社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

変更コード片の動的切替による自動プログラム修正のビルド時間削減の試み

古藤 寛大† 肥後 芳樹† 柗本 真佑† 楠本 真二†

† 大阪大学大学院情報科学研究科
大阪府吹田市山田丘 1-5
E-mail: †{k-kotou,higo,shinsuke,kusumoto}@ist.osaka-u.ac.jp

あらまし 欠陥を含むプログラムをコンピュータが自動的に修正する技術として自動プログラム修正がある．この技術のうち，対象プログラムの変更と評価を繰り返すことにより欠陥を含まないプログラムの生成を行う手法がある．この手法では，変更時に対象プログラムから修正するコード片を限局したのちそのコード片のみ変更したプログラムを生成し，評価時に生成プログラムに対しビルドとテストの実行をする．しかし，この手法は評価時のビルド処理に長い時間を要する．なぜならば，各コード片につき 1 つのプログラムを生成し，生成プログラムごとにビルドを行うため，変更するコード片が多いほどビルド実行を行わなければならないからである．そこで，我々の研究は自動プログラム修正におけるビルド時間の削減を目指す．具体的には，テスト実行時に変更コード片の反映を行うかを動的に切り替えられるプログラムを，すべての変更コード片に基づき生成及びビルドする．そしてテスト実行時に動的にコード片を切り替えることにより上記の問題点を克服する．この手法により，変更するコード片の数にかかわらず生成されるプログラムは１つだけであり，ビルドの回数を減らすことができる．実験の結果，従来手法と比較してビルド時間が削減され，それによって自動プログラム修正全体の時間を削減することに成功した．

キーワード 自動プログラム修正，ビルド時間，遺伝的アルゴリズム

1. はじめに

ソフトウェア開発において，プログラムから故障の原因となる記述 (以降，欠陥) の発生を回避するのは困難である．ソフトウェア開発において欠陥の特定と修正は多大な労力を必要とする作業であり，開発工数の大半を占めるといわれている [1].

近年，この問題を解決する技術として自動プログラム修正が盛んに研究されている [2]．自動プログラム修正とは，欠陥を含むプログラムとテストスイートを入力として受け取り，テストを全て通過するプログラムをコンピュータが自動で生成する技術である．この自動プログラム修正のうち，生成と評価に基づく手法がある [3]．この手法は，修正対象プログラムの実行情報を用いて欠陥限局し，その箇所に変更を加えたプログラムを評価して，評価値を基に新たなプログラムを変更する処理を繰り返してプログラム修正を行う．この手法は，評価の際にビルドとテストの実行を行うことによって欠陥の修正の成否を確認する．

しかし，この手法では自動プログラム修正における評価時のビルド実行に要する時間が全体の 22% 以上を占めるという問題点がある．その原因として，コード片を変更したプログラムを大量に生成し，生成された全てのプログラムに対しビルドを実行することが考えられる [4]．そのため，このオーバーヘッドにより自動プログラム修正の実行に長い時間を要する．

我々は上記の自動プログラム修正における問題点を解決するため，修正対象プログラムの変更コード片を生成先のプログラム上で動的に切り替える手法を提案する．この手法はプログラム生成において，プログラム上で実行する変更コード片を条件分岐により任意に決めることが可能なプログラムを生成する．テスト実行時に変更コード片が欠陥修正に対し有効か確認することで従来よりもビルド回数を減らしつつ，同一の評価機能を得られる．また，変更コード片を短時間で多く検証できれば，より多くの解候補を検証することが可能となり，複雑な欠陥を有するプロジェクトに対して修正案を多く検証できる．提案手法の評価のために実際の欠陥を含有するプロジェクトに対して従来手法と提案手法それぞれを適用した．その結果，提案手法のビルド時間が従来手法と比べて 89% から 46% に削減されたことを確認した．

2. 研究動機

プログラムを変更し，欠陥修正の成否を評価する自動プログラム修正手法の抱える問題点として，評価時に実行するビルドとテストに長い時間を要することがある [4]．自動プログラム修正では修正対象プログラムから，欠陥限局で得られた情報に基づいて選択された各コード片につき，そのコード片のみ部分的に変更したプログラムを生成する．そして生成プログラムごとにビルドを行う．対象プログラムから選択する変更コード片が多いほど，ビルド処理回数は多くなる．そのため，自動プロ

グラム修正の処理全体においてビルド実行が処理時間の面から支配的になりうる．

自動プログラム修正ツール kGenProg [5] を後述する小規模プロジェクト CloseToZero 及び Defects4J が保持する大規模プロジェクト Apache Commons Math の両方に実行した際の全体における実行時間の内訳を表 1 にて示す．このとき処理全体においてビルド時間が 22% 以上を占める結果となった．

このことから，自動プログラム修正においてビルド実行が全体の処理において大きな比重を占めていることが分かる．また，ビルド処理はプロジェクトの規模が大きくなるほど処理に時間を要するため，実際の開発において自動プログラム修正を利用する場合にはビルドだけで長い時間を要する．自動プログラム修正のビルド時間を削減することは，自動プログラム修正の実行時間の削減に大きく寄与する．プログラム修正の時間が削減されることにより，開発工程で発生するデバッグ作業が効率化される．すなわち，ビルド時間削減の研究はソフトウェア開発の分野に貢献しうる．

3. 準　　備

3.1 自動プログラム修正

3.1.1 概　　要

自動プログラム修正とは，欠陥を含むプログラムをコンピュータが自動的に修正する技術である．この技術のうち，対象プログラムの変更と評価を繰り返すことにより欠陥を含まないプログラムに修正する手法がある．この手法では以下の手順を繰り返すことによってテストを全て通過するプログラムの生成を行う．手法のフローチャートを図 1 に示す．

限局　修正対象プログラム中の欠陥を含む箇所を推測する．

生成　修正対象プログラムの中から変更するコード片を**限局**により選択し，そのコード片を挿入，削除，置換といった操作で変更したプログラムを生成する．

評価　**生成**で新たに作られたプログラムに対しビルドとテストを実行し，適切にプログラムが修正されたかを検証する．

3.1.2 欠 陥 限 局

デバッグを支援する手法として，欠陥限局の研究が行われている．欠陥限局とは，入力として欠陥を含むプログラムとテストスイートを受け取り，テスト実行時の実行経路情報であるカバレッジからプログラムのどの箇所に欠陥があるか推測する技術である．Aberu らは 7 つの欠陥限局の手法を比較し，Ochiai が優れた手法であると結論づけている [6].

表 1: 各処理の概要

処理	内容	実行時間の比率
ビルド	新しく生成したプログラムをバイナリ情報に変更する処理	22% ~ 60%
テスト	ビルド結果に基づきテストを実行する処理	7% ~ 63%
初期 AST 生成	プログラムのコード文字列を抽象構文木データに変更する最初の処理	2% ~ 24%
変更 AST 生成	変更コード片を修正対象プログラムの抽象構文木に反映する処理	6% ~ 26%
その他	上記以外の処理	1% ~ 36%

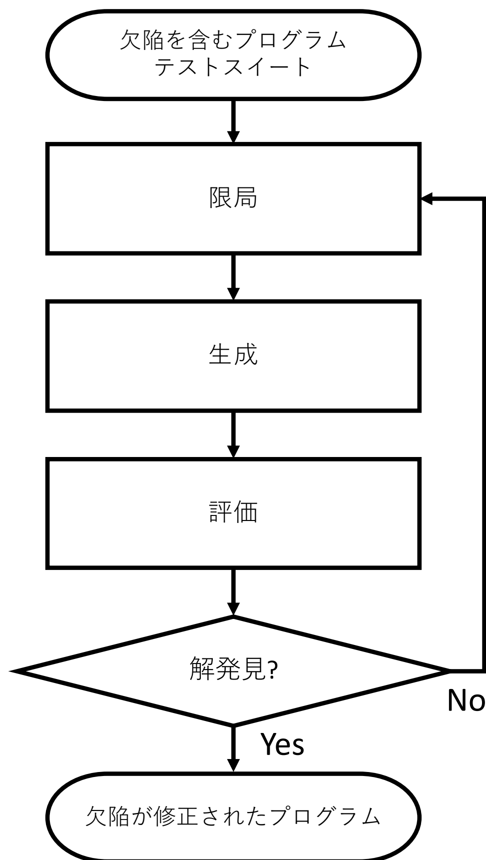

図 1: 自動プログラム修正のフローチャート

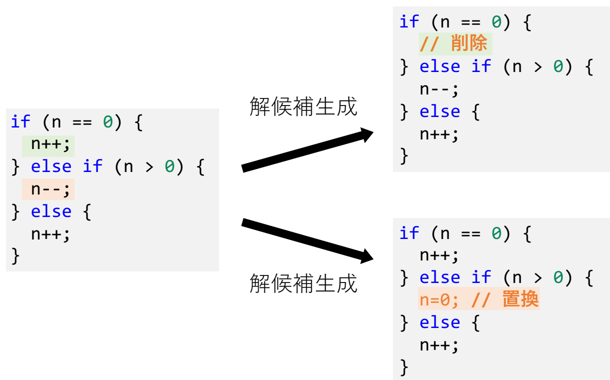

図 2: 従来手法のコード生成

3.1.3 プログラム生成と評価

欠陥限局により得られた情報に基づき，推定箇所のコード片ごとにそのコード片のみ変更したプログラムを生成する．次に推定箇所が変更されたプログラムにおいて欠陥が修正されているか評価する．従来の自動プログラム修正の手法では，限局されたコード片のみ変更したプログラムが変更の数だけ生成される．生成されたプログラム全てにビルドとテストを実行することでそのプログラムが欠陥を修正できたかを評価する．この時のコードの部分的な変更の実行例を図 2 に示す．

自動プログラム修正は，生成されたプログラムのビルドを実行し，ビルドを通過したプログラムのみテスト実行を行う．

3.2 遺伝的アルゴリズム

遺伝的アルゴリズムとは，生物学的な自然進化の仕組みを模倣した探索アルゴリズムの一種である [7]．探索する解候補データを生物の個体と見立て，対象の遺伝子のうち，求める解に近しい，すなわち適応度の高い個体を選択し，選択された個体に対して遺伝子操作と呼ばれるいくつかの操作を適用し変化させた新たな個体を産み出す．この操作を繰り返すことにより環境に適した，最適解に近似した個体を求めることを目的とする．遺伝的アルゴリズムは自動プログラム修正における解探索手法の 1 つとしてしばしば適用されている [8] [9]．

遺伝子操作には以下の種類がある．

選択 環境への適応度に基づき個体を選択する．適応度とは，最適解への近さを数値化した指標で，一定のアルゴリズムに基づき実装された評価関数により算出される．自動プログラム修正における適応度とは，入力されたテストスイートの成功度合いで定義される．

変異 個体内部の遺伝子情報の一部の処理を部分的に変更する．自動プログラム修正においては，プログラム内部のコード片を部分的に変更させる処理を指す．

交叉 親が持つ遺伝子同士を交配させて子供を生み出すことをモデル化した操作で，親個体の情報を掛け合わせることで新たな個体を生成する．

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，変異処理，すなわち欠陥限局により選択されたコード片に対して処理内容を変更させる方法として以下の操作がある．

挿入 限局箇所の前後のどちらかに，別のプログラムのコード片を挿入する．

置換 限局箇所のコード片を別のコード片へと置き換える．

削除 限局箇所のコード片を削除する．

4. 提案手法

4.1 概　要

本研究では，ビルド時間を削減するアイデアとして，テスト実行時にどの変更コード片を反映するかを動的に切り替えられるプログラムを生成及びビルドする．本研究では，欠陥限局された候補のうち無作為に選択された全ての変更コード片に基づき生成及びビルドする手法の提案する．

この手法により，生成プログラムをビルドした後，テスト実行にてプログラムの変更コード片を動的に切り替えられるようになる．したがって，従来手法では変更コード片の数だけ生成されていたプログラムを 1 つのプログラムの生成のみで実現可能となる．

従来手法と提案手法の時間はそれぞれ式 (1) と式 (2) となる．このとき，第一項がビルド時間，第二項がテスト時間，第三項が AST 生成時間を表す．式中の変数 n, m はそれぞれ世代内で変更するコード片数，テストスイート数を表す．

$$\text{従来手法} : O(n) + O(n * m) + O(n) \tag{1}$$

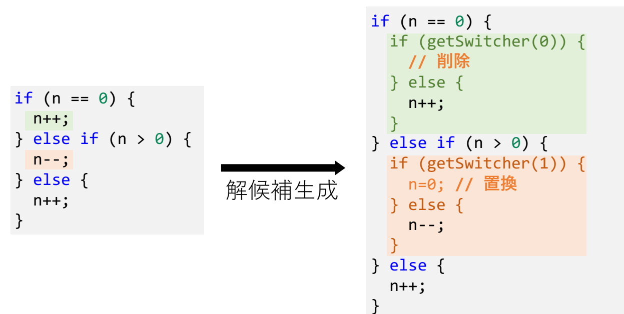

図 3: 提案手法のコード生成

$$\text{提案手法} : O(1) + O(n * m) + O(n) \tag{2}$$

従来手法では，生成するプログラムの数だけビルドを実行しなければならないのに対して，提案手法では各世代で生成されるプログラムが 1 つのみであるため，その単一のプログラムに対してのみビルドを実行すれば良い．

テスト実行に関しては，生成された変更コード片ごとにテストスイートで定義されたテストを行う必要があり，かつ従来手法と提案手法のそれぞれでこれらの数は変化しないため，時間計算量が等しくなる．AST 生成の時間計算量に関しても，生成される変更コード片の数は手法の間で変わらないため等しくなる．

4.2 実　装

提案手法の実装について説明する．今回提案手法の検証に用いた自動プログラム修正ツールは kGenProg [5] である．

4.2.1 kGenProg

kGenProg とは，遺伝的アルゴリズムを用いて実装された自動プログラム修正ツールである．

kGenProg は修正対象プログラムに対して欠陥限局したコード片を変異と交叉により変更する．この処理を繰り返すことによって欠陥を含まないプログラムを生成できる．なお，本研究でこのツールを用いる．

修正対象プログラムに対して新たに適応させるコード変更を `if` 文に組み込んで 1 つのコードに集約する．`if` の条件式では，`true` にすると切替を実行し，`false` では行わないように実装する．`if` の `then`，`else` 文の内部の変更内容に関しては，表 2 にて示す．

この時のプログラムの生成例を図 3 に示す．

4.2.2 動的コード切替

テスト実行時に変更コード片を切り替えるが，このときに自動プログラム修正ツールは修正対象プログラムの変更部の `if` 条件を切り替えることができるスイッチを持っており，このスイッチを切り替えることにより，修正対象プログラムのソース

表 2: 条件式の変更部分

	then(反映する場合の文)	else(反映しない場合の文)
挿入	変更後の文	なし
削除	なし	変更前の文
置換	変更後の文	変更前の文

```
List<boolean> sw = List.of(false, true);
public boolean getSwitcher(int i){
    return sw.get(i);
}
```

図 4: 動的切替の参照先

```
if (switcher(1)) {
   n=0; // 置換
} else {
   n--;
}
```

変更を反映 →

```
if (true) {
   n=0; // 置換
} else {
   n--;
}
```

図 5: 動的切替が実行されたプログラム

コードの動的な変更が可能になる．例として，先に挙げた図 3 における二番目のコード変更部分について，`switcher(1) = true` と動的にコード片を切り替える．

すると，自動プログラム修正ツールが保持している切替の参照データを管理するプログラムは図 4，実際に切り替えたコードは図 5 となる．

このように，変更コード片をテスト実行時に動的に切り替えることにより，各世代において 1 回のビルド結果に対して動的に切り替えたプログラムのテスト実行を行いプログラムが修正できたかを評価できる．

4.2.3 ビルドエラーの回避

提案手法では，各世代において変更された単一のプログラムに対してのみビルドを行う．しかし，この手法を実現するにあたり問題が生じたため，問題の概要及びその解決策に関して述べる．

提案手法は変更コード片を単一のプログラムに集約する．そのため，複数の変更コード片の中にビルドエラーを発生させるコード片が含まれていると，変更を集約しているプログラムがビルドエラーとなってしまい，自動プログラム修正を実行できなくなる致命的な問題が生じうる．

本問題を解決する手法として，1 度ビルドを行うことで，ビルドエラーを発生させている変更コード片の箇所を特定する．変更 AST の生成時にその変更を実行しないようにする．この案により，ビルドエラーを発生させるコード片を含まない，すなわちビルドエラーを発生させないコード片のみを含むプログラムを生成することが可能になる．

ここで，ビルドエラーが発生した箇所が変更箇所と同じ場合はコードの変更を行わない．変更箇所と異なる場合はコード内部でエラー原因を探索しその変更箇所の変更を行わないようにすることで当問題を解決する．前者は，変更箇所をコードから除去することによりビルドエラーを発生させないようにする．後者は，ビルドエラーが発生した箇所の付近にある変更コード片の場所を探索し，その変更がビルドエラーのエラー内容と関連性が高い場合はヒューリスティック処理で除去する．

ヒューリスティック処理の例を，実際のビルドエラーを発生させるコード片の探索例を挙げて説明する．

ビルドエラーが発生したプログラム内部のコード修復を行った後には，修復が正しくされているか確認するために再度ビルドを行う必要がある．この時のビルドはエラー発生原因となる変更コード片を全て除去した後に行うため，1 回のみで良い．

```
public class Operator {
    private final int a;
    public void sample() {
        a = 0;
        if (getSwitcher(0))
            b = 1; // 挿入
    }
}
```

図 6: エラー発生箇所と同じ場所の変更の場合

```
public class BigFraction {
    private final BigFraction TWO = new BigFraction(2);       // 初期化
    private final BigFraction ONE = new BigFraction(1);       // 初期化
    private final BigFraction ZERO = new BigFraction(0);      // 初期化
    private final BigInteger numerator;                       // 未初期化
    private final BigInteger denominator;                     // 未初期化
    public BigFraction() {
        long p2 = 0;
        long q2 = 1;
        if (!getSwitcher(1))
            numerator = BigInteger.valueOf(p2); // 削除
        denominator = BigInteger.valueOf(q2);
    }
}
```

図 7: エラー発生箇所と異なる場所の変更の場合

5. 評 価 実 験

実験について説明する．実験では提案手法で実装された kGenProg と既存の kGenProg のそれぞれを実行し実行時間を比較する．この実験の意図は，二つの手法の間で生じる実行時間の違いを確かめ，実行速度が上がっているかを確かめることにある．

5.1 実 験 設 計

自動プログラム修正の実行時に計測する時間データは以下の通りである．

ビルド 変更コード片に基づいて生成されたプログラムに対してビルドを実行する時間.

テスト 上記のビルド結果に基づいてテストスイートを用いてテスト実行を行う時間.

初期 AST 生成 自動プログラム修正を実行する際の初期の段階で，修正対象プログラムを文字列データから抽象構文木へ変換する時間.

変更 AST 生成 修正対象プログラムに対し欠陥限局された箇所のコード片を変更する時間.

その他 全体時間から上記処理を除いた時間.

5.2 実 験 対 象

本研究では，小規模なプロジェクトとして CloseToZero を，大規模なプロジェクトとして Apache Commons Math を実験対象として選択した．プロジェクトの内容を表 4 に示す．

表 3: kGenProg の設定

項目	CloseToZero	Apache Commons Math
バージョン	1.7.4	1.7.4
最大世代数	1,000	1,000
タイムアウト	360	360
シード値	0, 1, 2	15, 41

```java
public class CloseToZero {
    public int close_to_zero(int n) {
        if (n == 0) {
            n++; // bug here
        } else if (n > 0) {
            n--;
        } else {
            n++;
        }
        return n;
    }
}
```

図 8: CloseToZero のコード

CloseToZero

プログラムの一部を図 8 に示す[1]．与えるテストスイートは表 5 とする．プログラムの修正内容は，0 が引数として与えられたときに予測値である 0 ではなく 1 が出力されるという欠陥である．

Apache Commons Math

大規模プロジェクトとしては，Defects4J [10] に記録されている Apache Commons Math を実験対象とする．Defects4J とは，OSS の開発中に発生した欠陥を収集したデータセットであり，自動プログラム修正の評価に用いられる．欠陥の情報には，プログラムの欠陥発生箇所や失敗したテストケースの情報などが含まれている．本実験を行う際の Defects4J における欠陥 ID を 1 とした．

5.3 実験結果

実験結果を表 6 及び表 7 に示す．また，結果をグラフ化したものを図 9 及び図 10 に示す (左が従来手法，右が提案手法)．

実験結果から，CloseToZero ではビルド時間が 440 ミリ秒から 390 ミリ秒になりビルド時間が従来手法の 89% となった．Apache Commons Math では 20.7 秒から 9.5 秒になりビルド時間が 46% に削減されていることが分かる．

また，変更コード片を大量に生成するようにして自動プログラム修正を実行した場合，従来手法で実装された自動プログラム修正ツールでは大規模プロジェクトの欠陥修正時にメモリ不足による例外処理が発生し実行が不可能であった．一方で，提案手法ではメモリに負担をかけずに実行することができた．

(注1)：著者らが用意した題材である

表 4: 各プロジェクトの内容

修正対象	プログラム数	ファイルあたりの総行数
CloseToZero	1	28
Apache Commons Math	813	230

表 5: テストスイート

実行値	予測値
close_to_zero(10)	9
close_to_zero(100)	99
close_to_zero(0)	0
close_to_zero(-10)	-9

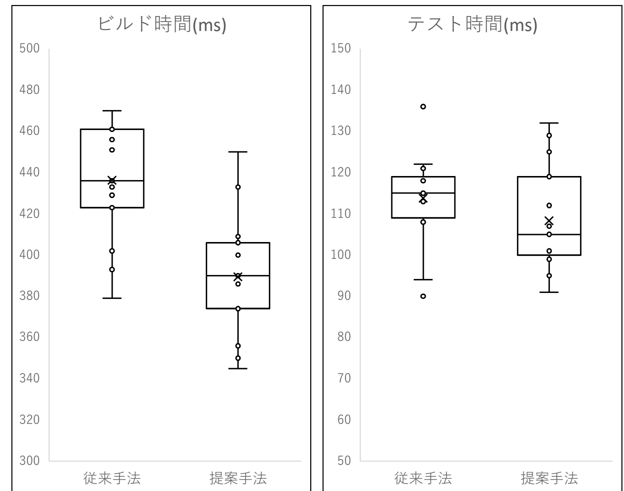

図 9: CloseToZero の実行結果

6. 考察

実験結果より，従来手法と比較して提案手法の方がプログラムの修正に要するビルド時間の削減に成功したことが確認できた．しかし，章 4. にて予想していたプログラムのビルド時間が占める時間計算量の削減率と比較して，削減率は大きくなかった．原因としては，検証ツールである kGenProg はビルド実行は実行開始時のビルド結果を利用した差分ビルドを行っており，変更がされた差分プログラムのビルド実行しか行っていないことが考察される．初期 AST 生成の時間が従来手法と提案手法の間で変化がなかったのは，単に実装上の変更を行わなかったことが理由である．変更 AST の生成時間が削減されたのは，ビルドエラーを回避するため AST 生成時に構文木チェックを導入したことによって，構文エラーを発生させる変更コード片を除去し，冗長な AST 生成処理が削減されたことが原因だと考察する．テスト時間の実行時間について，従来手法と提案手法のそれぞれで外れ値が出たことが平均時間の差異が発生しており大

表 6: CloseToZero の実験結果

	従来手法 (ms)	提案手法 (ms)	時間の削減率 (%)
ビルド	440	390	89
テスト	120	110	92
初期 AST 生成	240	240	100
変更 AST 生成	270	220	81
その他	340	200	59
合計時間	1,410	1,160	82

表 7: Apache Commons Math の実験結果

	従来手法 (s)	提案手法 (s)	時間の削減率 (%)
ビルド	20.7	9.5	46
テスト	33.7	10.5	31
初期 AST 生成	3.1	3.1	100
変更 AST 生成	0.9	0.8	89
その他	1.2	1.5	125
合計時間	56.6	25.4	43

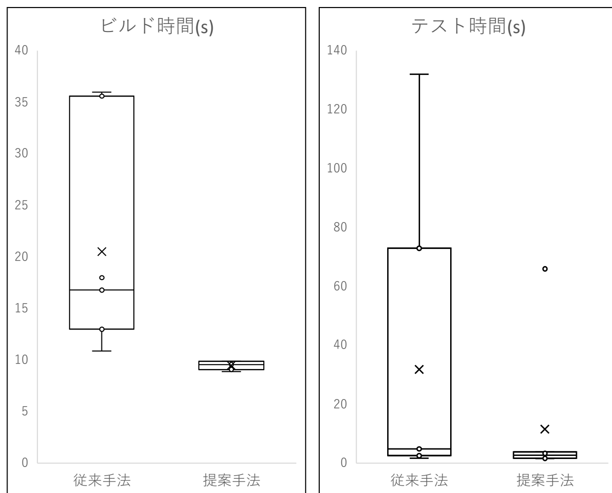

図 10: Apache Commons Math の実行結果

半のテスト時間は共に近い値であるため時間計算量に変化は.

7. 妥当性への脅威

提案手法の問題点として挙げたビルドエラー問題の解決策として，ビルドエラーが発生した箇所を特定し，その変更コード片の変更を取り消すことにより問題の解決策とした．現時点でヒューリスティックな解決策はまだ十分な数のデータが揃っていない．そのため，今後別のプロジェクトに対して提案手法に基づいた自動プログラム修正を実行したときに，これまでに認識されていなかった新たなビルドエラーの発生要因が生じたときにはビルドエラーの修復ができなくなる可能性がある．

8. おわりに

本研究では，従来の自動プログラム修正の実行過程で行うビルド実行が処理全体の内，大きな比重を占めていることから，ビルド時間を削減するというアプローチを試みた．結果として，プログラム内部の変更を行うコード片を多く含んでいるほど従来手法と比較して提案手法の方が短い時間での処理を可能とした．

今後の課題として，交叉アルゴリズムへの提案手法の適用が挙げられる．今回の研究を行うにあたり使用したツール kGenProg では，遺伝的アルゴリズムのうち選択と変異を従来手法と同一のアルゴリズムで行うようにし，生成した個体に関して評価するプロセスで生じるビルドというオーバーヘッドをなるべく削減することを目的とした．しかし，本実装では交叉アルゴリズムについての検証を行うことができていないため，アルゴリズムの適用により従来手法と比べて効率的なプログラム修正を期待できる [11].

謝辞　本研究は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：20H04166）の助成を得て行われた．

文　献

[1] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, “ Reversible Debugging Software: Quantify the time and cost saved using reversible debuggers ,” 2013.

[2] L. Gazzola, D. Micucci, and L. Mariani, “Automatic software repair: A survey,” IEEE Transactions on Software Engineering, vol.45, no.1, pp.34–67, 2019.

[3] M. Martinez and M. Monperrus, “Astor: Exploring the design space of generate-and-validate program repair beyond genprog,” Journal of Systems and Software, vol.151, pp.65–80, 2019.

[4] C.A.F. Liushan Chen, Yu Pei, “Contract-based program repair without the contracts,” ASE 2017: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pp.637–647, Oct. 2017.

[5] Y. Higo, S. Matsumoto, R. Arima, A. Tanikado, K. Naitou, J. Matsumoto, Y. Tomida, and S. Kusumoto, “kgenprog: A high-performance, high-extensibility and high-portability apr system,” Proceedings of 2018 25th Asia-Pacific Software Engineering Conference (APSEC), pp.697–698, 2018.

[6] R. Abreu, P. Zoeteweij, R. Golsteijn, and A.J.C. vanGemund, “A practical evaluation of spectrum-based fault localization,” J. Syst. Softw., vol.82, no.11, p.1780–1792, Nov. 2009.

[7] P. BAJPAI and M. Kumar, “Genetic algorithm - an approach to solve global optimization problems,” Indian Journal of Computer Science and Engineering, vol.1, pp.199–206, Oct. 2010.

[8] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, “A genetic programming approach to automated software repair,” Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, p.947–954, GECCO ’09, Association for Computing Machinery, New York, NY, USA, 2009.

[9] C. Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” Software Engineering, IEEE Transactions on, vol.38, pp.54–72, 03 2012.

[10] R. Just, D. Jalali, and M.D. Ernst, “Defects4j: A database of existing faults to enable controlled testing studies for java programs,” Proceedings of the 2014 International Symposium on Software Testing and Analysis, p.437–440, ISSTA 2014, Association for Computing Machinery, New York, NY, USA, 2014.

[11] V. Oliveira, E. Souza, C. Goues, and C. Camilo-Junior, “Improved crossover operators for genetic programming for program repair,” Proceedings of SSBSE, pp.112–127, Oct. 2016.