令和 4 年度

修士学位論文

GenProg を用いた Scratch プログラムの自動修正

Automatic Repair of Scratch Programs Using GenProg

1255110　　高橋 智哉

指導教員　　高田 喜朗

2023 年 2 月 28 日

高知工科大学大学院 工学研究科 基盤工学専攻

情報学コース

要 旨

GenProg を用いた Scratch プログラムの自動修正

高橋 智哉

現在，小学校においてプログラミング教育は必修であり，小学生に対するプログラミング教育の教材として，学習者が構文について意識することなくプログラムを作成できる Scratch がよく利用されている．プログラミング学習において，初学者が作成したバグを含むプログラムに対して，教育者がフィードバックを返すことは教育上良いとされている．しかし，学習者の人数が多くなると教育者の負担が大きくなる．そのため，教育者が学習者のプログラムを理解することを補助するシステムが求められている．自動プログラム修正（APR）手法によってバグの修正方法を提示することは，バグの位置の特定やバグの原因について理解することを補助する方法として有効と考えられる．しかし，既存の APR ツールは Scratch に対応していない．

本研究では，C 言語用の APR ツールである GenProg に Scratch 用のテストフレームワークである Whisker を適用した Scratch 用 APR ツールを開発し，その有用性を評価する．本システムでは主に 2 つの機能を実装する．1 つ目は，入力として与えられたバグを含む Scratch のプロジェクトを C 言語のソースコードに変換する機能である．2 つ目は，GenProg が生成した C ソースの変異プログラムを Scratch プロジェクトに逆変換する機能である．

3 つの Scratch の入門者用プロジェクトに対して，ランダムに選択されたブロックに変更を加え，本システムを適用した．その結果，入力として与えられたプロジェクト中に修正に必要なブロックがある場合，修正率は 65.0%であった．また，Scratch Web サイトにユーザが投稿している，入門者用プロジェクトを参考に作成されたと思われるプロジェクトの中からランダムに選択し，本システムを適用した．その結果，入力として与えられたプロジェク

ト中に修正に必要なブロックがある場合，修正率は約 66.7%であった．

キーワード　自動プログラム修正，Scratch，GenProg，Whisker

Abstract

Automatic Repair of Scratch Programs Using GenProg

Tomoya TAKAHASHI

Currently, programming education is compulsory in elementary schools in Japan, and Scratch, which allows students to create programs without being aware of the syntax, is often used as a teaching material for programming education for elementary school students. In learning programming, it is considered good for education if educators return feedback to programs containing bugs created by novice programmers. Therefore, there is a need for a system that assists educators in understanding the programs created by learners. Presenting bug fixes through automatic program repair (APR) is considered an effective way to assist users in locating bugs and understanding the causes of bugs. However, existing APR tools are not compatible with Scratch.

In this research, we develop an APR tool for Scratch by applying Whisker, a testing framework for Scratch, to GenProg, an APR tool for C, and evaluate its usefulness. Two main functions will be implemented in this system. The first is a function to convert a Scratch project containing bugs given as input into C source code. The second is a function to reversely convert the mutated program in C generated by GenProg into a Scratch project.

The system has been applied to three Scratch projects for beginners with modifications to randomly selected blocks. The results show that if the repairing can be achieved using the blocks in the project given as input, the fix rate is 65.0%. We also applied this system to randomly selected projects posted by users on the Scratch Web site that appeared to have been created with reference to the above-mentioned project

for beginners. The results show that if the repairing can be achieved using the blocks in the project given as input, the fix rate is 66.7%.

key words Automatic program repair, Scratch, GenProg, Whisker

目次

図目次

表目次

第 1 章

はじめに

1.1 研究の背景

現在，小学校においてプログラミング教育は必修であり，小学校におけるプログラミング教育の目指すものとして,「プログラミング的思考」を身に付けさせることが挙げられている [1]．小学校におけるプログラミング教育の円滑な実施に向けて，小学校プログラミング教育に関する研修教材が文部科学省から提示されている．そこでは,「Scratch 正多角形をプログラムを使ってかく」「Scratch ねこから逃げるプログラムを作る」が提示されている [2]．また，Scratch を用いたプログラミング教材も販売されており，プログラミング的思考を身につける方法として Scratch は有用と考えられる．

プログラミング学習において，学習者が作成したプログラムにバグが含まれている場合，教育者がそのプログラムを理解し，学習者にフィードバックを返す必要がある．しかし，学校やプログラミング教室において，教育者に対して学習者の人数は一般的に多いため，教育者の負担が大きくなる．そのため，教育者が学習者のプログラムを理解することを補助するシステムが求められている．

先行研究では，163 名の小学校教員研修生を対象に，バグのある Scratch プログラムを修正する課題を与え，自動的に生成されたバグパターンに関するヒントを与えた場合と与えなかった場合とを比較した結果，ヒントの自動生成により，バグの発見と修正に要する時間が 8.66 分から 5.24 分に短縮されると共に，正解率が 48%から 82%に向上した [3]．しかし，このツールによって与えられるヒントは，Scratch のよくあるバグの発生パターンに関するものであり，問題固有のバグに対するヒントを生成することが出来ない．

デバッグ支援における研究分野として自動プログラム修正（APR）がある [4]．自動プログラム修正とは，バグを含むプログラムとそのプログラムに対するテストケースを入力として，全てのテストケースに成功するプログラムを出力する手法である．APR 手法によってバグの修正方法を提示することは，バグの位置の特定やバグの原因について理解することを補助する方法として有効と考えられる．また，APR 手法を用いることで，問題固有のバグを修正することができる．しかし，既存の APR ツールは Scratch に対応していないため，既存の APR ツールを Scratch に対応させることが求められている．

1.2　研究の目的

本研究の目的は，既存の APR ツールを Scratch に適用させ，バグを含む Scratch スクリプトを自動的に修正するシステムを開発することである．そのため，本研究では，C 言語用の APR ツールである GenProg に Scratch 用のテストフレームワークである Whisker[5] を適用した Scratch 用 APR ツールを開発し，その有効性を評価する．

第 2 章

背景技術

2.1 自動プログラム修正

開発者は大きな労力をかけてソフトウェアの開発を行っているが，その中でもデバッグ作業は開発工数の約 50%を占めるといわれている [6][7]．そのため，デバッグを支援するための手法として様々なプログラム修正手法が研究されている [8][9][10][11][12][13][14][15][16][17]．

自動プログラム修正とは，プログラムの障害に自動的に対処するためのアプローチの１つであり，プログラムのバグを検出し，修正を適用できる場所を特定して，その位置に対してバグを修正するために必要な変更を行う [4]．このバグを修正するための変更はソースコードレベルで行われる．

修正プロセスは，以下の３つのステップで構成されている．

1. 位置特定ステップ

   障害位置特定手法や修正位置特定手法などによって修正を適用できる場所を特定する．障害位置特定手法は，障害のあるステートメントを特定することを目的としている．自動プログラム修正手法では，障害位置特定手法として，SpectrumBased Fault Localization（SBFL）[18] を広く使用している．

   SBFL は，多数の失敗したテストケースと少数の合格したテストケースによって実行されたプログラムの要素にバグがある可能性が高く，多数の合格したテストケースと少数の失敗したテストケースによって実行されたプログラムの要素は正しい可能性が高いという考え方に基づいて障害位置の推定を行う．

2. パッチ/修正ステップ

   位置特定ステップによって特定された位置にあるソースコードを変更する修正を生成する.

3. 検証ステップ

   パッチ/修正ステップによって生成された修正が実際にソフトウェアを修正したかどうかを判断する.

これらのステップは，障害が修正されるまで，修正を生成できなくなるまで，または修正プロセスに割り当てられた時間がなくなるまで，複数の場所に対して複数回繰り返される場合がある.

2.2　遺伝的プログラミング

遺伝的プログラミング（Genetic Programming：GP）とは，組み合わせ最適化問題の探索手法の 1 つであり，遺伝的アルゴリズムの遺伝子型を木構造に拡張し，遺伝子によって表されるプログラムを進化的に生成する手法である [19]．遺伝的プログラミングでは，解の候補となるプログラム（個体）を複数生成し，各個体を評価して適合度（適合度が高いほど，その個体が解に近いことを表す）を算出する．そして，適合度の高い個体を選択し，交叉や突然変異などの遺伝子操作を適用することで新たな個体を生成する．これらの手順を繰り返すことで解の探索を行う.

以下に遺伝的プログラミングの動作を示す.

1. 第一世代の個体群を生成
2. 各個体の適合度を計算（終了条件を満たせば終了）
3. 適合度に基づいて個体を削除する
4. 交叉や突然変異などを行い，次世代の個体群を生成する
5. ２～４を繰り返す

2.3 GenProg

自動プログラム修正手法の 1 つにソースコードの再利用に基づく手法である GenProg[8] がある．GenProg は，修正対象プログラム（バグを含むプログラム）およびテストスイート（テストケースの集合）を入力とし，遺伝的プログラミングに基づいてプログラムに変更を繰り返し加えることでプログラムの自動修正を行う．修正を発見した場合は，修正プログラム（入力として与えた全てのテストケースを通過するプログラム）を出力する．

以下では，GenProg の動作内容について述べる．図 2.1 は，GenProg の動作の流れを表したものである．

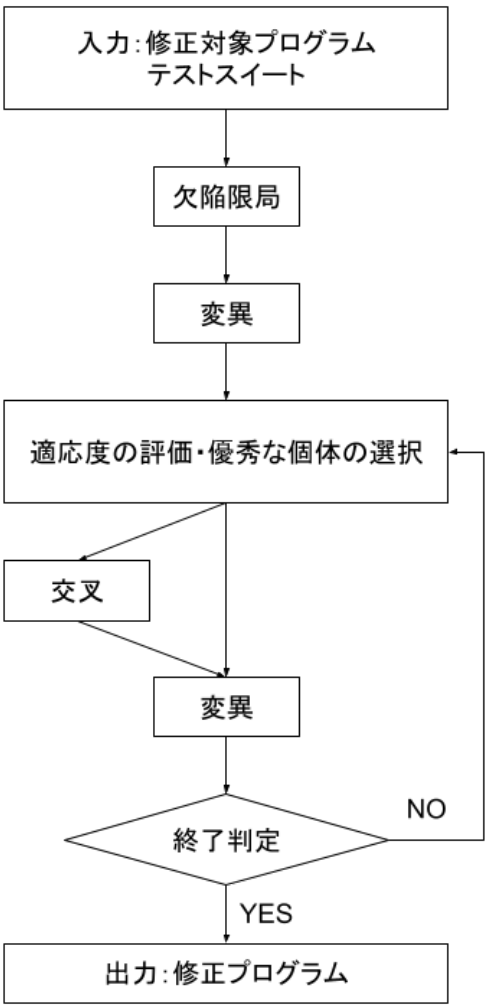

図 2.1 GenProg の動作の流れ

2.3 GenProg

まず，修正対象プログラムとテストスイートを用いて欠陥限局を行う [20]．欠陥限局とは，プログラム中の欠陥の位置を推測する手法である．GenProg では，以下の 3 種類にプログラムの各行を分類し，各行の疑惑値を計算する．なお，疑惑値は 0〜1 の範囲であり，1 に近づくほどその行に欠陥が存在する可能性が高くなる．

- 失敗したテストケースによって実行されない：疑惑値 = 0.0
- 失敗したテストケースによってのみ実行される：疑惑値 = 1.0
- 失敗したテストケースと合格したテストケースの両方で実行される：疑惑値 = 0.1

次に，修正対象プログラムの欠陥限局によって推測した位置に対して，変異の操作を行った個体群を生成する．変異とは，プログラムの欠陥箇所に変更を加え，新しい個体を生成する操作であり，GenProg では以下に挙げる処理のうち 1 つが行われる．なお，挿入や置換において，プログラムに対して挿入されるプログラム行は，修正対象プログラムに含まれる行から選択される．

挿入　欠陥箇所の次の行に修正対象プログラム中に存在しているプログラム行を挿入する処理

削除　欠陥箇所を削除する処理

置換　挿入と削除の両方を行う処理

次に，評価関数に基づいて各個体の適応度の評価を行う．適合度は個体と正しいプログラムとの近さを表しており，より多くのテストケースを通過する個体ほど適応度は高くなる．そして，個体群の中から適応度が高い優秀な個体を一定数選択する．選択された個体は，変異や交叉の対象となる．

次に，選択された個体に対して交叉および変異の操作を行い，次世代の個体群を生成する．交叉とは，2 つの個体から新しい個体を生成することであり，親となる個体の変更記録の一部を受け継いでいる．

最後に各個体にテストスイートを適用し，終了判定を行う．終了判定で終了条件を満たす

場合には GenProg を終了し，満たさない場合には，次世代の個体群に対して適応度の評価まで戻る．終了条件は以下に示す.

- 全てのテストケースを通過する個体を発見する
- 既定の世代数まで到達する

第 3 章

提案システム

本システムは，GenProg 及び Whisker を利用して作成しており，初学者が作成した Scratch プロジェクトを自動プログラム修正手法である GenProg を用いて自動的に修正することを目的としている．本システムでは，バグを含む Scratch プロジェクトとそのテストスイートを入力として受け取り，全てのテストケースに通過する Scratch プロジェクトが生成できた場合には出力する．

3.1 概要

本節では，Scratch プロジェクトを GenProg を利用して自動的に修正する手法について説明する．図 3.1 は提案手法の処理の流れである．入力は以下の 3 つである．

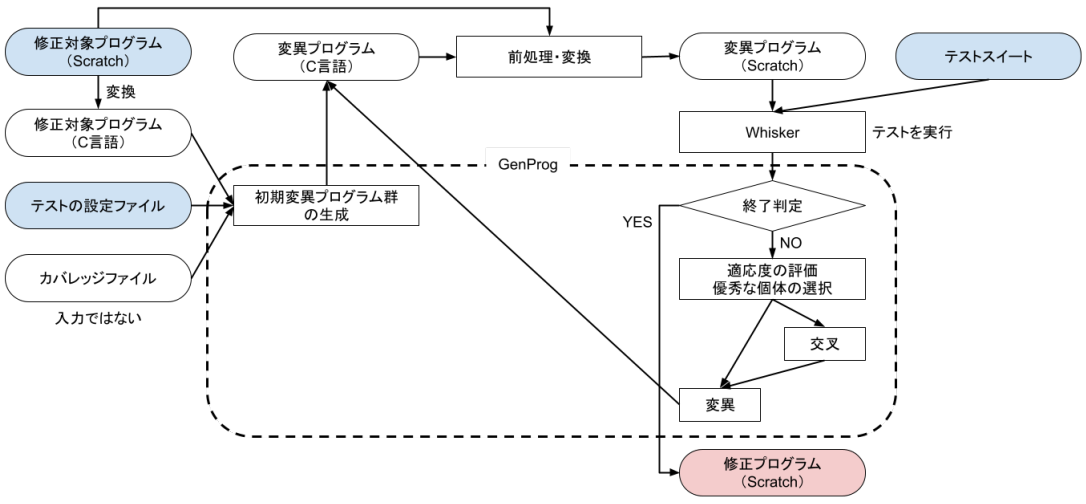

図 3.1　提案手法の処理の流れ（青は入力，赤は出力を表している）

- バグを含むプログラム（Scratch のプロジェクトファイル）
- テストに関する設定ファイル（テストプログラムのパスや修正対象プログラムのテスト結果を記述している）
- テストスイート

修正対象プログラムの修正に成功した場合，出力として全てのテストケースに通過する Scratch プロジェクトファイルが得られる．

提案手法は以下のステップで構成される．

Step1　Scratch プロジェクトから C ソースへの変換

Scratch のプロジェクトファイルから，そのブロック構造を表現した C 言語のソースファイルに変換する．このとき，Scratch のステージ及びスプライトに含まれるブロックを C ソースのどの関数で表現したのかを記録するファイルも生成する．このファイルは Step5 で C ソースから Scratch プロジェクトに逆変換する際に使用する．

Step2　カバレッジ情報の取得

カバレッジファイルを生成するための処理を行う．図 3.1 では，カバレッジファイルを生成する流れを省略しているが，入力として与えられたファイルではなく，システム内で生成されるファイルである．

Step1 で生成された C 言語のソースファイルと，入力として与えられたテストに関する設定ファイルを GenProg の入力として与える．テストに関する設定ファイルには，Whisker でテストを行う際に読み込む必要のあるテストファイルのパスや修正対象プログラムのテスト結果が記述されている．

欠陥限局を行うためにカバレッジ情報が必要となる．GenProg では，カバレッジ情報を収集するためのプログラムが生成され，そのプログラムを実行することによりカバレッジ情報を収集する．例えば，Listing3.1 のカバレッジを収集するために，Listing3.2 のようなプログラムが生成される．

本手法では C ソースコードを Scratch プロジェクトファイルに変換して実行するた

め，GenProg が生成するカバレッジ収集用のプログラムに関しても，Scratch プロジェクトに変換して Whisker 上で実行する．なお，C ソースコードを Scratch プロジェクトに変換する方法及びテストの実行に関しては，Step4〜6 で説明するためここでは省略する．

Listing 3.1 修正対象プログラム（C 言語）

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4 int a,b;
5 a = atoi(argv[1]);
6 b = atoi(argv[2]);
7 printf("%d\n", a-b);
8 return 0;
9 }
```

Listing 3.2 カバレッジ収集用のプログラム

```
1 extern __attribute__((__nothrow__)) void *( __attribute__((
      __nonnull__(1),
2 __leaf__)) memset)(void *__s , int __c , unsigned long __n ) ;
3 struct _IO_FILE ;
4 extern int fprintf(struct _IO_FILE * __restrict __stream ,
5                    char const * __restrict __format , ...) ;
6 extern struct _IO_FILE *fopen(char const * __restrict __filename ,
7                                char const * __restrict __modes ) ;
8 extern int fflush(struct _IO_FILE *__stream ) ;
9 extern int fclose(struct _IO_FILE *__stream ) ;
10 struct _IO_FILE *_coverage_fout ;
11 extern int ( /* missing proto */ atoi)() ;
12 extern int ( /* missing proto */ printf)() ;
13 int main(int argc , char **argv )
14 {
15   int a ;
16   int b ;
17
18   {
```

```
19  {
20  if (_coverage_fout == 0) {
21    {
22    _coverage_fout = fopen("/home/takahashi/genprog-code-main/test/
          gcd-test/./coverage.path", "wb");
23    }
24  }
25  }
26  {
27  fprintf(_coverage_fout, "1\n");
28  fflush(_coverage_fout);
29  }
30  a = atoi(*(argv + 1));
31  {
32  fprintf(_coverage_fout, "2\n");
33  fflush(_coverage_fout);
34  }
35  b = atoi(*(argv + 2));
36  {
37  fprintf(_coverage_fout, "3\n");
38  fflush(_coverage_fout);
39  }
40  printf("%d\n", a - b);
41  {
42  fprintf(_coverage_fout, "4\n");
43  fflush(_coverage_fout);
44  }
45  return (0);
46 }
47 }
```

Step3 初期変異プログラム群の生成

C 言語に変換された修正対象プログラム，テストに関する設定ファイル及び Step2 で生成したカバレッジファイルから，初期変異プログラム群を生成する．初期変異プログラムの生成は，GenProg で行う．

Step4 C から Scratch への変換を行うための前処理

Step3 または Step7 で生成される C 言語の変異プログラムを Scratch のプロジェクトファイルに変換するための前処理を行う．

GenProg によって生成される変異プログラムは，Step1 で生成するような Scratch のブロック構造を C 言語で表現するためのルールから反している．そのため，変異プログラムを Scratch プロジェクトに変換するための前処理として，Scratch のブロック構造を表現したソースコードに変換する．

Step5 C ソースから Scratch プロジェクトへの変換

Step4 によって前処理がされた変異プログラム，Step1 で生成したスプライトと関数の対応が書かれたファイル及び Scratch の修正対象プログラムを入力として受け取り，変異プログラムのブロック構造を持つ Scratch プロジェクトを生成する．

Step6 テストの実行

Scratch の自動テストフレームワークである Whisker を用いてテストを行う．Step5 で生成した Scratch プロジェクトファイルのパスとテストケースのパスを GenProg から Whisker に送信する．Whisker は GenProg から送られてきたパスに存在するファイルを読み込み，テストを実行して実行結果を GenProg に送信する．Whisker は読み込んだ Scratch プロジェクトがカバレッジ情報を収集するためのものであれば，収集したカバレッジ情報をファイルに出力する．

Step7 プログラム修正

Step6 で取得したテスト結果をもとに GenProg の終了判定，適応度の評価および優秀な個体の選択を行う．そして，選択された個体に対して交叉や変異の操作を行い，次世代の個体群を生成する．生成された次世代の個体は Step4 に送られる．終了判定において，全てのテストケースに通過する個体が存在した場合は，その個体を修正プログラムとして出力する．

3.2　実装

本システムでは，自動プログラム修正を行う対象を Scratch 3.0 のプロジェクトファイルとし，自動プログラム修正ツールは GenProg，自動テストフレームワークは Whisker を用いる．

以下に本システムにおける各ステップの実装について説明する．

Step1　Scratch プロジェクトから C ソースへの変換

Scratch 3.0 のプロジェクトファイルから，そのブロック構造を表現した C ソースファイルに変換する．変換用のプログラムは C++で実装している．

Listing3.3 は，図 3.2 のブロック構造を C ソースファイルに変換した結果である．図 3.3 のように，ほとんどのブロックは C 言語の関数呼び出しとして表現され，ブロックの引数に入れている変数やブロックなどは，関数呼び出しに与える引数として表現する．また，Scratch のブロックの繋がりを表現するために，連結されているブロックは C ソースでは同一の関数内にまとめる.「ずっと」「もし〜なら」ブロックのような繰り返しや条件分岐をするブロックに関しては，図 3.4 や図 3.5 のように while・if・else 文を用いて変換している．

変数やリストなどに関しては,「すべてのスプライト用」のものはグローバル変数として定義し,「このスプライトのみ」のものは，その変数を使用しているブロックが存在する関数内でローカル変数を定義する．メッセージはグローバル変数として定義する．

以下に Scratch のプロジェクトファイルを C 言語のソースファイルに変換する流れを示す.

1. Scratch プロジェクトファイルの拡張子を.sb3 から zip に変換し，展開する．展開したフォルダ内の project.json を読み込む．project.json には変数やブロック構造などが記述されている．
2. project.json のステージに関するデータを解析する．ステージで定義されている変数やリストなどはグローバル変数として C ソースで表現される．

3. project.json を解析し，連結されたブロック列ごとに関数を定義する．関数内での関数呼び出しの順番は Scratch ブロックが連結されている順番と同じになるようにする．各ブロックの引数を解析し，関数呼び出しの引数を設定する．条件分岐に関するブロックは C では if 文で表現する．繰り返しに関するブロックは，while 文の条件部分をブロックに対応した関数呼び出しにする．
4. Scratch のスプライト（またはステージ）と C ソースコードで定義した関数の対応を記述したファイルを出力する．

また，ソースコードの関数と Scratch のスプライトを対応付けるための JSON ファイルも生成する．この JSON ファイルは，Step5 で使用される．図 3.2 の変換時には以下のような JSON データが生成される．

$$\{\text{"}Stage\text{"} : null, \text{"スプライト 1"} : [0, 1]\} \tag{3.1}$$

図 3.2　Scratch スクリプトの例

Listing 3.3　図 3.2 を C ソースコードに変換したプログラム

```
1  void fun0_event_whenflagclicked(){
2  event_whenflagclicked();
3  motion_setx("0");
4  while(control_forever()){
5  motion_movesteps("10");
6  if(operator_gt(motion_xposition(),"100")){
7  control_stop("all");
8  }
9  }
10 }
11 void fun1_event_whenkeypressed(){
12 event_whenkeypressed("space");
13 motion_setx("0");
14 }
```

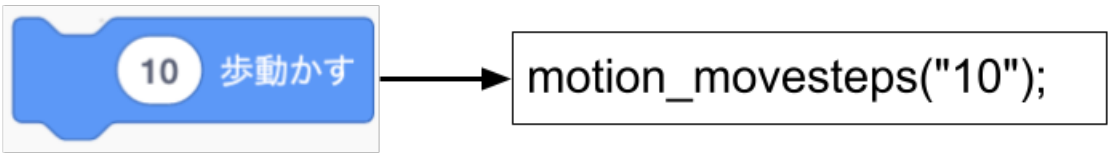

図 3.3　ブロックを C ソースに変換する例

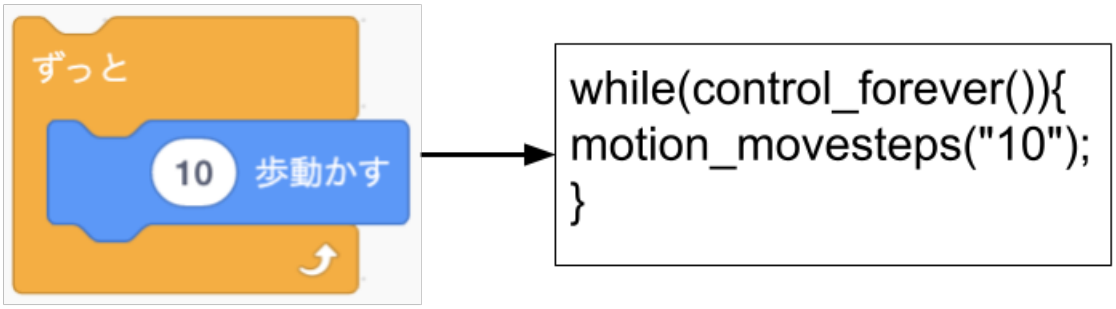

図 3.4　繰り返しブロックを C ソースに変換する例

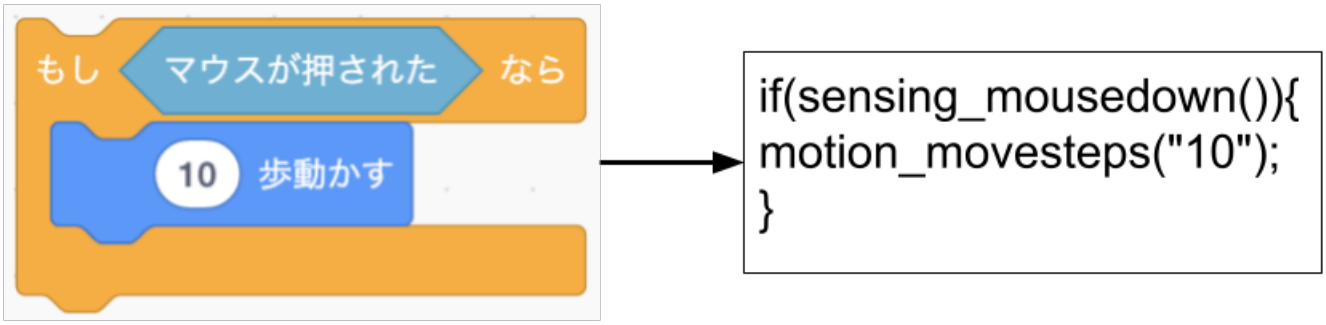

図 3.5　条件分岐ブロックを C ソースに変換する例

Step2　カバレッジ情報の取得

カバレッジ情報を取得するために，Whisker の scratch-vm モジュールに拡張機能としてログを収集する writeLog ブロックを実装する．writeLog ブロックが実行されると，引数として受け取った文字列を window オブジェクトの coverage プロパティに追記する（coverage プロパティの初期値は空文字列）．coverage プロパティの文字列は，Whisker でテストが終了したときに，coverage.path ファイルとして出力してブラウザからダウンロードする．そして，次にカバレッジを収集するときに備えて coverage プロパティの文字列を空文字列にする．なお，ダウンロードしたファイルを GenProg から利用するために，coverage.path の内容を GenProg がカバレッジファイルとして参照するファイルに追記する．

Step3　初期変異プログラム群の生成

GenProg を用いて初期変異プログラムを生成する．

Step4　C から Scratch への変換を行うための前処理

C から Scratch への変換を行うための前処理を行うプログラムは C++で実装した．Listing3.4 は GenProg によって生成された変異プログラムの例であるが，Scratch のブロック構造を C 言語で表現するためのルールから反している．そのため，Listing3.4 のような変異プログラムを Listing3.5 のような Scratch プロジェクトに変換しやすい表現に変換する．前処理では主に以下のことを行う．

- 空行，インデントや行末の不要なスペースを削除する．

- extern や return 文などの Scratch と無関係な行を削除する．
- 不要な括弧を削除する．
- fprintf 文をログを出力するための newblocks_writeLog 関数の呼び出しに変換する．
- Listing3.4 の 46–47 行目を Listing3.5 の 18 行目に変換しているように，tmp 変数に代入している関数呼び出しを，その後の関数呼び出しの引数として tmp を参照しているところに入れる．
- Listing3.4 の 20–26 行目のように，無限ループの開始直後に if 文によってループの終了条件を書いている部分を，while 文の条件部分に継続条件を書く表現に変換する．
- 一番上でしか使えないブロックの関数呼び出し（event_whenflagclicked や event_whenkeypressed など）が関数内の先頭以外で呼び出されている場合，関数内の先頭に移動する．また，複数存在している場合は，その中で一番上に書かれているもの以外削除する．

Listing 3.4 変異プログラムの例

```
1  char *score = (char *)"0";
2  extern int ( /* missing proto */ event_whenflagclicked)() ;
3  extern int ( /* missing proto */ data_setvariableto)() ;
4  extern int ( /* missing proto */ motion_gotoxy)() ;
5  extern int ( /* missing proto */ looks_show)() ;
6  extern int ( /* missing proto */ looks_sayforsecs)() ;
7  extern int ( /* missing proto */ looks_hide)() ;
8  extern int ( /* missing proto */ control_wait)() ;
9  extern int ( /* missing proto */ control_forever)() ;
10 void fun0_event_whenflagclicked(void)
11 {
12   int tmp ;
13
14   {
15   event_whenflagclicked();
16   data_setvariableto("0", score);
```

```
17    motion_gotoxy("0", "0");
18    looks_show("");
19    looks_sayforsecs("Click me to score points", "2");
20    while (1) {
21      tmp = control_forever();
22      if (tmp) {
23
24      } else {
25        break;
26      }
27      looks_hide();
28      control_wait("1");
29      looks_show();
30      control_wait("0.7");
31    }
32    return;
33  }
34  }
35  extern int ( /* missing proto */ event_whenthisspriteclicked)() ;
36  extern int ( /* missing proto */ data_changevariableby)() ;
37  extern int ( /* missing proto */ sound_play)() ;
38  extern int ( /* missing proto */ sound_sounds_menu)() ;
39  void fun1_event_whenthisspriteclicked(void)
40  {
41    int tmp ;
42
43    {
44    event_whenthisspriteclicked();
45    data_changevariableby("1", score);
46    tmp = sound_sounds_menu("zoop");
47    sound_play(tmp);
48    return;
49  }
50  }
```

Listing 3.5 Listing3.4 に前処理を行ったプログラム

```
1 string score = "0";
2 void fun0_event_whenflagclicked(void){
3 event_whenflagclicked();
4 data_setvariableto("0", score);
5 motion_gotoxy("0", "0");
6 looks_show("");
7 looks_sayforsecs("Click␣me␣to␣score␣points", "2");
8 while(control_forever()){
9 looks_hide();
10 control_wait("1");
11 looks_show();
12 control_wait("0.7");
13 }
14 }
15 void fun1_event_whenthisspriteclicked(void){
16 event_whenthisspriteclicked();
17 data_changevariableby("1", score);
18 sound_play(sound_sounds_menu("zoop"));
19 }
```

Step5 C ソースから Scratch プロジェクトへの変換

C ソースから Scratch プロジェクトに変換するプログラムは C++で実装した.

Scratch の変数やブロックなどは一意な ID と名前のセットで扱うため，C ソースを解析し，変数やリストなどは ID がキーで名前を値とする JSON データに変換する．C ソースの各関数定義は，Scratch の連結したブロック列に対応しているため，C ソースから Scratch プロジェクトに変換する際には，関数ごとに解析する．関数呼び出しを上から順番に読み，対応する Scratch ブロックの JSON データに変換する．また，各ブロックの JSON データに，そのブロックと連結している前後のブロック ID を入れる．更に，ブロックの引数となるデータも入れる．ブロックの引数がリテラルの場合は，その値を直接入れ，引数が変数やブロックの場合は ID も入れる．while や if 文の JSON データを生成する場合は，SUBSTACK プロパティに括弧内の一番上のブロック ID を入れる必要がある.

C ソースの各関数定義及び変数定義を JSON データに変換した後，Step1 で生成した Scratch のスプライトと C ソースの関数の対応が記述されたデータを読み，生成した JSON データを修正対象プログラムの対応する場所に入れる．

最後に，生成した JSON ファイル及び修正対象プロジェクトの画像や音声ファイルを同じフォルダに入れ，そのフォルダを圧縮して zip ファイルにし，生成した zip ファイルの拡張子を.sb3 にする．

Step6 テストの実行

Scratch の自動テストフレームワークである Whisker を用いてテストを行う．Whisker は Firefox 上で動作させる．GenProg と Whisker を接続するためのサーバプログラムを作成し，GenProg と Whisker はこのサーバを経由してデータのやり取りを行う．

テストを実行し，テスト結果を GenProg に送るまでの流れを以下に示す．

1. GenProg から Whisker に対して実行する Scratch プロジェクトとテストケースのパスを送信する．
2. Whisker が GenProg から送られてきたパスからファイルを読み込み，テストを実行する．
3. Whisker がテスト結果を GenProg に送信する．このとき，実行した Scratch プロジェクトがカバレッジを収集するためのものであれば，Whisker は収集したカバレッジ情報をファイルとして出力しダウンロードする．

Step7 プログラム修正 Step6 で取得したテスト結果をもとに GenProg でプログラムの修正を行う．

第 4 章

実験

4.1 実験方法

本研究では，以下の 2 つの実験を行う．

実験 1 Scratch の入門者用プロジェクト [*1] として紹介されている 3 つのプロジェクト（Pong Starter, Maze Starter, Hide and Seek）に対して，以下の変更を行いバグを生成し，本システムによって生成したバグの修正を試みる．なお，各プロジェクトに対して，変更内容ごとに 2 種類のバグプログラムを作成した計 24 個のプログラムで実験を行う．

- ランダムなブロックを 1 つ削除
- ランダムにブロックを 1 つ選択し，その下にランダムなブロックを追加
- ランダムなブロックを 1 つ選択し，ランダムなブロックに置換
- ランダムなブロックを 1 つ選択し，ランダムに選択した別のブロックの下に移動

実験 2 Scratch Web サイト [*2] にユーザが投稿している実験 1 で使用した入門者用プロジェクトを参考に作成したと思われるプロジェクトの中からランダムに選択し，本システムによってバグの修正を試みる．入門者用プロジェクトを参考に作成されたかどうかは，以下の要素を総合的に判断して決定する．なお，各入門者用プロジェクトに対して 3 つのユーザが作成したプロジェクトを選択し，実験を行う．

- プロジェクト名が酷似している

[*1] https://scratch.mit.edu/starter-projects

[*2] https://scratch.mit.edu/explore/projects/all

- 使用されている画像データが同じである
- スプライトの設定が酷似している
- プロジェクトの機能またはブロック構造が酷似している

4.2　実行環境

本実験は以下の実行環境の VirtualBox 上で実行した.

- OS：Windows 10 Home バージョン 21H2
- プロセッサ：Intel（R） Core（TM） i7-8700 CPU 3.70GHz
- メモリ：24.0GB
- VirtualBox：バージョン 6.1.40r154048（Qt5.6.2）
  - OS：Ubuntu 18.04.6 LTS
  - プロセッサー数：6
  - CPU 使用率制限：100%
  - メモリ：16384MB

4.3　実験結果

実験 1　表 4.1 は，実験 1 の結果である.「プロジェクト名」は変更を加えた入門者用プロジェクトの名前である.「変更内容」はバグを生成するためにプロジェクトに対して行った変更内容である．変更を加えたプロジェクトのブロック群に修正に必要となるブロックが存在している場合,「必要ブロックの存在」は O になる.

表 4.2 は，表 4.1 のデータから，変更内容ごとに修正率と平均実行時間を計算したものである．修正率は約 54.2%であり，平均実行時間は約 642.96 秒であった．また,「必要ブロックの存在」が O であるプロジェクトの修正率は 65.0%であった.

表 4.1　実験 1 の結果

プロジェクト名	変更内容	修正結果	必要ブロックの存在	実行時間（s）
Pong Starter	1 ブロック削除	X	O	1576.1
Pong Starter	1 ブロック削除	O	O	68.4
Pong Starter	1 ブロック追加	O	O	611.5
Pong Starter	1 ブロック追加	O	O	541.4
Pong Starter	1 ブロック置換	X	X	1356.0
Pong Starter	1 ブロック置換	O	O	682.0
Pong Starter	1 ブロック移動	X	O	1314.7
Pong Starter	1 ブロック移動	O	O	403.5
Maze Starter	1 ブロック削除	X	O	1123.8
Maze Starter	1 ブロック削除	X	O	901.1
Maze Starter	1 ブロック追加	O	O	414.4
Maze Starter	1 ブロック追加	O	O	83.3
Maze Starter	1 ブロック置換	O	O	446.7
Maze Starter	1 ブロック置換	X	O	1301.5
Maze Starter	1 ブロック移動	X	O	974.9
Maze Starter	1 ブロック移動	O	O	104.5
Hide and Seek	1 ブロック削除	X	X	623.9
Hide and Seek	1 ブロック削除	X	X	810.4
Hide and Seek	1 ブロック追加	O	O	155.6
Hide and Seek	1 ブロック追加	O	O	141.3
Hide and Seek	1 ブロック置換	X	X	728.8
Hide and Seek	1 ブロック置換	X	O	617.3
Hide and Seek	1 ブロック移動	O	O	316.7
Hide and Seek	1 ブロック移動	O	O	113.1

表 4.2　変更内容ごとの修正率及び修正時間

変更内容	修正率	実行時間（s）
1 ブロック削除	1/6	850.6
1 ブロック追加	6/6	324.6
1 ブロック置換	2/6	858.7
1 ブロック移動	4/6	537.9

実験 2　表 4.3 は，実験 2 の結果である.「入門者用プロジェクト」は，修正対象プログラムが参考にしたと思われる入力者用プロジェクトの名前である．修正率は約 44.4%であり，平均実行時間は約 628.89 秒であった．また,「必要ブロックの存在」が O であるプロジェクトの修正率は約 66.7%であった.

表 4.3　実験 2 の結果

入門者用プロジェクト	修正結果	必要ブロックの存在	実行時間（s）
Pong Starter	O	O	359.2
Pong Starter	X	X	1226.7
Pong Starter	X	X	1133.3
Maze Starter	O	O	93.4
Maze Starter	X	X	562.6
Maze Starter	X	O	489.7
Hide and Seek	O	O	760.5
Hide and Seek	O	O	503.4
Hide and Seek	X	O	531.1

4.4　評価

表 4.2 より，本システムは修正対象プログラムのブロックを削除または移動することにより修正可能であるバグに対して有用と考えられる．一方，修正対象プログラムにブロックを追加する修正やブロックを他のブロックに置換する必要のある修正では極端に修正率が低くなった．また，表 4.3 より，ユーザが作成したプロジェクトを修正することが可能であることが分かったが，修正率は約 44.4%と低かった．しかし，修正対象プログラム中に修正に必要なブロックが存在している場合の修正率は，実験 1 では 65.0%，実験 2 では約 66.7%であったことから，修正対象プログラム中に修正に必要なブロックが存在している場合において，本システムによる自動プログラム修正はある程度は有用と考えられる．また，本システムの平均実行時間は，実験 1 では約 642.96 秒であり，実験 2 では約 628.89 秒であったことから，教育者がフィードバック作業の補助として運用する場合には，作業を行う前日に本システムを実行するなどの工夫が必要と考えられる．

4.5　考察

本システムの修正率が低かった原因の 1 つとして，修正対象プログラム中に修正に必要なブロックが存在しないプロジェクトが存在することが挙げられる．GenProg の変異において，修正対象プログラムに挿入されるプログラムは，修正対象プログラムの中から選択される．そのため，修正対象プログラム中に修正に必要なブロックが存在しない場合，修正は不可能となる．

修正対象プログラム中に修正に必要なブロックが存在している場合でも，修正率が約 66.7%と低かった原因として，主に 2 つ考えられる．1 つ目は，本研究で行った実験の設定では 1 世代で生成できる個体数及び世代数の最大値が少ない可能性である．生成する個体を増やせば，様々な変更を試すことができるため，正しい変更を発見できる可能性も上がると考えられる．ただし，生成する個体数を増やすと実行時間が長くなるため，許容できる実行時間に応じて，1 世代に生成する個体数や世代数の最大値を設定する必要がある．2 つ目は，

GenProg の障害位置の推測があまり効果的ではない可能性である．図 4.1–4.3 は，修正に成功したプロジェクトの例であり，図 4.2 にバグが存在する．このプロジェクトを修正した結果，Ball スプライトは図 4.4 に修正された．修正前のプロジェクトでは，上向き矢印を押す必要のある全てのテストケースに失敗していたため，障害位置の疑惑値は 1.0 になる．一方，Ball スプライトのブロックが図 4.5 の場合，修正は失敗した．このプロジェクトは，図 4.5 の左下にある「左向き矢印キーが押されたとき」ブロックを「緑の旗が押されたとき」ブロックに変更することで修正できると考えられる．しかし，このプロジェクトは失敗したテストケースによって障害位置が実行されないため，疑惑値は 0.0 になる．このように，実行されるべきブロックが実行されないことによって発生するバグでは，疑惑値が 0.0 になる．

本研究では，入門者用プロジェクトを対象に実験を行なったため，テストの実行時間が長くはなかった．しかし，ゲームを作成する課題における,「全てのステージをクリアしたときにゲームクリア画面が表示されるかのテスト」のようなテストの実行時間が非常に長くなるテストも考えられる．このようなテストケースを用いて本システムで修正を行うと，変異プログラムが生成される度にテストを行うため，実行時間が非常に長くなる可能性がある．そのため，本システムをフィードバック作業の補助として使用する場合には，入門者に与える課題内容およびテストの内容について精査する必要があると考えられる．

4.5　考察

図 4.1　修正に成功したプロジェクトの例

図 4.2　修正に成功したプロジェクトの例（Ball スプライトのブロック）

図 4.3　修正に成功したプロジェクトの例（Goal スプライトのブロック）

図 4.4　図 4.2 の修正結果

図 4.5　修正に失敗したプロジェクトの例（Ball スプライトのブロック）

第 5 章

おわりに

5.1 まとめ

学習者が構文について意識することなるプログラムを作成できる Scratch が，コンピュータプログラミングの体験や学習する際の導入として，若い学習者を中心に利用されている．プログラミング学習において，学習者が作成したバグを含むプログラムに対して，教育者がフィードバックを返すことは教育上良いとされているが，学習者の人数が多くなると教育者の負担が大きくなる．よって，本研究では APR ツールである GenProg に Scratch 用テストフレームワークである Whisker を適用した Scratch 用 APR ツールの開発を行い，その有用性を評価した．その結果，修正対象プログラム中に修正に必要なブロックが存在している場合において，Scratch Web サイトに投稿されている 9 つプロジェクトのうち 4 つを平均約 628.89 秒で修正することができた．したがって，本システムによる自動プログラム修正はある程度の有用性を示したと言える．

5.2 今後の課題

今後の課題は，本システムによって実際に学習者が Scratch の課題を解く際に生まれたバグを修正することができるのかどうか評価することである．実験 1 で生成されたバグは機械的に作り出しているため，学習者が課題を解く際に自然に生まれるバグとは性質が異なる可能性がある．また，実験 2 についてもユーザは入門者用プロジェクトを参考にしてプログラムを作成しているため，参考にしたプロジェクトのブロック構造を知った上でプログラムを

作成している可能性がある．そのため，このバグも学習者が課題を解く際に自然に生まれるバグとは性質が異なる可能性がある．そのため，学習者が課題を解く過程で実際に生まれたバグに対して実験を行い，有用性を評価する必要がある．

本システムでは修正対象プログラム中に修正に必要なブロックが存在しない場合は，プログラムを修正することができなかった．そのため，教育者が課題で必要となりそうなブロックを GenProg に入力として与えることにより，修正対象プログラム中のブロックに加えて教育者によって与えられたブロックに関しても，GenProg の変異で挿入できるようにする方法が考えられる．ただし，挿入するブロックの選択肢が増えたことにより，正しいブロックが挿入される可能性が低下し，結果的に修正率が低下してしまう可能性がある．

また，GenProg の障害位置を推測する手法が Scratch の自動バグ修正において効果的でない可能性があるため，Scratch の自動バグ修正に有効な障害位置特定手法について検討したい．

最後に，本システムでは入力としてテストスイートを必要としている．しかし，小学校の教員がプログラミングやテスト設計に慣れていない可能性は高く，本システムを使用することが困難である可能性がある．そのため，プログラミングやテスト設計に慣れていない人でも簡単にテストスイートを作ることができるシステムが求められる．

謝辞

本研究を進めるにあたり，多大なご指導を賜りました指導教員の高田喜朗教授に心から感謝申し上げます．また，横山和俊教授と竹内聖悟先生より，貴重なご助言を賜りました．感謝申し上げます．最後に，所属する研究室のみなさまより多くのご支援をいただきました．お礼申し上げます．ありがとうございました．

参考文献

[1] 文部科学省, “小学校段階におけるプログラミング教育の在り方について（議論の取りまとめ）”, http://www.mext.go.jp/b_menu/shingi/chousa/shotou/122/attach/1372525.htm, 2023 年 1 月 27 日.

[2] 文部科学省, “小学校プログラミング教育に関する研修教材”, https://www.mext.go.jp/a_menu/shotou/zyouhou/detail/1416408.htm, 2023 年 1 月 27 日.

[3] Luisa Greifenstein, Florian Obermueller, Ewald Wasmeier, Ute Heuer, and Gordon Fraser, “Effects of Hints on Debugging Scratch Programs: An Empirical Study with Primary School Teachers in Training”, The 16th Workshop in Primary and Secondary Computing Education（WiPSCE）, No.3, pp.1–10, 2021, https://doi.org/10.1145/3481312.3481344.

[4] Luca Gazzola, Daniela Micucci, and Leonardo Mariani, “Automatic Software Repair: A Survey”, IEEE Transactions on Software Engineering, Vol.45, Issue.1, pp.34–67, 2019.

[5] Andreas Stahlbauer, Marvin Keris, and Gordon Fraser, “Testing Scratch Programs Automatically”, ESEC/FSE 2019: the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, p165–175, 2019.

[6] James Hoey, Ival Lanese, Naoki Nishida, Irek Ulidowski, and Germán Vidal, “A Case Study for Reversible Computing: Reversible Debugging of Concurrent Programs”, Reversible Computation: Extending Horizons of Computing, pp.108–127, 2020.

[7] Undo Software, “Increasing software development productivity with reversible de-

bugging”, https://undo.io/media/uploads/files/Undo_ReversibleDebugging_Whitepaper.pdf, 2023 年 1 月 29 日閲覧.

[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”, the 34th International Conference on Software Engineering, vol.38, no.1, pp.54–72, 2012.

[9] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest, “Automatically Finding Patches Using Genetic Programming”, the 31st International Conference on Software Engineering, pp.364-374, 2009, https://doi.org/10.1109/ICSE.2009.5070536.

[10] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra, “SemFix: Program Repair via Semantic Analysis”, International Conference on Software Engineering (ICSE) , pp.772–781, 2013, https://doi.org/10.1109/ICSE.2013.6606623.

[11] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim, “Automatic Patch Generation Learned from Human-Written Patches”, 35th International Conference on Software Engineering (ICSE) , pp.802–811, 2013, https://doi.org/10.1109/ICSE.2013.6606626.

[12] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus, “Automatic repair of buggy if conditions and missing preconditions with SMT”, the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis, pp.30–39, May 2014, https://doi.org/10.1145/2593735.2593740.

[13] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard, “Automatic error elimination by horizontal code transfer across multiple applications”, ACM SIGPLAN Notices, Vol.50, Issue.6, June 2015, pp.43–54, https://doi.org/10.1145/2813885.2737988.

[14] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer,
"Generating Fixes from Object Behavior Anomalies", IEEE International Conference on Automated Software Engineering (ASE) , pp500–554, 2009, https://doi.org/10.1109/ASE.2009.15.

[15] Thomas Ackling, Bradley Alexander, and Ian Grunert, "Evolving patches for software repair", the 13th annual conference on Genetic and evolutionary computation, pp.1427–1434, 2011, https://doi.org/10.1145/2001576.2001768.

[16] Fan Long, Martin Rinard, "Staged program repair with condition synthesis", Joint Meeting on Foundations of Software Engineering, pp.166–178, 2015, https://doi.org/10.1145/2786805.2786811.

[17] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang, "The strength of random search on automated program repair", the 36th International Conference on Software Engineering, pp.254–265, 2014, https://doi.org/10.1145/2568225.2568254.

[18] Pragya Agarwal, Arun Prakash Agrawal,
"Fault-localization techniques for software systems: a literature review", ACM SIGSOFT Software Engineering Notes, Vol.39, Issue.5, pp.1–8, September 2014, https://doi.org/10.1145/2659118.2659125.

[19] John R. Koza, "Genetic programming as a means for programming computers by natural selection", Statistics and Computing, Vol.4, pp.87–112, 1994.

[20] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan L.C van Gemund, "A Practical Evaluation of Spectrum-based Fault Localization", Journal of Systems and Software, Vol.82, Issue.11, pp.1780–1792, 2009.