実行履歴の比較を用いた学習者向けデバッグ支援方法の提案
―繰り返しプログラムを対象として―

2017SE042 丸井優花　2017SE100 渡辺実那
指導教員：蜂巣吉成

1　はじめに

プログラミング学習において，プログラムのコンパイルは成功するが意図していない実行結果になる場合がある．このとき，学習者はソースコードを読んで実行の流れを理解しようとするが，動作理解が不十分な学習者はこの方法でフォールト箇所を特定することは難しい．printf デバッグやデバッガ，実行履歴を表で出力する動作理解支援ツールといった既存の方法も，printf やブレークポイントの設定，繰り返しプログラムで用いると実行ステップや表の数が増えてしまうという問題があり，学習者にとって負担が大きくフォールト箇所の特定が難しい．

本研究は，繰り返しプログラムに対して模範解答プログラムと学習者プログラムの実行履歴を比較する学習者のデバッグ支援方法を提案する．模範解答と学習者プログラムの実行履歴を比較することで学習者プログラムのフォールト箇所を特定する．まず模範解答と学習者プログラムのソースコードを比較して，対応する箇所を探す．構造が異なる 2 つのプログラムをどう対応づけるかが課題となるが，本研究は最初に大まかに分け，そこで対応付かなかったものは細かい条件で分けるという段階的な分類で解決する．具体的には複数の評価関数を用いて対応付けを行う．対応付けができたら，模範解答や学習者プログラムの実行履歴，すなわち，プログラム実行時に実行された文と変数の値を取得し，ソースコードで対応付いた箇所を比較する．異なる箇所にはフォールトが含まれている可能性が高いので，その箇所を学習者に表示する．

提案方法により，実行履歴のどこに着目したらよいかを体験的に学ぶことで，学習者が独力でフォールト箇所を特定できるようになることが期待される．学習用以外の一般的なプログラムについても，リファクタリングやバージョンアップの場面で本研究を適用できると考えられる．例えば，リファクタリングで意図した動作をしないとき，コードを書き換える前のプログラムを模範解答，書き換えた後のプログラムを学習者プログラムとして置き換えれば本研究の手法でフォールト箇所を見つけることができる．

2　関連研究

長谷川らは，プログラムの動作を可視化するために，ソースコードの変数を取得して変数変化や条件文の真偽を表で表示している [1]．1 つの繰り返しにつきソースコードと条件式の真偽，各変数の値が 1 行ずつ書かれた表が表示されることにより，注目したい行と他の行の比較が容易になる．このツールを通してプログラムの誤り発見やデバッガの利便性についても理解することができる．[1] の研究はプログラム全体の動作理解支援として有効であるが，繰り返し回数が多くなると表の数が増え，どこに着目すればよいのかわかりにくいという問題がある．

Spectrum-Based Fault Localization はテストケースによる実行経路情報を利用するフォルトローカライゼーション技術である [2]．テストケース実行によるテスト結果と実行経路情報から各ステートメントの疑惑値を計算することで疑惑値が高い箇所を不具合の原因箇所と推定することができる．しかし，ソースコードをフォールト箇所として出すと学習者は機械的に提示された箇所を訂正し，動作理解の支援に繋がらず，学習向けの方法ではない．

Relative Debugging はソフトウェアのバージョンアップや移植などにおいて，期待された出力をするプログラムと誤った出力をするプログラムを変数の値の違いに注目して比較し，デバッグを行う方法である [3][4]．開発者は 2 つのプログラムで比較する変数や箇所を指定して，2 つのプログラムの差を利用してデバッグしていく．しかし，比較する変数や箇所を指定させるので学習者に模範解答を渡す必要があり，その模範解答を学習者が写してしまう可能性があるという点で学習の場面においては有効な手段ではない．また，構造が異なる 2 つのプログラムの対応付けも課題である．

3　問題分析

ソースコード 1 は文字列から任意の文字を削除するプログラムの一部である．このプログラムのフォールトは，12 行目の else が抜けている点である．例えば文字列 abcbbcba から b を削除する場合，4 行目にブレークポイントを置くと，1 文字目は 24 回の実行ステップが必要となり，学習者にとって負担となる．

ソースコード 1　文字列 s から文字 c を削除する関数

```
 1 void strdel(char *s, char c)
 2 {
 3     char *t;
 4     while(*s) {
 5         if (*s==c) {
 6             t=s;
 7             while(*t) {
 8                 *t=*(t+1);
 9                 t++;
10             }
11         }
12         s++;
13     }
14 }
```

4　実行履歴の比較を用いたデバッグ支援方法の提案

本研究では，模範解答と学習者プログラムの実行履歴を比較し，実行履歴が異なっている箇所を特定して表形式で

表示する．実行履歴を比較する流れを以下に示す．

1. 模範解答と学習者プログラムの対応付け
2. 実行履歴の抽出
3. 実行履歴の比較
4. 比較結果の出力

前提条件

本研究はプログラミング学習における模範解答が存在する演習問題を対象とする．演習問題の main 関数は指導者によって記述済みであり，学習者は事前に指導者が型や変数名，引数を用意した関数の内部を記述する．この場合，関数内部で学習者が定義する局所変数は模範解答と一致すると限らない．そこで，石元の変数名の置き換え [5] を利用し，変数の対応付けを行う．例えばソースコード 1 では，模範解答と学習者プログラムの s 同士，c 同士を石元の研究を用いて対応付ける．しかし，ソースコード 1 で定義されている変数 t は役割が模範解答と異なる．この場合は変数 t の対応付けをせず，変数 t を用いた文の対応付けや実行履歴の比較も行わない．

本研究は繰り返しプログラムを対象としているので，学習者が記述する関数と模範解答双方に繰り返し構文が少なくとも 1 つは含まれているとする．繰り返しの対応付けについては for 文と for 文というように同じ種類の繰り返し文の組み合わせに対応しており，for 文と while 文といった異なる種類の繰り返し文は対応付けない．

実行履歴は田中・戸田の研究 [6] により取得する．対応付いた命令文が模範解答と学習者プログラムのそれぞれ何行目に記述されているかという情報を田中・戸田の研究で作成されたツールに渡すことで実行履歴を取得する．

取得する実行履歴について，仮引数がポインタの場合，取得する値はポインタの指す先の値とする．ポインタの指し先が文字列のときは文字列を取得する．アドレスはプログラムによって異なり模範解答と比較する意味がないので取得しない．

4.1 模範解答と学習者プログラムの対応付け

模範解答と学習者プログラムの実行履歴を比較するために，繰り返し文と繰り返し内の文でそれぞれ役割が一致するものを対応付ける．対応付けには評価関数を用いる．

ソースコードの対応付けは模範解答と学習者プログラムで繰り返し構文の数や繰り返し内部の命令文の違いがあり単純に同じ行を結びつけることができない．本研究は最初に大まかな分け方をしてそこで対応付かなかったものは細かい条件で分けるという段階的な分類方法でこの課題を解決する．評価関数を用いた対応付けのアルゴリズムは複数の評価関数を用意し，評価関数 1 から対応付けしてその評価関数で対応付かなかった場合は次の評価関数で再帰的に対応付けをする．

繰り返しの対応付けで用いる評価関数は 3 つとする．繰り返しの分類は基本的に条件文がどれだけ一致しているかで判断するので，共通の変数と演算子が分かれば良い．本研究は先に大まかな分類をしてそのあとに細かな分類をするというアプローチであるので，変数は模範解答と別の変数を定義している可能性が低く模範解答との共通箇所が最も多いということから大まかな分類は変数を使い (評価関数 1)，細かな分類は演算子を使う (評価関数 2)．このとき，2 つの評価関数では振り分けられないものが出てくる場合があるので，その場合は出現順で分類する (評価関数 3)．

以上から，模範解答の繰り返しの集合を a，集合 a の添字を i，学習者プログラムの繰り返しの集合を x，集合 x の添字を j として，評価関数を次のように定義する．添字はソースコード中の出現順序を表す．

評価関数 $1(a_i, x_j)$ = 繰返し a_i，x_j に出現する共通の変数の個数

評価関数 $2(a_i, x_j)$ = 繰返し a_i，x_j に出現する共通の変数に対する演算の一致する個数

評価関数 $3(a_i, x_j)$ = i と j が等しいかどうか (e.g. 出現順が同じ)

繰り返し内の文の対応付けで用いる評価関数は 3 つとする．評価関数 1 で共通の変数での大まかな分類を行うことは繰り返しの対応付けと変わらない．細かな分類は繰り返しのように演算子で分類すると，演算子が一致しても右辺が一致しない場合に模範解答と学習者プログラムでその文の役割が異なる可能性がある．例えば，$a = b + 1$ と $a = c + 2$ は演算子は一致するが右辺の変数及び数字が全く一致しないので，この 2 式の役割は異なる可能性が高い．そこで，繰り返し内の文の対応付けについては，共通の変数の個数で対応付かなかった場合，演算子と右辺両方で分類を行う (評価関数 2)．このとき，2 つの評価関数では振り分けられないものが出てくる場合があるので，その場合は出現順で分類する (評価関数 3)．

以上から，模範解答の繰り返し内の文の集合を a，模範解答の繰り返しがいくつ目かを表す添字を i，学習者プログラムの繰り返し内の文の集合を x，模範解答の繰り返しがいくつ目かを表す添字を j とすると，繰り返し内の文の評価関数は次のように定義できる．

評価関数 $1(a_i, x_j)$ = 文 a_i，x_j 中に出現する共通の変数の個数

評価関数 $2(a_i, x_j)$ = 文 a_i，x_j 中に出現する共通の変数に対する演算子と右辺の変数・数字が一致する個数

評価関数 $3(a_i, x_j)$ = i と j が等しいかどうか (e.g. 出現順が同じ)

ここで，ソースコード 1 を例にして対応付けを行う．ソースコード 2 はソースコード 1 の模範解答である．

ソースコード 2 ソースコード 1 の模範解答

```
1 void strdel(char *s, char c)
2 {
3     char *t;
4     t=s;
5     while(*s!='\0'){
6         if(*s!=c){
7             *t=*s;
```

```
8               t++;
9           }
10          s++;
11      }
12      *t='\0';
13  }
```

はじめに繰り返しの対応付けを行うと，評価関数 1 で模範解答の 5 行目と学習者プログラムの 4 行目に記述されている繰り返し同士が対応付く．次に，繰り返し内の文の対応付けを行うと，評価関数 1 で模範解答の 6 行目と学習者プログラムの 5 行目，模範解答の 10 行目と学習者プログラムの 12 行目が対応付く．よって，ソースコード 1 とソースコード 2 は図 1 のように対応付く．

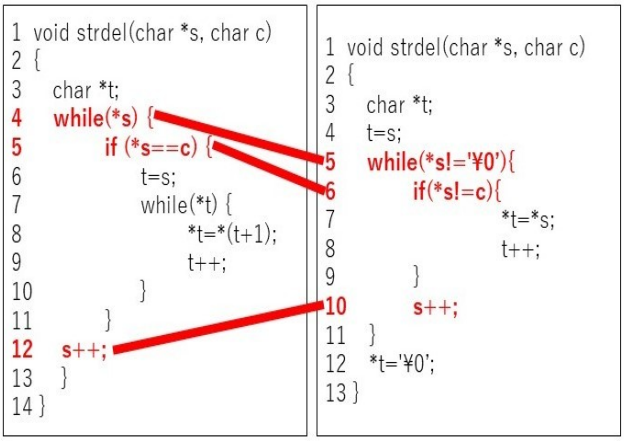

図 1　ソースコード 1 とソースコード 2 の対応付け結果

4.2　実行履歴

フォールトの存在する関数を実行履歴を取得したい関数とし，その関数を f とする．まず，関数 f の実行された文の実行情報を文を実行後の関数 f の仮引数と局所変数の変数名と値の組の集合とする．関数 f の実行履歴は， 関数 f の実行された文の行番号とその文の実行情報の組の列と定義する．ソースコード 1 の実行履歴は次のようになる．ソースコード 1 は文字列 s を”abcbbcba”として削除する文字 c を’b’とする．

ソースコード 1 の実行履歴の一部

```
(4, {(s, " abcbbcba" ), (c, ' b' )})
(5, {(s, " abcbbcba" ), (c, ' b' )})
(12, {(s, " bcbbcba" ), (c, ' b' )})
(4, {(s, " bcbbcba" ), (c, ' b' )})
(5, {(s, " bcbbcba" ), (c, ' b' )})
(12, {(s, " bbcba" ), (c, ' b' )})
...
(4, {(s, "" ),(c, ' b' )})
```

4.3　実行履歴の抽出

実行履歴を比較するときにソースコードが対応づいている箇所の実行履歴のみを用いるために実行履歴の抽出を行う．関数 f の実行履歴の中で関数 f’ と対応づいた文の実行情報を抽出し，関数 f’ と対応づいていない変数とその値の組を削除したものを関数 f’ と対応づいた関数 f の実行履歴と定義する．この定義に従い，模範解答の関数 f_{answer} と対応づいた学習者の関数 $f_{learner}$ の実行履歴 $h_{learner}$ と学習者の関数 $f_{learner}$ と対応づいた模範解答の関数 f_{answer} の実行履歴 h_{answer} を求める．ソースコード 1，2 の抽出した実行履歴の一部を以下に示す．

文字列削除の学習者の実行履歴 $h_{learner}$ の一部

```
(4, {(s, " abcbbcba" ), (c, ' b' )})
(5, {(s, " abcbbcba" ), (c, ' b' )})
(12, {(s, " bcbbcba" ), (c, ' b' )})
(4, {(s, " bcbbcba" ), (c, ' b' )})
(5, {(s, " bcbbcba" ), (c, ' b' )})
(12, {(s, " bbcba" ), (c, ' b' )})
...
(4, {(s, "" ),(c, ' b' )})
```

文字列削除の模範解答の実行履歴 h_{answer} の一部

```
(5, {(s, " abcbbcba" ), (c, ' b' )})
(6, {(s, " abcbbcba" ), (c, ' b' )})
(10, {(s, " bcbbcba" ), (c, ' b' )})
(5, {(s, " bcbbcba" ), (c, ' b' )})
(6, {(s, " bcbbcba" ), (c, ' b' )})
(10, {(s, " cbbcba" ), (c, ' b' )})
...
(5, {(s ,"" ), (c, ' b' )})
```

4.4　実行履歴の比較

実行履歴の比較をするために $h_{learner}$ と h_{answer} の差分を求める．考えられる差分を以下に示す．

不足　— h_{answer} において実行された文が $h_{learner}$ にない (i.e. 学習者の繰返しが少ない)

余剰　— $h_{learner}$ において実行された文が h_{answer} にない (i.e. 学習者の繰返しが多い)

変更　— $h_{learner}$ と h_{answer} で共通に実行された文の変数の値が異なる (i.e. その文がフォールトの原因の可能性がある)

ソースコード 1 では，文字列 s の値が異なる箇所があることがわかる．この場合は考えられる差分の 3 つ目に該当する．繰り返し回数についても着目すると模範解答よりも少ないので考えられる差分の 1 つ目にも該当する．このソースコードにおける違いをフォールトと推測する．

4.5　比較結果の出力

推測されたフォールトをもとに比較結果を表で出力する．図 2 はソースコード 1 の比較結果の出力である．

この場合は文字列 s の値が異なるので該当箇所を赤文字で強調し，さらに繰り返し回数が模範解答と異なるのでメッセージを出力している．この例のようにメッセージ出力や赤文字でフォールトと推測される箇所を強調することで動作理解を促し，支援する．

5　考察

リファクタリングやバージョンアップの場面で本研究の手法を応用できると考えられる．リファクタリングではコードの書き換え後に意図した動作をしないとき，フォールトのある関数を把握できていれば，コードの書き換える

	ソースコード	s	c
4	while(*s){	"abcbbcba"	'b'
5	if(*s==c)	"abcbbcba"	'b'
12	s++;	"bcbbcba"	'b'
4	while(*s){	"bcbbcba"	'b'
5	if(*s==c)	"bcbbcba"	'b'
12	s++;	"bbcba"	'b'

.
.
.

	ソースコード	s	c
4	while(*s){	""	'b'

繰り返し回数が少ないです

図 2　ソースコード 1 の出力結果の一部

	ソースコード	s	c
5	while (*s) {	“nanzan”	‘a’
6	if (*s != c) {	“nanzan”	‘a’
9	s++;	“anzan”	‘a’
5	while (*s) {	“nzan”	‘a’
6	if (*s != c) {	“nzan”	‘a’
9	s++;	“zan”	‘a’

.
.
.

	ソースコード	s	c
5	while (*s) {	“”	‘a’

繰り返し回数が少ないです

図 3　ソースコード 4 の出力結果の一部

前のプログラムを模範解答，書き換えた後のプログラムを学習者プログラムとして置き換えてフォールト箇所を見つけることができる．例えば，ソースコード 3 をリファクタリング前のプログラムとする．

ソースコード 3　リファクタリング前のプログラムの一部

```
1  void strdel(char *s, char c)
2  {
3    char *t;
4
5    while (*s) {
6      if (*s == c) {
7        t = s;
8        while (*t) {
9          *t = *(t+1);
10         t++;
11       }
12     } else {
13       s++;
14     }
15   }
16 }
```

ソースコード 3 は二重ループとなっており，これを単一ループへ変更するようリファクタリングしたプログラムをソースコード 4 とする．

ソースコード 4　リファクタリング後のプログラムの一部

```
1  void strdel(char *s, char c)
2  {
3      char *t;
4      t = s;
5      while (*s) {
6        if (*s != c) {
7            *t = *s;
8            t++;
9            s++;
10       }
11       s++;
12     }
13     *t = '\0';
14 }
```

しかし，ソースコード 4 を実行すると削除したい文字以外の文字も一部削除されてしまう．このとき，ソースコード 3 を模範解答，ソースコード 4 を学習者プログラムと置き換えて本研究の手法を用いると，比較結果は図 3 のようになる．

今回は評価関数 3 でソースコード 3 の 13 行目とソースコード 4 の 9 行目が対応付いているが，ソースコード 4 のように複数の同じ命令文がある場合，評価関数 3 での対応付けでは精度が低いので，今後対応付けの精度を高める必要がある．実行履歴の変数の値を比較して模範解答と一致している回数が多い方を対応付けるという方法が挙げられる．

6　おわりに

本研究では，実行履歴を用いた初学者向けデバッグ支援方法を提案した．模範解答と学習者プログラムをソースコードで対応付け，対応付けに基づき実行履歴を抽出・比較することで，フォールト箇所を特定できると考えた．また，フォールト箇所を強調して出力することで，学習者に動作理解をさせ，printf 文やブレークポイントの挿入箇所について学べるようにした．今後の課題としては，提案に基づくツールの実現を行う必要がある．

参考文献

[1] 長谷川洸也，川地周作：“命令型プログラミングにおける動作理解支援に関する研究”, 南山大学情報理工学部 2014 年度卒業論文 （2015）.

[2] W. E. Wong, R. Gao, et al：“A Survey on Software Fault Localization”, IEEE Trans. on Software Engineering, Vol. 42, No. 8, pp. 707-740 （2016）.

[3] D. Abramson, I. Foster, J. Michalakes, R. Sosic：“Relative debugging and its application to the development of large numerical models”, Proc. of IEEE Supercomputing （1995）.

[4] D. Abramson, C. Chu, D. Kurniawan, A Searle：“Relative debugging in an integrated development environment”, Software - Practice and Experience, Vol. 39, Issue 14, pp. 1157–1183 （2009）.

[5] 石元慎太郎：“プログラミング学習者の編集途中のソースコードに対するフィードバック方法の提案”, 南山大学大学院理工学研究科 2019 年度修士論文 （2020）.

[6] 田中裕人，戸田順也：“学習者向けの自己参照構造体の動作理解支援ツールの作成のためのプラットフォームの提案—データ構造の操作をする C プログラムを題材として—”, 南山大学理工学部 2020 年度卒業論文 （2021）.