特別研究報告

題目

自動プログラム修正における変更コード片の畳み込みによるビルド時間削減の提案

指導教員

楠本 真二 教授

報告者

古藤 寛大

令和 3 年 2 月 9 日

大阪大学 基礎工学部 情報科学科

令和 2 年度 特別研究報告

自動プログラム修正における変更コード片の畳み込みによるビルド時間削減の提案

古藤 寛大

内容梗概

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複数個限局したのちコード片ごとにその箇所のみ変更したプログラムを生成し，評価時に全ての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，本研究では，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．

主な用語

自動プログラム修正，ビルド時間，遺伝的アルゴリズム

目次

図目次

表目次

1　はじめに

ソフトウェア開発において，プログラム内部にて開発者が予期しない動作，すなわち欠陥の発生の回避は困難である．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 [?].

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 [?]．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[*1]に基づく手法がある [?]．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 [?]．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．

本研究で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．

以降，2 章では本研究を行うに至った研究の動機について説明する．3 章では本研究の提案手法に必要な事柄について説明する．4 章では研究目的である自動プログラム修正のビルド時間削減を実現させる手法，及び提案手法の実装について述べる．5 章では提案手法の適用実験について述べ，6 章で実験結果についての考察を行う．7 章で妥当性への脅威について述べ，8 章で本研究のまとめと今後の課題について述べる．

[*1] Generate and Validate

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 [?]．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[?] 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．

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%

3 準備

3.1 抽象構文木

抽象構文木 (以降，AST) はソースコードを構文解析することで得られる木構造のデータである．1 つのソースファイルに対して 1 つの AST を構築し，AST の各ノードは以下の情報を持つ．

親ノード ：AST の各ノードは，木構造上の親ノードへの参照を持つ．ただし，根ルートの親は存在しないので何も保持しない．

子ノード ：AST の各ノードは，木構造上の子ノードへの参照を持つ．ただし，葉ルートの子は存在しないので何も保持しない．

3.2 自動プログラム修正

3.2.1 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．

3.2.2 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．欠陥箇所の限局手法はプログラムを解析して欠陥の候補を探索する手法であり，最も欠陥の可能性が高い候補を提示して欠陥修正を補助する．欠陥限局には，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からプログラムのどの箇所に欠陥があるか推測する手法 (SBFL[*2]) がある．各テストスイートの実行経路情報と成否情報に基づいてプログラムの各コード片に対して疑惑値を出力する．疑

[*2] Spectrum-Based Fault Localization

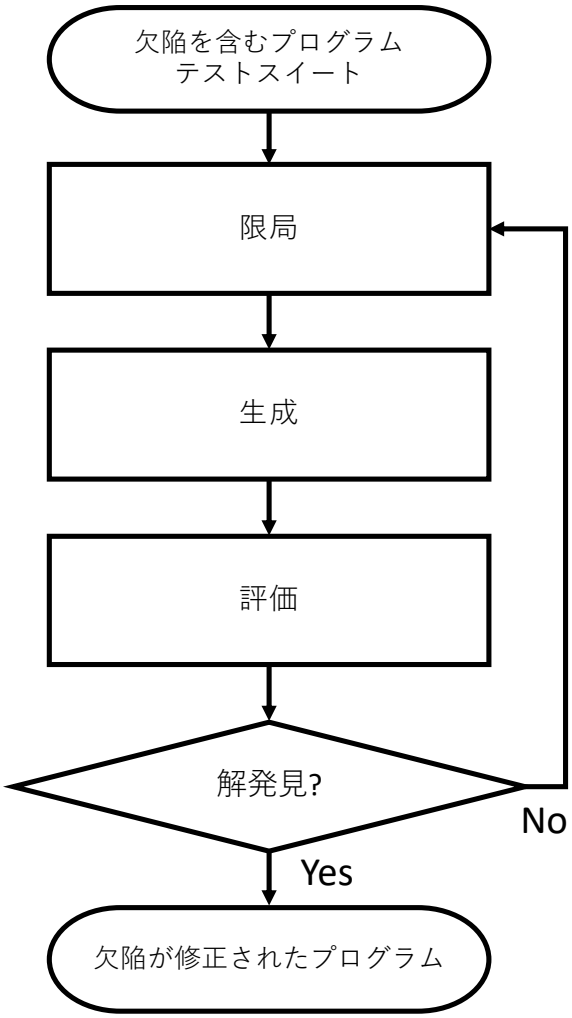

図 1　自動プログラム修正のフローチャート

惑値とは，欠陥箇所である可能性の高さを示す値である．Abreu 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 [**?**]．そのため，本研究においても疑惑値の計算に用いる．Ochiai の計算式を式 1 に示す．

$$susp(s) = \frac{fails(s)}{\sqrt{totalfail * (fail(s) + pass(s))}} \tag{1}$$

s ：疑惑値計算対象の文

$totalfail$ ：失敗テストの総数

$fails(s)$ ：s を実行した失敗テストの数

$pass(s)$ ：s を実行した成功テストの数

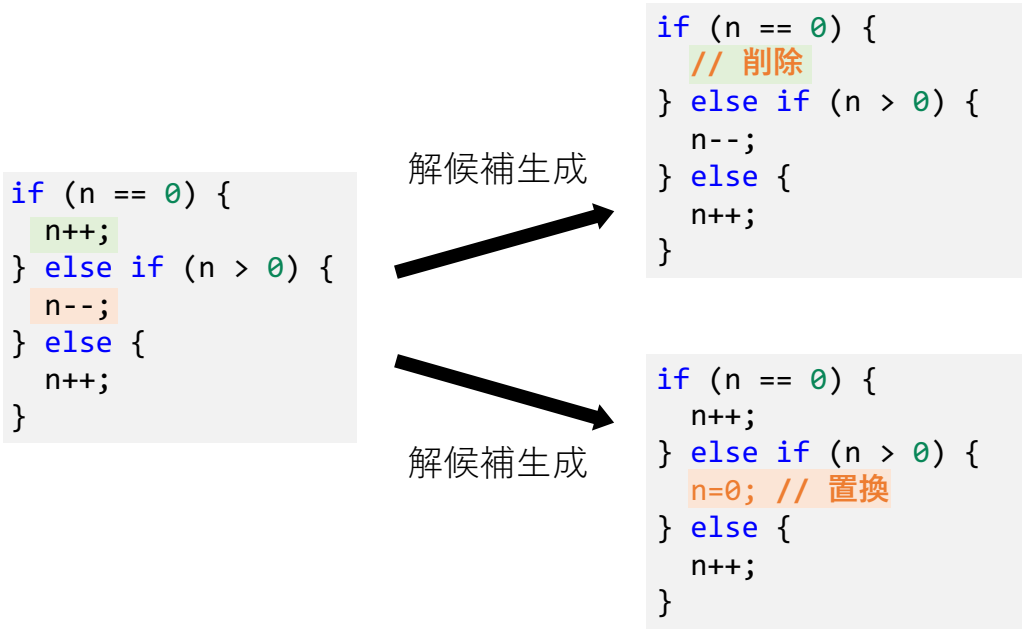

図 2　従来手法のコード生成

3.2.3　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．

3.3　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 [?]．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 [?][?]．

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させる．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．代表的な交叉の手法は一点交叉，一様交叉およびランダム交叉である．

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[?] 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，本研究では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` 文の子ノードである `then` 文，`else` 文の変更内容は表 2 にて示す．この時のプログラムの生成例を図 3 に示す．

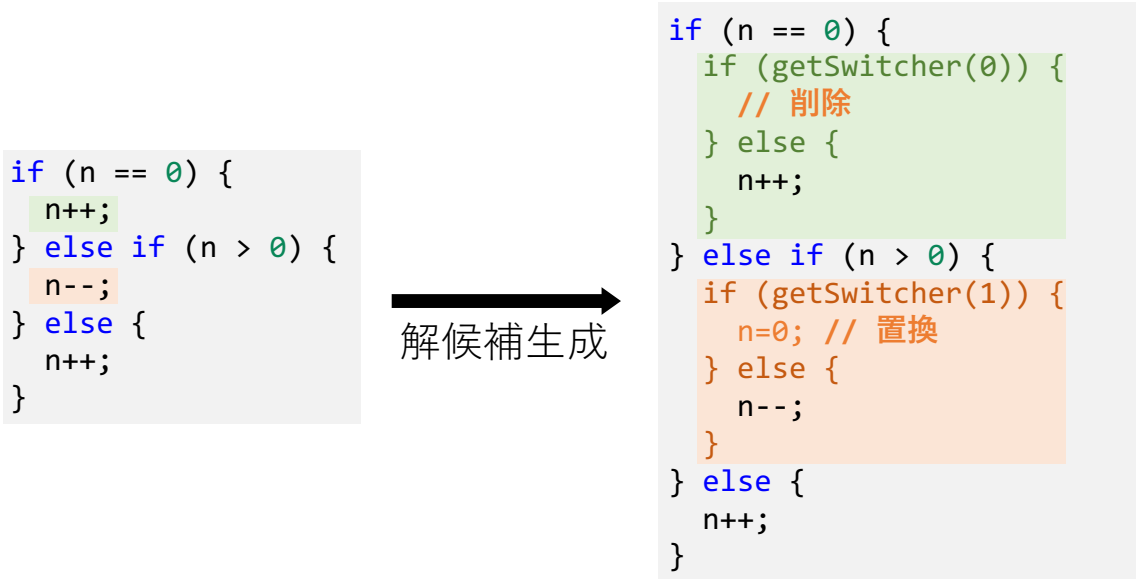

図 3　提案手法のコード生成

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動的な変更が可能になる．テスト実行時におけるスイッチの切替の様子を図 4 に示す．テスト実行をする前に自動プログラム修正ツールが保有するスイッチの情報を自動プログラム修正ツールから設定する．そしてテスト実行中に修正対象プログラムが設定されたスイッチの情報を修正対象プログラムより呼び出す．この実装によって，自動プログラム修正ツールがテスト実行時におけるプログラムの内容を切り替えることができる．例として，先に挙げた図 3 におけるコード変更部分について，自動プログラム修正ツールが図 5 のように切替スイッチを設定したとする．すると，修正対象プログラム側からスイッチを呼び出した時の値はそれぞれスイッチ 1 が `false`，スイッチ 2 が `true` である．この操作により，実際に切り替えたコードが図 6 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

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

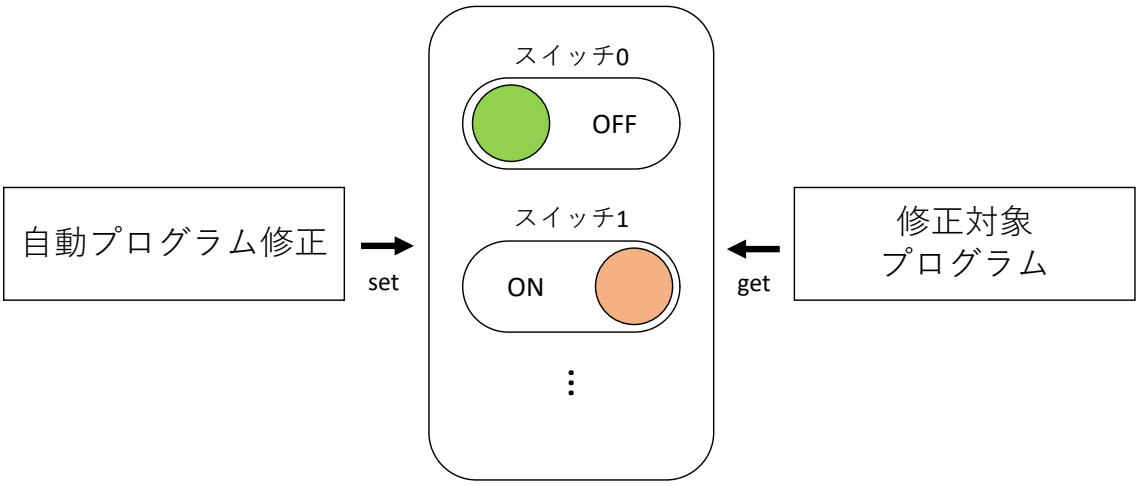

図 4　動的コード切替の全体像

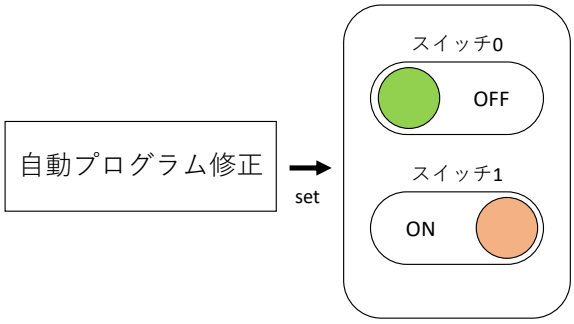

図 5　スイッチの設定

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する．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．

自動プログラム修正は，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可能性があり，エラーを発生させるかビルドを実行するまでは分からない．そのため従来手法ではビルドエラーを発生させない個体のみ候補として選択するが，提案手法では変更コード片を単一のプログラムに集約するため，集約するプログラム内部にビルドエラーを発生させるコード片が含まれるとプログラム全体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問題が発生する．

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

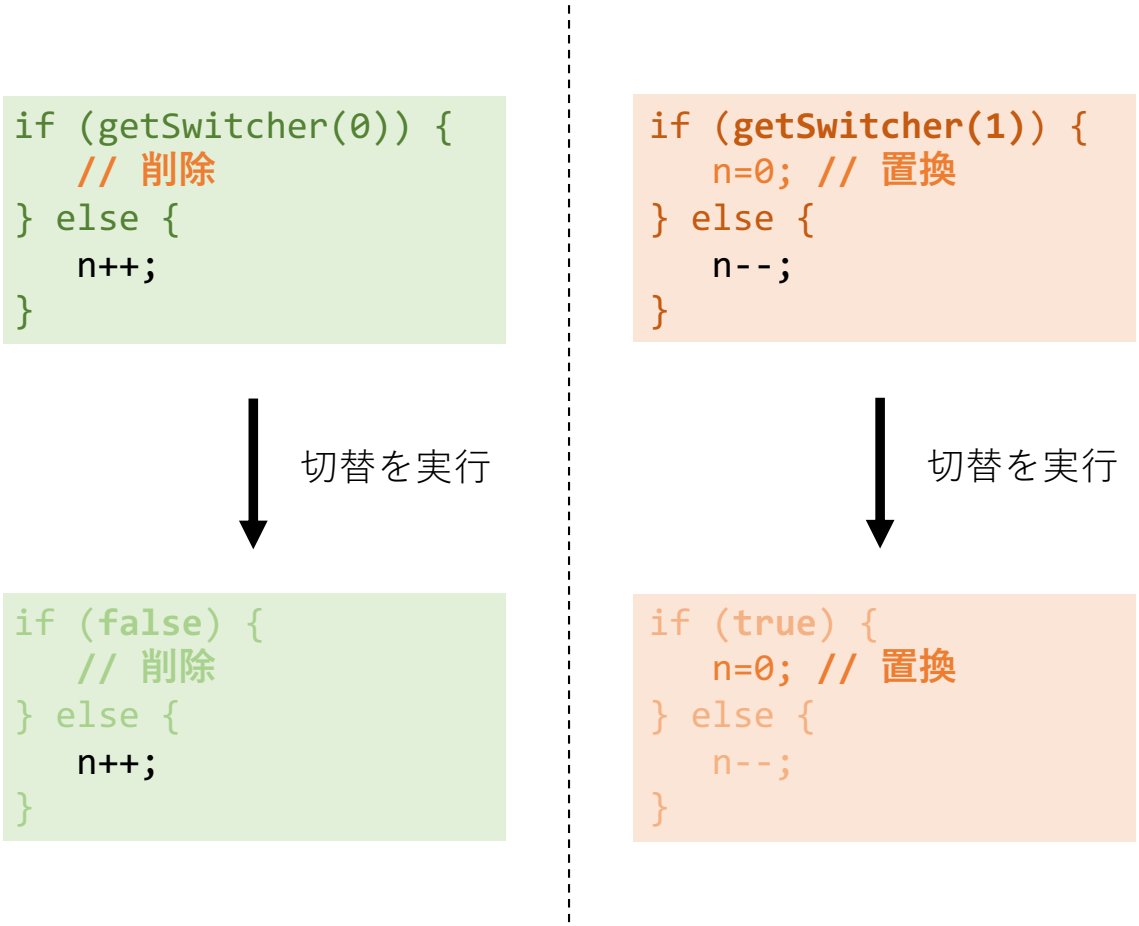

図 6　動的切替が実行されたプログラム

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．除去するコードはエラー発生箇所である AST ノードの親ノードにあたる `if` 文である．図 7 では，新たな変更コード片 `b = 1;` が挿入されているが，クラス内部で宣言されていない変数 `b` を利用しているため，ビルド時に変更コード片の箇所でエラーが発生する．そのため変更の取り消しを行うことでエラーを防ぐ．

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

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

図 7　エラー発生箇所と同じ場所の変更を取り消したプログラム

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．図 8 では既存のコード片 `numerator = BigInteger.valueOf(p2);` を削除する変更を行っている．このプログラムには，`final` 修飾子がついた変数 (以降，`final` 変数) のうち初期化された変数に `TWO`，`ONE` 及び `ZERO` が存在し，初期化されていない変数に `numerator` と `denominator` が存在する．しかし，コンストラクタ内部にて `final` 変数 `numerator` を初期化するコード片が自動プログラム修正によって削除され，その後変数を初期化する処理が現れないため，`final` 変数の未初期化によるエラーがコンストラクタ末尾にて発生する．そのため，ヒューリスティックな解決策として，プログラム内部で `final` 変数が宣言されておりかつ初期化されていない場合はその変数情報をストックし，エラーが発生したコンストラクタ内の変更コード片でその変数の操作を行っている場合は変更を取り消す．すると，既存コード片 `numerator = BigInteger.valueOf(p2);` は `final` 変数 `numerator` の初期化を実行しているため，この変更コード片の削除処理を取り消す．この取り消しによりビルドエラーの発生原因を取り除くことができる．

ビルドエラーが発生したプログラム内部のエラー箇所の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 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 8　エラー発生箇所と異なる場所の変更を取り消したプログラム

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するビルド実行を行う時間．

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 AST へ変換する処理を行う時間．

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．

本実験を行った計算機及び kGenProg の各種設定を表 3 と表 4 に示す．

表 3　本実験で利用した計算機

項目	内容
CPU	2GHz クアッドコア Intel Core i5
メモリ	16GB

```java
package example;

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 9　CloseToZero のプログラム

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 5 に示す．

CloseToZero

プログラムを図 9 に示す[*3]．与えるテストスイートは表 6 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

表 4　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	1000	1000
タイムアウト	360	360
シード値	0, 1, 2	15, 41

表 5　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

[*3] 著者が用意した題材である

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[?] 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

図 9 にて示した CloseToZero のプログラムに対して修正を行った際のプログラムの変更を図 10 及び図 11 に示す． 実験結果を表 7 及び表 8 に示す．また，結果をグラフ化したものを図 12 及び図 13 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．

表 6 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

表 7 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	1410	1160	82

```java
package example;

public class CloseToZero {
  public int close_to_zero(int n) {
    if (n == 0) {
      if (getSwitcher(0))                    // getSwitcher(0) = false
        if (!getSwitcher(16))                // getSwitcher(16) = false
          n--;
        else if (!getSwitcher(2))            // getSwitcher(2) = true
          n++;
    } else if (n > 0) {
      n--;
    } else {
      n++;
    }
    if (getSwitcher(5))                      // getSwitcher(5) = true
      if (getSwitcher(84))                   // getSwitcher(84) = false
        return n;
      else
        return n;
    else if (getSwitcher(26))                // getSwitcher(26) = false
      return n;
    else
      return n;
  }
}
```

図 10　修正が成功するスイッチの組み合わせが見つかったプログラム

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

表 8　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

```java
package example;

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

図 11　修正後のプログラム

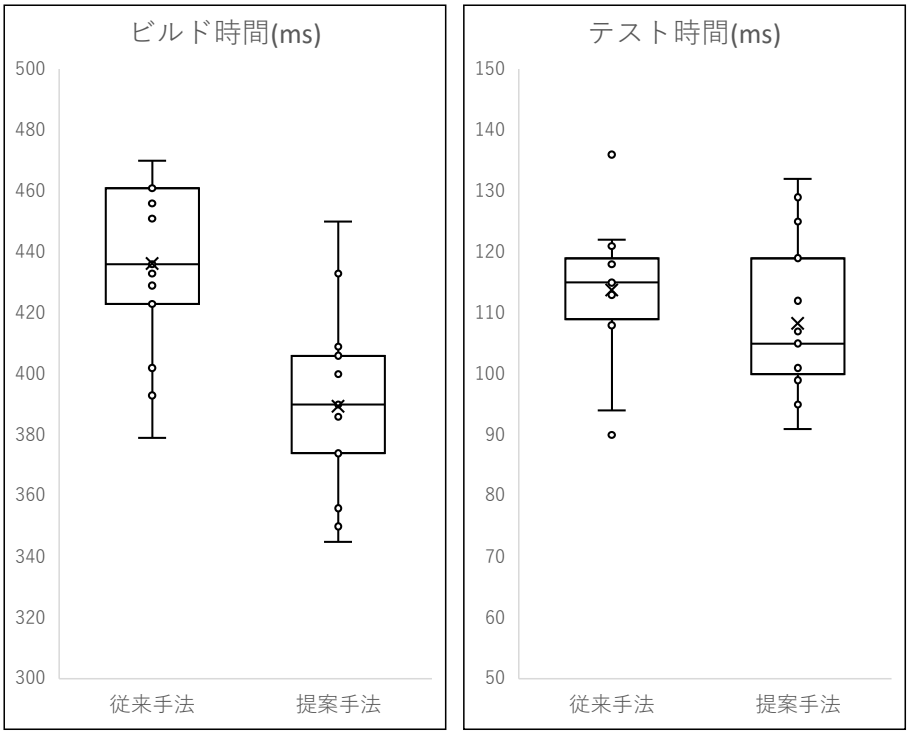

図 12　CloseToZero の実行結果

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

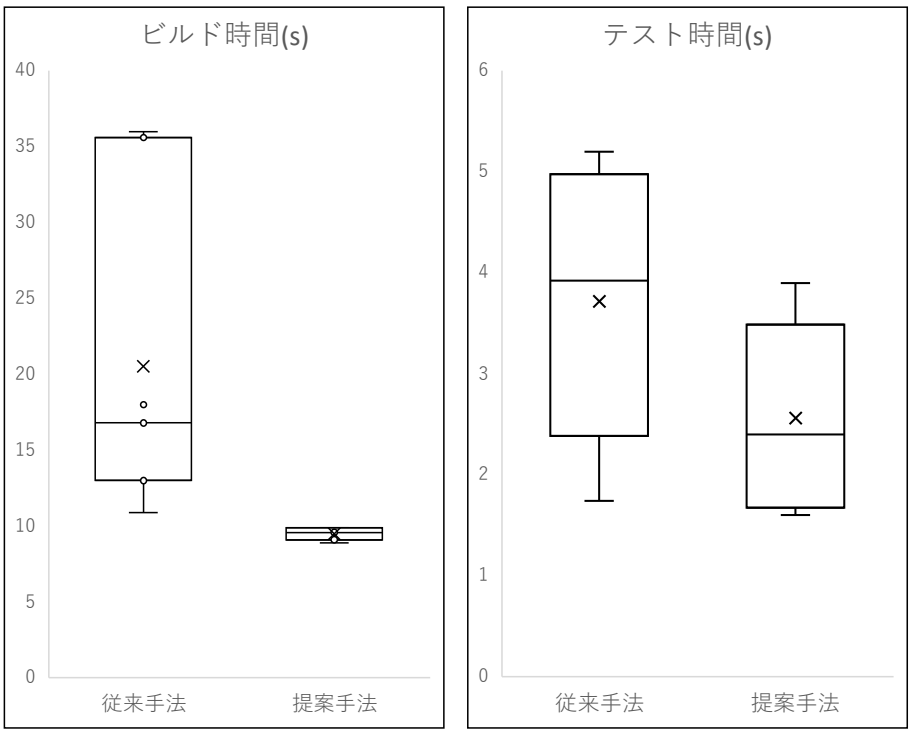

図 13　Apache Commons Math の実行結果

されたのは，ビルドエラーを回避するために構文木チェックを AST 生成時に導入したことにより，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．

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．

また，本研究で検証の対象としたプロジェクトの数が少ないため，その他のプロジェクトに対して提案手法を行ったときに本実験とは異なる結果となる可能性がある．

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において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．特にプロジェクトの規模が大きいほどビルドとテストの時間は大きくなり自動プログラム修正において支配的になることから，開発規模の大きいプロジェクトにおける提案手法の有用性を主張できる．

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 [**?**].

謝辞

本研究を行うにあたり，理解あるご指導を賜り，暖かく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.

本研究の全過程を通し，研究を行う際に不足していた知識を熱心にご教授いただき，個別のミーティングにおいて丁寧なご指導を賜りました肥後芳樹准教授に深く感謝申し上げます.

本研究に関して，実装における有益なご助言や研究に関連した興味深いお話を頂きました，柗本真佑助教に深く感謝申し上げます.

本研究の過程において，生活や事務の面から支えていただいた事務員の神谷智子氏に心から御礼申し上げます.

本研究を進めるにあたって，本研究の分野に関する知識や論文の紹介をしていただいた大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程 2 年の九間哲士氏，富田裕也氏，ならびに論文執筆にあたり細部に至るまで多くのご助言をいただいた同専攻博士前期課程 2 年の中川将氏に深く感謝申し上げます.

本研究を進めるにあたり，様々な形で励まし，ご助言を頂きましたその他の楠本研究室の皆様のご協力に心より感謝致します.

本研究に至るまでに，講義，演習，実験等でお世話になりました大阪大学基礎工学部情報科学科の諸先生方に，この場を借りて心から御礼申し上げます.

最後に，これまで様々な面で支えて頂き，励まして頂いた家族，祖父母にも心より感謝いたします.