期個体の適応度と等しい場合は白色になり，大きい場合は 1 に近いほど緑色に近づき，小さい場合は 0 に近いほど赤色に近づく．斜め十字のノードはコンパイルに失敗した，または枝刈りの対象になった個体（以降，不要な個体とする）の集合に対応している．その右の数字は不要な個体の数を表している．テスト結果



(a) ノードの凡例　　(b) エッジの凡例

図 4: ノード・エッジの凡例

やソースコードの差分のような可視化が難しい情報は可視化しない．これらの情報は利用者が必要とするときに表示する．また，木構造による可視化だけでは情報の粒度が細かく，適応度の推移のような世代毎の情報が把握しにくくなる．そのため，それらの情報は折れ線グラフや棒グラフなどのグラフを用いて可視化する．

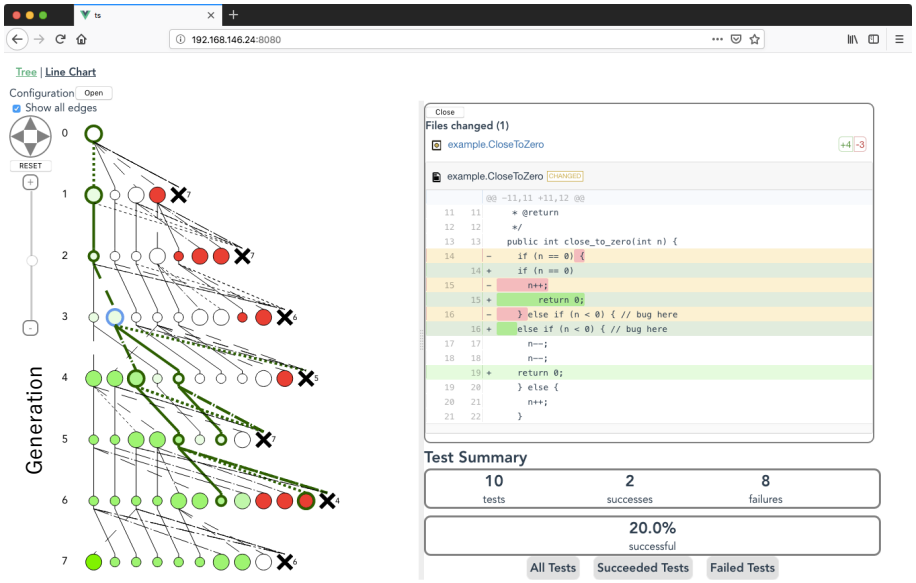


図 5: 生成された個体を表示した画面の例

4 実装

4.1 概要

本研究の提案手法を実装したツール（以降，可視化ツールという）は JavaScript で実装しており，ブラウザ上で動作する．可視化ツールは木構造による進化過程の可視化に加えて，適応度のプロットによる可視化をする．

4.2 木構造

木構造の画面の例を図 5 に示す． 3.2 節で説明したようにノードは個体，エッジは個体に適用した操作に対応している．利用者がノードをクリックすることで対応する個体のテストケースの成否・テストの成功率（図 6），初期個体とのソースコードの差分（図 7），祖先と子孫（図 8）が表示される．また，画面に収まり切らない要素を表示するために画面のズームや移動ができる．

4.3 適応度のプロット

この可視化は図 9 のような世代毎の適応度の最大・平均・最小値をプロットをした折れ線グラフを表示する．この可視化によって利用者は実行時のパラメータやアルゴリズムを変えたときの探索性能の簡単な比較ができる．

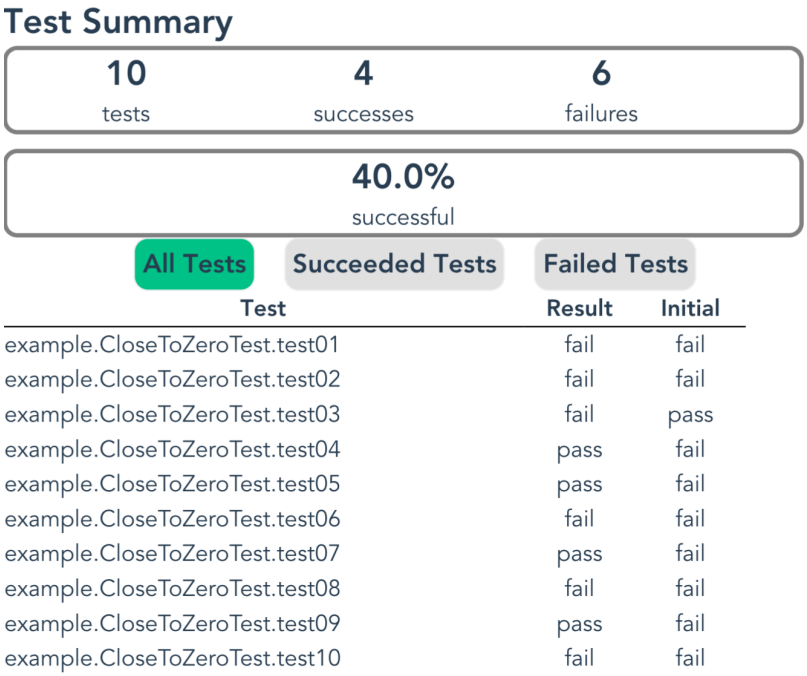


図 6: テストの通過率の表示例

図 7: 差分の表示例

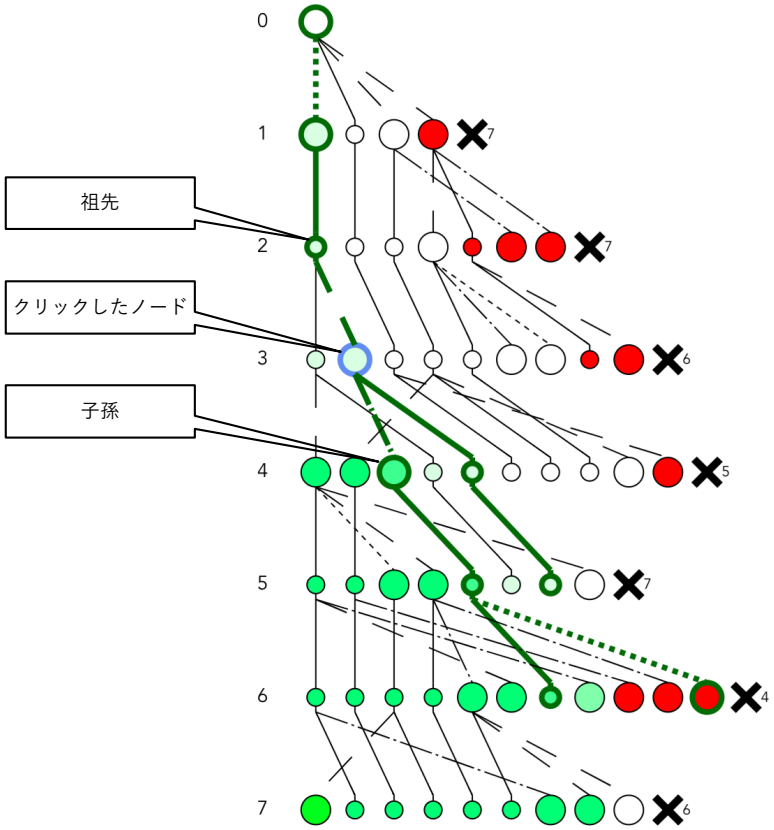


図 8: 子孫および祖先の表示例

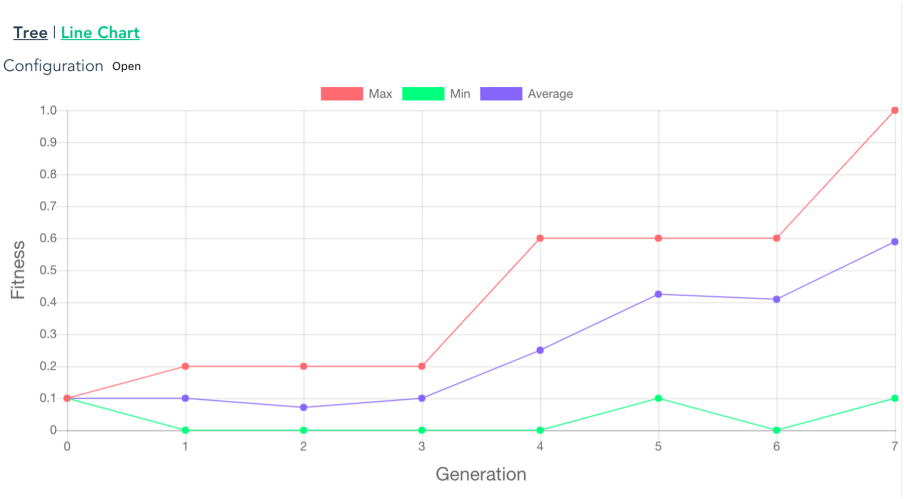


図 9: 適応度をプロットした折れ線グラフの例

5　被験者実験

5.1　実験目的

実際の欠陥に対して自動プログラミング修正の進化過程を可視化した結果を用いて，提案手法が有用かどうか評価するために被験者実験を行った．

5.2　被験者

本実験の被験者は大阪大学大学院情報科学研究科に所属する教員 2 名，同研究科に所属する修士の学生 5 名，大阪大学基礎工学部情報科学科に所属する学部生 1 名の計 8 名である．また，全ての被験者は kGenProg の開発者である．

5.3　実験対象

本実験の可視化対象は Apache Commons Math[15] というオープンソースソフトウェア（以降，OSS という）の開発過程で生じた欠陥に対して，kGenProg を適用したときの進化過程である．これらの欠陥は Defects4j[16] という欠陥のデータセットより取得した．表 1 に示された欠陥に対して，kGenProg の以下の設定を変化させたときの進化過程を用意し，それらを対象に実験を行なった．

- 最大世代数
- 変異で生成する個体の数
- 交叉の手法（ランダム交叉，一様交叉，一点交叉）

用意した題材は以下の 3 種類である．

題材 A　実行に失敗したテストが少ない欠陥に対し，最大世代数および変異で生成する個体の数のみを変化させたときの進化過程．

題材 B　実行に失敗したテストが多い欠陥に対し，最大世代数および変異で生成する個体の数のみを変化させたときの進化過程．

題材 C　実行に失敗したテストが少ない欠陥に対し，交叉の手法のみを変化させたときの進化過程．

5.4　実験手順

被験者実験は以下の流れで実施した．

1. 可視化ツールの説明：可視化ツールの使い方を被験者に説明する．

2. 操作の練習：簡単なプログラムの欠陥を kGenProg で修正したときの可視化結果に対して，可視化ツールを操作してもらい，ツールの使い方を習得してもらう．
3. 議論：被験者に 5.3 節で述べた題材に対して可視化から得られたことについて議論してもらう．各議論は意見が出なくなるまで行う．

5.5 実験結果

各題材についての議論から得られたことを以下の 3 つに分類した．

- 遺伝的アルゴリズムの振る舞い
- kGenProg のアルゴリズムの改良案
- kGenProg の欠陥

5.5.1 遺伝的アルゴリズムの振る舞い

以下に遺伝的アルゴリズムの振る舞いに関する意見をまとめた．

交叉の親の選択方法 題材 B ではテスト結果が異なる個体を用いた交叉によって，解が得られた．そのため，ランダムに交叉の親を選ぶのではなく，テスト結果が大きく異なる 2 つの個体を交叉の親に選んだ方が交叉の性能が向上するのではないのかという意見が得られた．

実行時の設定の調整 題材 B では図 10 のような無意味なプログラム文が挿入された解が得られた．世代を重ねると個体に少しずつ変更が加わるが，世代を重ねても適応度が必ずしも高くなるわけではない．そのため，各世代で生成する個体の数を少なくすると世代を重ねてしまい，生成される

表 1: 実験対象にした欠陥の一覧

題材	欠陥 ID	失敗した テストケースの数	最大世代数	変異で生成する 個体の数	交叉で生成する 個体の数	選択する 個体の数	交叉の手法
A	Math95	1	300	10	0	5	-
	Math95	1	40	75	0	5	-
B	Math43	6	200	10	5	5	ランダム交叉
	Math43	6	40	55	5	5	ランダム交叉
C	Math2	1	400	2	40	20	ランダム交叉
	Math2	1	400	2	40	20	一様交叉
	Math2	1	400	2	40	20	一点交叉

```
 public Mean() {
   incMoment = true;
+  final int prime = 31;
   moment = new FirstMoment();
 }
```

図 10: 無意味なプログラム文が挿入された解のソースコードの一部

個体に多くの変更が加わった結果，無意味なプログラム文が挿入された解が得られたと被験者は考えた．一方で，各世代で生成する個体の数を多くすれば，無意味なプログラムが挿入されていない解が得られた．このことから，各世代で生成する個体の数を多くすることで生成される個体に加わる変更が少なくなり，可読性の高い解が得られるのではないかという意見が得られた．

5.5.2 kGenProg のアルゴリズムの改良案

以下に kGenProg のアルゴリズムの改良案をまとめた．

挿入の改良 変数を使用したプログラム文を挿入するときに，その変数の宣言文も同時に挿入する操作を実装すれば，効率的に解を探索できるのではないのかという改良案が得られた．題材 A では図 11 のように変数を使用する文が挿入されることが多く，他の操作と比べて挿入は不要な個体を多く生成している．そのため，挿入によって生成された個体は未定義変数を使用したプログラム文が挿入される可能性が高く，コンパイルが失敗しやすいと被験者が考えた結果得られた改良案である．

選択の改良 不要な個体を多く生成した個体を選択しないという改良案が得られた．題材 B を可視化した結果から不要な個体を多く生成した個体からはまだ発見されていない個体が生成されにくい．そのため，世代が進むにつれて，不要な個体の数が増え，個体の多様性が乏しくなると被験者が考えた結果得られた改良案である．

適応度の計算方法の改良 題材 A を可視化した結果から遺伝的アルゴリズムの利点を活かした動作をするために，適応度の計算を改良する案が得られた．kGenProg では適応度をテストの成功率と定義している．しかし，実際のソフトウェアでは，失敗するテストケースの数が少ない場合が多いため，解と入力されたプログラムの間を適応度を用いて適切に表現することができず，個体が徐々に改善するという遺伝的アルゴリズムの利点を活かした動作をしていない．

```
 public SumOfLogs() {
   value = 0d;
   n = 0;
+  var = evaluate(values, m, begin, length);
 }
```

図 11: 変数を使用したプログラム文が挿入された例

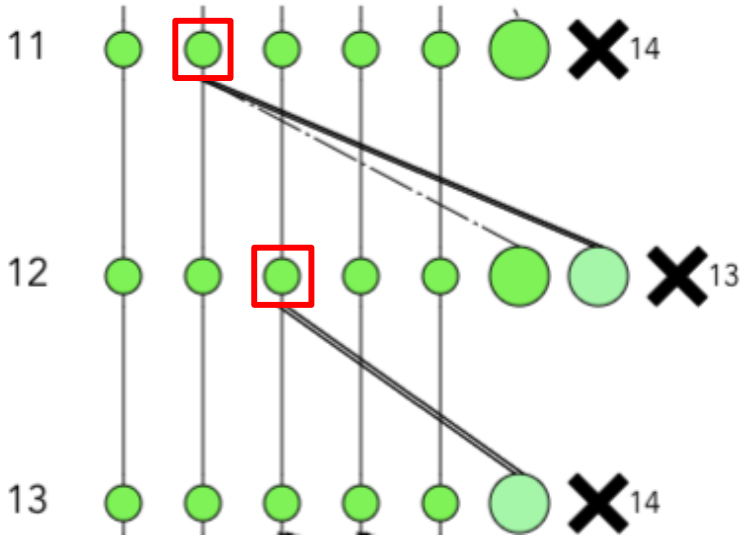


図 12: 交叉に用いる個体が 1 つだけの例

5.5.3 kGenProg の欠陥

以下に実験中に発見した kGenProg の欠陥をまとめた．

選択の欠陥 個体を選択するときに，適応度が同じならば，意図せず古い世代の個体が優先的に選択されるという欠陥を発見した．

交叉の欠陥 交叉は 2 つ以上の個体を親とすべきであるが，図 12 中の赤線で囲んだ部分のような交叉に用いる親が 1 つだけである欠陥を発見した．

5.6 考察

5.6.1 遺伝的アルゴリズムの振る舞いに関する考察

交叉の親の選択方法 可視化ツールは行った操作ごとにエッジのパターンを変えるため，行われた操作が一目でわかるようになる．そのため，被験者が交叉によって得られた解の親に当たる個体のテスト結果を比べた結果，テスト結果が異なる 2 つの個体を交叉の親に選んだ方が交叉の性能が向上するのではないのかという意見が得られたと考えられる．

実行時の設定の調整 可視化ツールはクリックした個体と入力プログラムとのソースコードの差分を表示できるため，操作によって変化した部分が把握しやすくなる．そのため，被験者が世代を重ねて得られた解には無駄なコードが挿入されていることを発見し，1 世代に生成する個体数を増やした方が可読性の高い解が得られるのではないのかという意見が得られたと考えられる．

5.6.2 kGenProg のアルゴリズムの改良案に関する考察

挿入の改良 可視化ツールはエッジのパターンで行った操作を表現できることに加え，クリックした個体と入力プログラムとのソースコードの差分の表示ができるため，操作によって変化した部分を把握しやすくなる．そのため，被験者が挿入によって生成された個体のソースコードの差分を表示すると，変数を使用した文が多く表示されていたことを発見し，挿入の改良案が得られたと考えられる．

選択の改良 可視化ツールは選択した個体を一回り小さいノードで表現するため，選択された個体が目立つようになる．そのため，被験者が選択された個体から生成された個体に注目した結果，複数回選択された個体からはまだ発見されていない個体が生成されにくいことを発見し，選択の改良案が得られたと考えられる．

適応度の計算方法の改良 可視化ツールはノードの色で個体の適応度を表現できることに加え，適応度のプロットもできるため，適応度の変化を把握しやすくなる．そのため，題材 A の可視化結果から被験者がプロットした適応度の最大値が解を得るまで変化しないことと多くのノードの色が白色であったことを発見し，適応度の計算方法の改良案が得られたと考えられる．

5.6.3 kGenProg の欠陥に関する考察

選択の欠陥 可視化ツールは選択された個体を生成された個体よりも一回り小さくするため，選択された個体が目立つようになる．そのため，被験者が kGenProg の選択のアルゴリズムに関する欠陥を発見できたと考えられる．

交叉の欠陥　可視化ツールは操作をエッジのパターンで表現することで，どの操作が行われたか一目でわかるようになる．そのため，被験者が kGenProg の交叉に関する欠陥を発見できたと考えられる．

5.6.4　全体

kGenProg のアルゴリズムの改良案や遺伝的アルゴリズムの振る舞いに関する意見が得られたことから，遺伝的アルゴリズムを基づいた自動プログラム修正の進化過程の可視化は開発者にとって有用であると結論づけることができる．特に，kGenProg のアルゴリズムの改良案が得られたことから，進化過程の可視化は遺伝的アルゴリズムに基づいた自動プログラム修正ツールの改良に繋がると言える．

6 性能評価実験

6.1 実験目的

本実験の目的は可視化ツールの実行性能が実用的か調査することである．

6.2 実験設計

Defects4j に記録されている欠陥に対して，表 2 に示す設定で kGenProg を適用したときに得られた進化過程を使用する．対象とした OSS は，Joda-Time[17]，JFreeChart[18]，Apache Commons Math である．計測対象を以下に示す．

- 初期化時間：進化過程を記録したファイルの読込を開始してから画面が表示されるまでの時間
- 応答時間：ズームあるいは移動をしてから再び画面が描画されるまでの時間

いずれの時間も 5 回ずつ実行し，その平均を計測値とする．また，初期化時間の計測時には以下の時間も計測した．

- ファイル読込・解析時間：入力ファイルの読込および解析にかかる時間
- 座標・レイアウト計算時間：入力ファイルの読込および解析が終わってから，ノードやエッジの座標およびレイアウトが決定されるまでの時間
- 描画時間：ノードやエッジの座標およびレイアウトが決定されてから，画面に表示されるまでの時間

実験で用いた計算機の性能およびブラウザを表 3 に示す．

6.3 性能目標

利用者が違和感なく操作ができていると感じることができる時間は最長で 100 ミリ秒，注意を維持できる時間は最長で 10 秒と言われている [19]．これらの値に基づき，可視化ツールの性能目標として初期化時間を 10 秒，応答時間を 100 ミリ秒に設定した．

表 2: kGenProg の設定

変異で生成する個体の数	交叉で生成する個体の数	選択する個体の数	交叉の手法	動作させる時間（分）
10	5	5	ランダム交叉	30

6.4 実験結果

実験対象にした欠陥に対して表 2 の設定で kGenProg を適用した結果を表 4 に示す．初期化時間および応答時間の計測結果を図 13，14 に示す．グラフの縦軸はミリ秒単位の時間，横軸は実験対象にした欠陥 ID，赤色の線は目標値を示している．図 13 から初期化時間は Time1 以外では目標値を達成できていることがわかる．図 14 から Math2 以外の応答時間は目標値を大幅に上回っていることがわかる．以上のことから可視化ツールはズームや移動の実行性能が悪いため，応答時間の改善が必要であることがわかる．

6.5 考察

初期化時間に注目すると，描画時間が初期化時間の 70% 以上を占めている．その原因として可視化ツールはノードやエッジを SVG を用いて表現しており，初期化時に全ての SVG 要素の構築および描画をしていることが考えられる．そのため，画面外の SVG 要素の構築を行わないように可視化ツールを改良すれば，大幅に初期化時間を削減できると考えられる．さらに，同じ理由でズーム・移動の応答時間も改善できると考えられる．

表 3: 計測に用いた計算機の性能およびブラウザ

OS	Windows10
CPU	1.5GHz
コア数	12 コア
GPU	NVIDA Quadro K2200
メモリ	32GB
ブラウザ	Google Chrome

表 4: kGenProg を適用した結果

欠陥 ID	到達世代数	生成した個体の数
Time1	464	6,885
Chart1	186	2,775
Math2	49	930

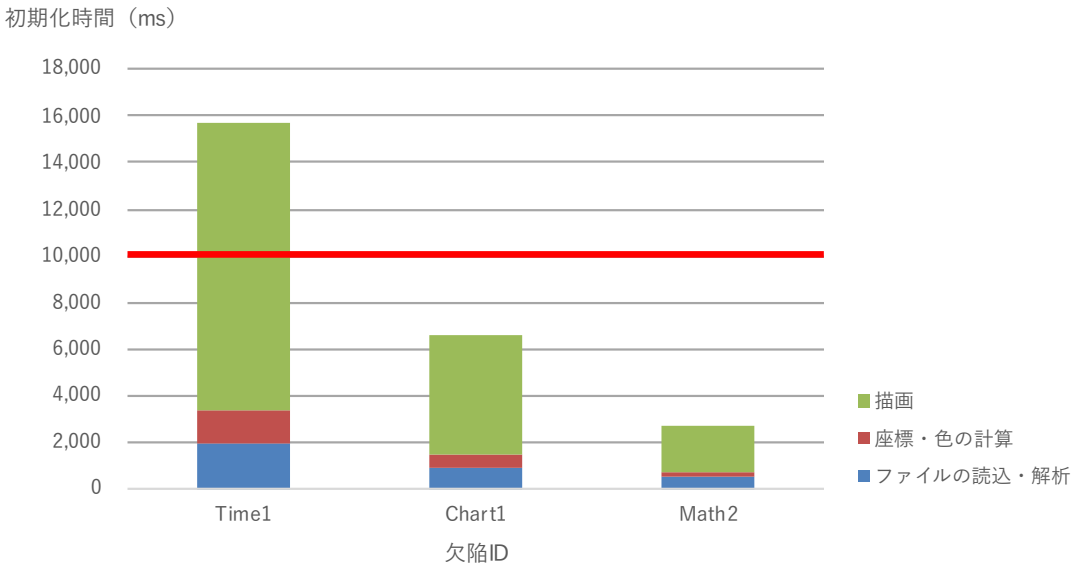


図 13: 初期化時間の計測結果

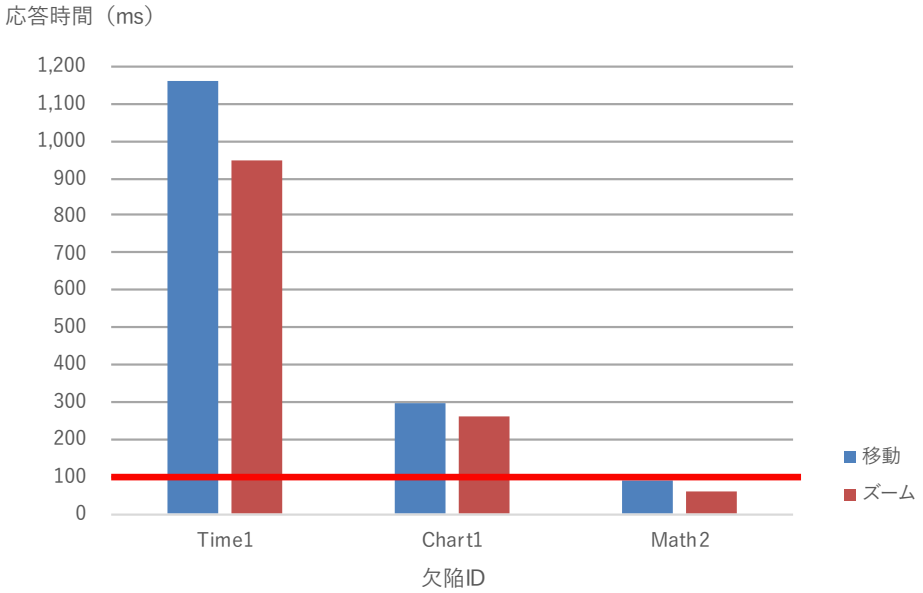


図 14: 応答時間の計測結果

7　妥当性への脅威

7.1　被験者実験

実験対象ソフトウェアは Apache Commons Math のみである．ソフトウェアの欠陥によって，進化過程も変わるため，他のソフトウェアで実験を行えば，異なる意見が得られる可能性がある．

7.2　性能評価実験

実験対象にするソフトフェアや実行時のパラメータによって，差分のデータが大きく変化するため，読み込み時間が変化する可能性がある．

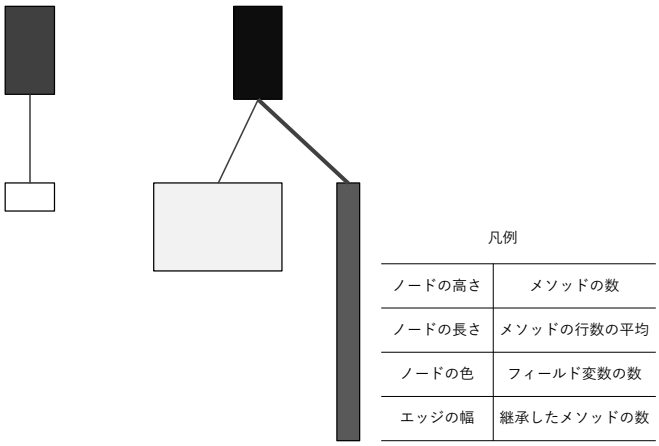


図 15: Polymetric Views の例

8 関連研究

本章では，本研究に関連する既存研究について説明する．

8.1 Polymetric Views

Lanza らは，オブジェクト指向言語で書かれたソフトウェアの構造を理解するための手法として，Polymetric Views[20] を提案した．Polymetric Views では，図 15 のようにソフトウェア中のクラスをノード，クラス間の関係をエッジで表現する．さらに，コードの行数，メソッドの数などのクラスに関するメトリクスをノードの色，座標，大きさに対応づける．エッジの幅や色もノードと同様に，子クラスの数などのクラス間の関係に関するメトリクスに対応づける．Lanza らは，Polymetric Views を用いてソフトウェアの構造を可視化することによって，開発者がリバースエンジニアリングを行うときの助けになり得ると報告している．

8.2 GAVEL

Hart らは，遺伝的アルゴリズムの過程を解析するためのツールとして GAVEL[21] を開発した．GAVEL では解となった個体の祖先のみを可視化の対象にしており，以下に示す情報を可視化できる．

- 個体に行われた操作
- 個体の祖先と子孫

さらに，グラフのプロットに必要な以下の情報を出力できる．

- 適応度の最大・平均・最小値
- 生成された個体に対して解の祖先に当たる個体が占める割合
- 新しく見つけた個体の割合

9　あとがき

本研究では，遺伝的アルゴリズムに基づいた自動プログラム修正の可視化を提案した．提案手法が有用であるか確認するために被験者実験を行った．被験者実験によって，アルゴリズムの改良案や遺伝的アルゴリズムに関する意見が得られた．よって，進化の過程の可視化は開発者にとって有用であると結論づけることができる

今後の課題として，可視化する情報を増やすことが考えられる．例えば，以下に示す拡張が考えられる．

操作サマリ　操作別にその操作が行われた回数や生成した不要な個体の数を表示する．この拡張によってどの操作が有用かどうか理解しやすくなると考えられる．

個体とその親との差分　利用者が選択した個体とその親に当たる個体との差分，すなわち操作によってプログラムが変更された部分を表示する．この拡張によって行われた操作についてより理解しやすくなると考えられる．

進化の履歴　利用者が選択した個体に対して適用された操作およびプログラムの変更箇所を一覧で表示する．この拡張からどの操作によって解が得られたかがより理解しやすくなると考えられる．

現在，可視化ツールは kGneProg の進化過程しか可視化できない．他の遺伝的アルゴリズムに基づいた自動プログラミング修正ツールの進化過程を可視化できるようにすることも今後の課題である．

また，可視化ツールの性能を上げることも今後の課題として考えられる．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，励まして頂きました楠本真二教授に，心より感謝申し上げます．

本研究に関して，貴重で有益な助言をして頂きました，肥後芳樹准教授に深く感謝申し上げます．

本研究の全過程を通し，研究に対する方針や実現方法など，終始丁寧かつ熱心なご指導を賜りました柗本真佑助教に，心より感謝申し上げます．

本研究を進めるにあたり，様々な形で励まし，協力を頂きました楠本研究室の皆様のご協力に心より感謝申し上げます．

参考文献

[1] Graham Carver Paul Cheak Tom Britton, Lisa Jeng and Tomer Katzenellenbogen. Reversible Debugging Software: Quantify the time and cost saved using reversible debuggers , 2013.

[2] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification . *IBM Systems Journal*, Vol. 41, pp. 4–12, 2002.

[3] L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs . *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, pp. 215–222, Sep. 1976.

[4] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-Directed Random Test Generation . In *Proceedings of the 29th International Conference on Software Engineering*, ICSE 2007, pp. 75–84, 2007.

[5] Daniele Zuddas, Wei Jin, Fabrizio Pastore, Leonardo Mariani, and Alessandro Orso. MIMIC: Locating and understanding bugs by analyzing mimicked executions . *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pp. 815–825, Sep. 2014.

[6] James A. Jones and Mary Jean Harrold. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique . In *Proceedings of the 20th ACM/IEEE International Conference on Automated Software Engineering*, pp. 273–282, 2005.

[7] K. F. Man, K. S. Tang, and S. Kwong. Genetic algorithms: concepts and applications [in engineering design] . *IEEE Transactions on Industrial Electronics*, Vol. 43, No. 5, pp. 519–534, Oct. 1996.

[8] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each . In *Proceedings of the 34th International Conference on Software Engineering*, pp. 3–13, 2012.

[9] Rupak Majumdar and Koushik Sen. Hybrid Concolic Testing . In *Proceedings of the 29th International Conference on Software Engineering*, pp. 416–426, 2007.

[10] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A Practical Evaluation of Spectrum-based Fault Localization . *J. Syst. Softw.*, Vol. 82, No. 11, pp. 1780–1792, Nov. 2009.

[11] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The Strength of Random Search on Automated Program Repair . In *Proceedings of the 36th International Conference on Software Engineering*, pp. 254–265, 2014.

[12] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program Repair via Semantic Analysis . In *Proceedings of the 2013 International Conference on Software Engineering*, pp. 772–781, 2013.

[13] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs . *IEEE Transactions on Software Engineering*, Vol. 43, No. 1, pp. 34–55, Jan. 2017.

[14] Yoshiki Higo, Shinsuke Matsumoto, Ryo Arima, Akito Tanikado, Keigo Naitou, Junnosuke Matsumoto, Yuya Tomida, and Shinji Kusumoto. kGenProg: A High-performance, High-extensibility and High-portability APR System . In *the 25th Asia-Pacific Software Engineering Conference*, pp. 697–698, Dec. 2018.

[15] Apache Commons Math . `http://commons.apache.org/proper/commons-math/`.

[16] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs . In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 437–440, 2014.

[17] Joda-Time . `https://www.joda.org/joda-time/`.

[18] JFreeChart . `http://www.jfree.org/jfreechart/`.

[19] P. Pirolli. Powers of 10: Modeling Complex Information-Seeking Systems at Multiple Scales . *Computer*, Vol. 42, No. 3, pp. 33–40, Mar. 2009.

[20] M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering . *IEEE Transactions on Software Engineering*, Vol. 29, No. 9, pp. 782–795, Sep. 2003.

[21] E. Hart and P. Ross. GAVEL - a new tool for genetic algorithm visualization . *IEEE Transactions on Evolutionary Computation*, Vol. 5, No. 4, pp. 335–348, Aug. 2001.