修正前後のプログラムの差分に基づく書換えパターン生成についての研究
―書換えパターンの記述範囲決定の支援―

2009SE015 朝岡百合子　2009SE096 神谷優希
指導教員：吉田敦

1　はじめに

ソフトウェア開発では，既存の処理と同じような処理を記述するときに，既存の記述をコピーするなど，ほぼ同じ記述をそのまま書くことがある．その際に複数行にわたる類似した記述であるコードクローン [3] や，コードクローンの条件には満たないが，重複した内容を持つ定型的な記述などが発生する．それらの記述に変更を加える場合，すべての変更箇所に同様の書換え作業が必要となる．このとき，変更箇所の見落としがあれば変更漏れが発生したり，単純作業の繰り返しの修正で記述ミスが起こる可能性がある．それらの人為的なミスを防ぐにはパターンによる書換えが効果的である．しかし，書換えパターンは，書換えを行いたい箇所の書換え漏れがなく，かつ，書換えを行わない箇所の記述を含まないように作る必要がある．部分的な抽象化や，書換え箇所の前後に出現する記述の共通性に基づいた制約を記述するなど，人の手で生成するには手間がかかる．

本研究では，修正前と後の 2 つのプログラムの差分を用いた書換えパターンの生成支援を提案する．具体的には，ユーザがプログラムの一箇所だけを修正し，修正前後の差分から書換えパターンの雛形を生成する．さらに，ユーザに指示を求め，補正を行う．雛形はユーザが必要に応じて編集することを前提とする．書換えパターン生成の際に，パターンを直接記述するよりも実際にプログラムを書換える方が書換え作業を具体的に考えやすく，書換え後の正しさも検証しやすい．なお，本研究では，プログラム書換え系として TEBA[5] を利用し，C 言語のプログラムを書換え対象とする．

複数の修正対象の間で異なる記述を吸収する方法として，パターン変数などを用いた一般化が必要である．しかし，一般化によって，書換えない箇所もパターンに適合する可能性がある．そこで，一般的には，同一の処理の前後に共通の記述が出現することから，書換え箇所の記述に加え書換え箇所の前後に出現する記述も含めて書換パターンを生成することで適用範囲を絞る．そのために，実際に書換パターンを適用して書換えられた箇所から正しい書換え箇所をユーザに選択させ，それらの前後に出現する共通する記述をパターンに追加する．なお，ユーザに選択させる理由は，正しい箇所を自動的に判定することが難しいことにある．

技術的な課題は，書換えパターンに追加する記述の範囲の決定方法と，その記述範囲の差異を吸収する方法を明らかにすることである．

```
%before
if ((${v:ID_VAR} = ${cast:EXPR}malloc(${size:EXPR})) == NULL)
${stmt:STMT}$;
%after
${v} = ${cast}my_malloc(${size});
%end
```

図 1　書換えパターンの例 [5]

```
void test (void)
{
  char *p;
  if ((p = (char *)malloc(sizeof(char) * 100)) == NULL){
    fprintf(stdrr, "no memory!\n");
    exit(1);
  }
  proc(p);
  free(p);
}
```

図 2　TEBA を用いた書換えの入力例

2　プログラム書換え系ツール

本研究ではプログラムの書換えに TEBA を用いる．TEBA を用いると編集中のテキストから書換え対象の断片をコピーすることで書換え作業を記述でき，抽象構文木などの他の表現に写像する必要がない．それにより，書換えパターンの生成支援ツールの実装が簡単になる．

図 1 は TEBA の書換えパターン，図 2 と図 3 は図 1 を使用した書換えの入力と出力を示す．TEBA の書換えパターンでは書換え前を表現する記述を「変換前パターン」，書換え後を表現する記述を「変換後断片」と呼ぶ．書換えパターンは図 1 のように`%before` の直後に変換前パターンを，`%after` の直後に変換後断片を記述する．パターン変数は可変部分となる箇所に記述し，一般化をする．変換前パターンでは “`${名前:型}`”，また変換後断片では “`${名前}`” と記述する．型は適合する構文要素を表し，字句の種別か，ユーザが定義する字句パターンの名前を用いる．

3　書換えパターンの生成方法

3.1　書換えパターンの生成方法の全体の流れ

書換えパターン生成方法全体の流れを図 4 に示す．入力として修正前および修正後プログラムを与えると，雛

```
void test (void)
{
  char *p;
  p =  (char *)my_malloc(sizeof(char) * 100);
  proc(p);
  free(p);
}
```

図 3　TEBA を用いた書換えの出力例

形となる「初期パターン」を生成する．初期パターンは，差が生じた範囲だけを切り出し，目的となる他の箇所にも適用するよう手作業で記述の一部を一般化したものである．修正前プログラムにその書換えパターンを適用し，その結果を差分としてユーザに提示する．書換え対象外の箇所が書き変わっていたら，書換え対象の箇所をユーザが指定する．その情報を元に再度書換えパターンを補正し，再び書換え結果をユーザに提示する．ユーザが目的のパターンを得られるまで，これを繰り返す．

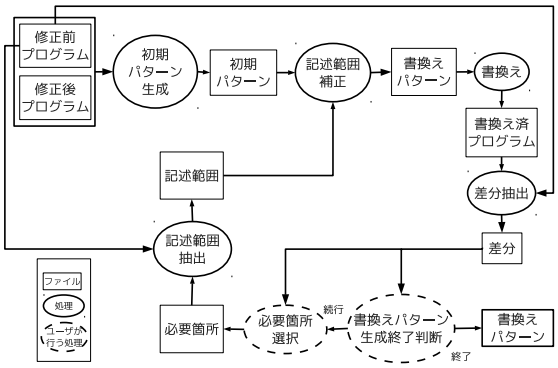

図 4 書換えパターン生成方法全体の流れ

3.2 初期パターン生成の詳細

図 5 に図 4 の初期パターン生成の詳細を示す．プログラムの修正前後の差分から，差異が生じた範囲の断片をそのまま記述したパターンを生成する．次に，他の書換えたい箇所に適用するために，手作業で記述の一部をパターン変数に置き換える一般化を行い，初期パターンを生成する．

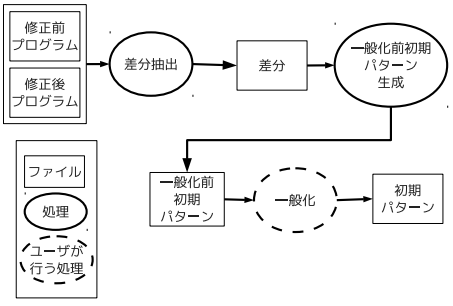

図 5 図 4 の初期パターン生成の詳細

3.3 書換えパターンに記述する範囲

書換えパターンに記述する修正前後のプログラムの範囲を「記述範囲」と呼ぶ．差分から，直接変更した箇所と前後を含めた共通部分を特定し，記述範囲を決定する．

4 書換えパターンの記述範囲決定について

4.1 記述範囲決定の処理について

書換えパターンが適用された箇所の中で，書換えの目的の箇所を「書換え必要箇所」，それ以外の箇所を「書換え不要箇所」と呼ぶ．書換え必要箇所の前後で共通する字句を持つ範囲を「共通範囲」と呼ぶ．

図 6 に図 4 の必要箇所選択後のパターン生成処理の詳細を示す．初期パターンによる書換えを適用したプログラムと修正前プログラムの差分からユーザは書換えられた箇所を確認し，必要箇所を選択する．必要箇所を入力すると，記述範囲を出力する．差分として抽出された字句をすべて含む構文要素を書換えパターンに記述し，その後，書換え必要箇所の前後に出現する記述の種別の並びを比較し，記述範囲を決定する．

書換え必要箇所の前後に出現する記述を TEBA を用いて解析し，属性付き字句系列から種別の並びを比較をする．属性付き字句系列とは，抽象構文木を字句系列で表現したもので，すべての字句は種別属性を持ち，その値は字句解析の字句定義で与えられる名前である．種別で比較をすることで識別子などの差異を吸収する．また，図 6 の共通範囲抽出では記述範囲に必要な部分の記述の差異を吸収するために一般化を行う．

図 7 に必要箇所の共通範囲を用いた記述範囲決定の流れを示す．選択された書換え必要箇所の前後に出現する記述を抽出し，任意の 2 つの書換え必要箇所の前後に出現する字句の種別の並びを比較，共通範囲を抽出し記述範囲とする．書換え必要箇所が 3 つ以上ある場合，すでに抽出された共通範囲と共通範囲抽出を行なっていない任意の 1 つの書換え必要箇所を比較し共通範囲を求める．その共通範囲が，すでに抽出した共通範囲より小さい場合，新しく求めた共通範囲を全箇所に対する共通範囲として選択する．この選択をすべての書換え必要箇所に対して繰り返し，共通範囲を決定する．なお，提案手法では必ずしも正確に書換え必要箇所だけに限定できないので，不必要な書換えについては手作業で戻せるよう，書換え前の状態を前処理命令の`#ifdef` で囲って残す．

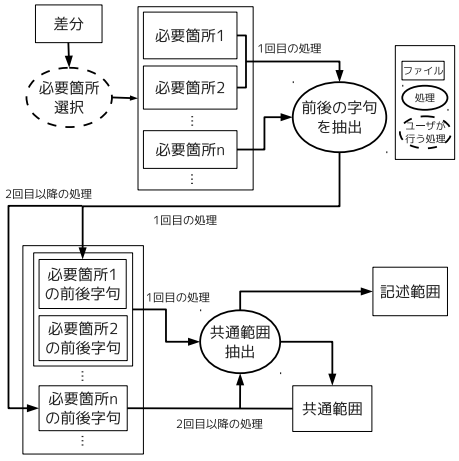

図 6 必要箇所選択以降の処理の手順

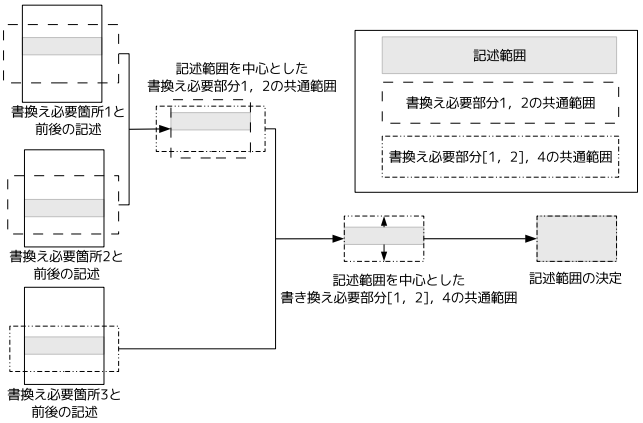

図 7 書換え必要箇所の共通範囲を用いた記述範囲決定の流れ

```
…
/*必要な記述範囲　開始*/
    if(option == 'p')
      col = get_col (f_col);//この関数内の引数が異なる
    if (col == NULL)
      {
        error (0, errno, _("col_get failed"));//この関数を書換え
        ok = false;
       }
    else
      set_col(col);
/*必要な記述範囲　終了*/
    ok &= change_file_owner (fts, ent, uid, gid,
                             required_uid, required_gid, chopt);
…
```

図 8 書換え必要箇所 1[2]

4.2 字句の種別の並びの抽象化の方法

書換え必要箇所の前後に共通範囲を求める際に，完全に一致する箇所を求めると 1 つの字句の種別が異なるだけでも範囲がそこまでに留まり，必要な箇所だけへの限定が十分に行えない．そこで，一部の字句の種別の差異を許容する方法を用いる．図 8 と図 9 は書換えを行いたい箇所の例である．例えば，図 8 と図 9 では，変更された箇所の前にある `col = get_col (f_col);` と `col = get_col (n_col[n-1]);` の記述は字句は完全には一致しないが，意味を考えれば，本来は共通範囲に含めるべき文である．そこで，プログラムの意味を考えるうえで重要な部分だけに着目するよう抽象化を行う．本研究で抽象化を行う箇所は以下の 2 つである．

- 制御文の条件式が異なる箇所
- 同一名の関数呼び出しの実引数が異なる箇所

これらの箇所を任意の式を表すパターン変数に置き換え，詳細を捨象する．

図 10 は図 8 と図 9 を書換え必要箇所として，抽象化せずに生成した書換えパターン，図 11 は抽象化を用いて生成した書換えパターンを示す．

5 評価と考察

生成された書換えパターンが目的の箇所のみに適用されるか評価をする．評価実験は，図 4 の書換えパターン生

```
…
/*必要な記述範囲　開始*/
    if(option == 'q')
      col = get_col (n_col[n-1]);//この関数内の引数が異なる

    if (col == NULL)
      {
        error (0, errno, _("col_get failed"));//この関数を書換え
        ok = false;
       }
    else
      set_next_col(col);
/*必要な記述範囲　終了*/
    ok &= change_file_owner (fts, ent, uid, gid,
                             required_uid, required_gid, chopt);
…
```

図 9 書換え必要箇所 2[2]

```
%before
    if (col == NULL)
      {
        error (0, errno, _("col_get failed"));//この関数を書換え
        ok = false;
       }
    else
      set_next_col(col);
%after
    if (col == NULL)
      {
        get_error (errno);//書換えられた関数
        ok = false;
       }
    else
      set_next_col(col);
```

図 10 種別の並びがすべて一致した共通範囲の書換えパターン

成についてツールを実装し，初期パターンと記述範囲のパターン変数は手動で整合させた．GNU Core Utilies[2] にコードクローン検索ツールである NiCad[1] を適用し，コードクローンセットを得た．そのコードクローンセット内の 1 つのコードクローンに対して一箇所変更を加え，差分を抽出し，書換えパターンを生成した．評価方法は以下の通りである．

1. 修正前後のコードクローンの差分から書換えパターンを生成
2. プログラムに適用し，書換え必要箇所だけに適用されているかを確認

```
%before
    if(option == 'q')
      col = get_col (${expr:EXPR*});//この関数内の引数が異なる
    if (col == NULL)
      {
        error (0, errno, _("col_get failed"));//この関数を書換え
        ok = false;
       }
    else
      set_next_col(col);
%after
    if(option == 'q')
      col = get_col (${expr});//この関数内の引数が異なる
    if (col == NULL)
      {
        error (0, errno, _("col_get failed"));//この関数を書換え
        ok = false;
       }
    else
      set_next_col(col);
%end
```

図 11 種別の並びを抽象化し，生成した書換えパターン

```
1 %\begin{verbatim}
2 …
3 /*必要な記述範囲　開始*/
4 if (!target_directory)
5 {
6   if (2 <= n_files&&
7       target_directory_operand (file[n_files - 1],
8                                 &sb, &new_dst))
9     target_directory = file[--n_files];
10  else if (2 < n_files)
11    error (EXIT_FAILURE, 0,
12           _("target %s is not a directory"),
13           quote (file[n_files - 1]));//この関数を書換え
14 }
15 /*必要な記述範囲　終了*/
16 …
17 %\end{verbatim}
```

図 12　書換え必要箇所 1[2]

```
1 %\begin{verbatim}
2 …
3 /*必要な記述範囲　開始*/
4 if (!target_directory)
5 {
6   assert (2 <= n_files);
7   if (target_directory_operand (file[n_files - 1]))
8     target_directory = file[--n_files];
9   else if (2 < n_files)
10    error (EXIT_FAILURE, 0,
11           _("target %s is not a directory"),
12           quote (file[n_files - 1]));//この関数を書換え
13 }
14 /*必要な記述範囲　終了*/
15 …
16 %\end{verbatim}
```

図 13　書換え必要箇所 2[2]

```
%before
error (EXIT_FAILURE, 0,
       _("target %s is not a directory")${expr:EXPR*});
%after
target_error(${expr});
%end
```

図 14　記述範囲を広げる前の書換えパターン

提案手法の精度を適合率と再現率で評価した．それぞれ以下の式 1，式 2 で求める．

$$\text{適合率} = \frac{\text{書き換わった書換え必要箇所の数}}{\text{実際に書換えられた箇所の数}} \quad (1)$$

$$\text{再現率} = \frac{\text{書き換わった書換え必要箇所の数}}{\text{書換えを行いたい箇所の数}} \quad (2)$$

5 つのコードクローンセットに対して評価を行なったところ，適合率は平均 0.87，再現率は平均 1.0 となった．再現率 1.0 からすべての書換え必要箇所に書換えが適用されることがわかった．しかし，提案手法だけでは書換え必要箇所に限定して書換えられないので，適合率が 0.87 となった．図 12 と図 13 に提案手法では失敗した例を示す．図 14 は記述範囲を広げる前の書換えパターンである．図 12 は 11 行目の `error` 文を `target_error` 文に書き換え，図 13 も同様に 10 行目の `error` 文を `target_error` 文に書き換える．記述範囲として図 12 と図 13 の 4 行目までの記述が必要であった．図 13 で 6 行目に `assert (2 <= n_files);` が存在したので，抽象化を用いても図 14 を最適な必要な記述範囲まで広げられなかった．これを改善するためには，種別の並びの抽象化方法を拡張していく必要がある．

評価方法として，コードクローンセットから書換え必要箇所をあらかじめ選択したが，プログラムの意味を考慮していないので，現実的な書換え対象の選び方とは異なる．前後に出現する文脈が類似したコードクローンを書換え必要箇所として設定するなどより現実的な書換えを用いた評価が必要である．

6　おわりに

本研究ではプログラムに存在するコードクローンや定型的な記述を書換えるパターン変換の自動生成の支援を目指し，修正前，修正後プログラムの差分から書換えパターンを生成し，書換え必要箇所の前後に出現する共通の記述を自動的に書換えパターンに組み込む手法を提案した．実装と評価を行い，提案手法によって高い精度で書換え箇所を目的の箇所に絞り込めることを確認した．

今後の課題は次の 3 つである．1 つ目は差分の選択の自動化である．差分の抽出では LCS を求めるアルゴリズムを使用したが，LCS は複数存在し，作業内容と一致しない差分が抽出される場合がある．本研究では作業内容に一致する差分を手作業で選択し使用した．2 つ目に一般化の自動化があり，本研究では図 4 の初期パターン一般化は手作業で行った．これら 2 つを自動化することで，より労力と時間の削減が可能となる．3 つ目として記述範囲を広げる方法の改善がある．本研究では，記述範囲を広げるために抽象化を行う箇所を 2 つ提案し，評価を行なったが，高い精度で記述範囲を決定できるよう改善することが求められる．

参考文献

[1] C.K. Roy and J.R. Cordy, “NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization,” Proc. ICPC, pp.172–181, Jun. 2008.

[2] GNU Project, “Coreutils — GNU core utilities,” http://www.gnu.org/s/coreutils/, Jan. 2011.

[3] 神谷 年洋，“コードクローンとは，コードクローンが引き起こす問題，その対策の現状，” 電子情報通信学会誌，vol.87，no.9，pp.791–797，Sep. 2004.

[4] 神谷 年洋，楠本 真二，井上 克郎，“コードクローン検出における新手法の提案および評価実験，” 電子情報通信学会技術研究報告，ソフトウェアサイエンス，vol.100，no.570，pp.41–48，Jan. 2001.

[5] 吉田 敦，蜂巣 吉成，沢田 篤史，張 漢明，野呂 昌満，“属性付き字句系列に基づくソースコード書き換え支援環境，” 情報処理学会論文誌，vol.53，no.7，pp.1832–1849，Jul. 2012.