# パターン変換記述における表現力向上の研究

2009SE091 伊藤佑規　2009SE095 上條敬太　2009SE145 葛谷貴亮
指導教員：吉田敦

## 1　はじめに

プログラム開発過程における保守・運用において仕様変更やリファクタリングなどを行う際には，保守性や可読性向上のためにプログラミングスタイルを統一してソースコードを変更する．しかし，コードクローンの同時編集やコーディング規約に応じた書換えなど，分散して存在する典型的な記述に対して同じ書換えを適用する際に，手作業では誤りが混入する危険性がある．そこで，誤りの混入を防ぎ作業効率向上のために，一律に書換えができるパターンマッチングを利用した書換え環境が提案されている [1][2][3]．このような支援環境は有用ではあるが，現実的な作業に適用したときに，制約を詳細に記述できず，利用できないことがある．例えば，一部の記述においてコーディング規約で求められている表記が欠落しているときに，それを一律に追加するためには，規約を満たしてない箇所に限定する制約条件を記述する必要がある．しかし，既存の環境では，そのような制約条件を記述する方法を提供していない．

本研究では，パターン変換記述に求められる現実的な制約条件を，事例を基に検討するとともに，複数の観点から派生して考られる条件も明らかにする．さらに，制約条件に対応するよう実装を行い，その実現可能性を確かめる．なお，実装にあたっては，小規模で拡張が容易な TEBA[2] を用いる．

以降では，第 2 章でどのような制約条件が必要なのかを分析し，問題点を整理する．第 3 章では，どのようにパターン適用範囲を制限し拡張していくかを述べる．第 4 章では，ツールを評価し考察を行う．第 5 章で更なる表現力の向上のための機能拡張について述べる．

## 2　問題分析

### 2.1　構文単位での否定的な書換え箇所の限定

事例として，コンパイル時の警告を無視して作成した，return 文が欠落した関数を含むプログラムに対し，返り値を書くよう指示を付加した return 文を関数の最後に追加する修正を行うことを考える．この場合，return 文を正しく記述している場合には，挿入すべきではないので，「return 文が存在しない」ということを制約条件として記述する必要がある．すなわち,「構文要素が存在しない」ということを表す制約条件が必要である．

存在しないことを制約条件として記述するのであれば，存在することも条件として考えられる．さらに，構文要素についての制約条件を記述するのであれば，字句を構成する文字に対する制約条件も考えられる．そこで,「存在の有無」,「構文要素と字句」という 2 つの観点の組み合わせで構成される 4 つの制約条件について検討する．この章ですでに述べた事例は「構文要素に対する否定」であるので，残りの 3 つの条件について，それぞれ現実的にありうる事例を検討する．

### 2.2　字句単位での否定的な書換え箇所の限定

字句に対する否定的な限定としても 2.1 節で示した事例について考える．この場合，返り値がない void 型関数定義には挿入するべきでないので，型指定子を記述する箇所に “void” が存在しないことを制約条件として記述する必要がある．すなわち,「特定箇所に字句が存在しない」ということを表す制約条件が必要である．

### 2.3　構文単位での肯定的な書換え箇所の限定

事例として，コードクローン内の特定の記述を関数呼び出しに置き換える場合を考える．コードクローンは類似しているが完全には一致していないので，関数化したい箇所は，必ずしも同じ位置や同じ数になるとは限らない．よって，コードクローンを表すパターンの中に，書換えたい内容を含めて記述することはできない．あらかじめ書換え対象をコードクローンだけに限定できていれば，記述を関数呼び出しに置き換えるパターン変換を適用し一律に同時編集することが可能である．よって，書換えを行う前にパターン変換記述を適用させる箇所を特定する方法が必要である．

### 2.4　字句単位での肯定的な書換え箇所の限定

大学におけるプログラミング実習での課題の採点作業を手作業で行うことは，学生ごとに記述が異なるので，手間がかかり，間違いを見落とす可能性がある．一つの解決策が，プログラムにテストケースを適用し，出力を模範解答と比較して自動的に判定する方法である．しかし，学生のプログラムは書式が必ずしも統一されておらず，単純に比較すると書式の違いによって差異が生じ，正しく判定できない．そこで，書式を統一する書換えを考える．しかし，計算された値のみ出力し，出力の可読性を高める補助的な文字列を除去しようとした場合，文字列定数の中身に対するパターン変換記述を作成する必要がある．よって，字句を構成する文字列に対して制約条件を記述できるパターン変換記述が必要である．

## 3　制約条件に対する処理の実現方法

### 3.1　TEBA

TEBA における書換えは，パターン変換記述を書換え前を表現するテキスト断片の変換前パターンと書換え後を表現する断片の変換後断片に分割し，それぞれを構文解析器を通して得られた字句系列パターンと置換字句系列を結合させ字句系列変換規則を生成する．しかし，パターン変換記述に書換え前後の断片をそのまま記述しただけでは意図通りには書き換えられない場合がある．そこで，差異のある要素を抽象化するためにパターン変数を用いて表現する．パターン変数は，変換前パターンでは

“${名前：種別}” の形式で記述し，変換後断片では，“${名前}” の形式で記述する．また，字句系列変換規則は，字句系列に対応するように正規表現を用いて構成される．これを用いて，書換え対象であるソースコードを構文解析器を通して得られた属性付き字句系列の書換えを行う．

属性付き字句系列とは

種別（属性値)* ’<’ テキスト’>’

で表されるものであり，種別には変数，カンマや括弧などの記号，文の始めと終わりを示すものなどがある．

### 3.2 構文単位での書換え箇所の否定的な限定方法

#### 3.2.1 制約条件を付与したパターン変換記述の提案

パターン変換記述の変換後断片において，Not_Exist_Begin と Not_Exist_End の間の行に存在すべきでない字句を記述し，Not_Exist_Begin 後の括弧内に “’ 字句’:種別” の形式で，書き換える際に存在すべきでない字句を記述する．この 2 つの記述により,「字句が存在しない」ことを表現する．括弧内に複数の字句を記述する場合はカンマで区切り，いずれの字句も存在しない場合にのみ書換えが適用される．

図 1 は,「return が存在しない」という条件を付与し，2.1 節で例として挙げた「関数定義に return 文を挿入する書換え」を表すパターン変換記述である．なお，DECR は宣言子，ID_FUNC は関数名，DECL は宣言，STMT* は複数の文を表す．

#### 3.2.2 制約条件を付与した書換えの実現方法

構文要素を構成する字句が存在しないことを判定して書換えるためには，パターンマッチのエンジンに対し，字句が存在しないことを判定する仕組を組込むことが必要であるが，その拡張は容易ではない．また，TEBA による書換えは，パターン変換記述を内部で文字列の正規表現に変換して書換えを実現しているので，正規表現を用いて「字句が存在しない」という「文字列の否定」を表現できれば制約条件を付与した形で記述できる.「文字の否定」は，文字の種類の数が有限なので補集合の要素である a 以外の文字の種類の数も有限となり，正規表現を用いて “[^a]” と表現できる．しかし,「文字列の否定」は，文字列が無数に存在し，補集合の要素である return 以外の文字列も無数に存在するので，正規表現を用いて表現

```
% before
${decr:DECR} ${func:ID_FUNC}( ${int:DECL})
{
     ${stmts:STMT*}$;
}
% after
${decr} ${func}( ${int})
{
     ${stmts}$;
      Not_Exist_Begin(’return’:RESERVE)
      return (${decr})NEED_RETURN_VALUE;
      Not_Exist_End
}
% end
```

図 1 ユーザが記述するパターン変換記述

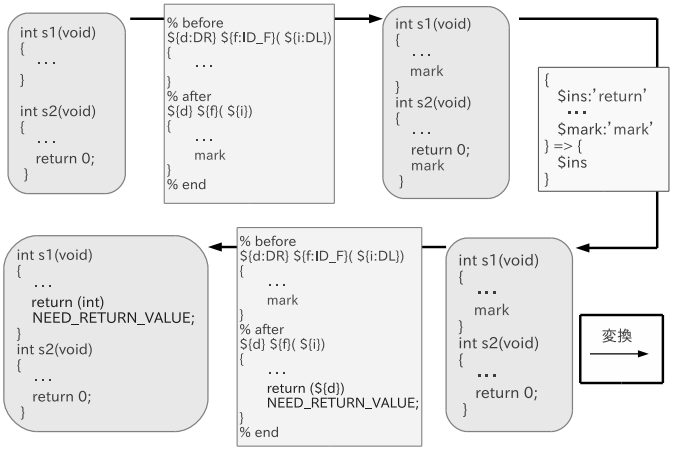


図 2 目印を用いた書換えの流れ

できない．そこで，否定的な制約条件がないときに，書換えの候補となる箇所から，否定的な制約条件を満たす箇所を取り除き，その残りの箇所にのみ書換えを適用する方法を用いる．

制約条件を含むパターン変換記述から，目印を挿入するパターン変換記述，不要な目印を削除する字句変換規則，最終的に字句を挿入するパターン変換記述の順に生成する．その後，それらのパターンを順次適用させ，対象ソースコードにおいてパターン変換記述にマッチングし，書換えにより最終的に字句を挿入する可能性のある箇所に，1 つずつ目印を付与していき，不要な目印を削除した後に目印のある箇所にのみ字句を挿入する書換えを行うことで 3.2.1 節で付与した条件を考慮した書換えを実現する．return を関数定義に挿入する書換えの処理の流れを図 2 に示す．ここでの不要な目印とは，Not_Exist_Begin 後の括弧内に記述した字句が既に存在している場合に，書換え対象箇所に付与されたものを指す．

各パターンの生成方法は次の通りである．図 1 のようなパターン変換記述の Not_Exist_Begin の行から Not_Exist_End の行までの記述を “mark” に置き換えることで目印を挿入するパターン変換記述を生成する．次に，Not_Exist_Begin 後の括弧内に記述した字句 1 つに対し 1 つ，不要な目印を削除する字句系列変換規則を生成する．最後に，図 1 のパターン変換記述の Not_Exist_Begin の行と Not_Exist_End の行を取り除くことで，最終的に字句を挿入するパターン変換記述を生成する．

### 3.3 字句単位での書換え箇所の否定的な限定方法

#### 3.3.1 パターン変数に条件を付与する方法の提案

パターン変換記述におけるパターン変数の種別のあとに

${名前:種別!/字句パターン/}

の形式で否定する字句のパターンを記述することで,「特定の場所に字句が存在しない」ことを表現する．

2.2 節で例として挙げた return 挿入の書換えを事例とした場合，型宣言子の部分に “void” という字句が存在する場合は挿入するべきでないので，

```
${decr:DECR!/void/}
```
と記述する．

### 3.3.2 条件を付与した書換えの実現方法

3.2.2 節と同様に，目印を挿入するパターン変換記述，不要な目印を削除するパターン変換記述，字句を挿入するパターン変換記述の 3 つを生成し，順に適用する．ただし，条件を付与したパターン変数の後に目印を挿入する点が異なる．

## 3.4 構文単位での書換え箇所の肯定的な限定方法

書換え対象を TEBA のパターン記法における書換え前パターンのみを記述することで表現する．これをターゲット記述と呼ぶ．ターゲット記述に適合した字句の「書換え対象」属性を「対象」に設定し，それ以外を「非対象」にする．パターン変換を行うときに,「対象」の部分のみを書換える．なお，実装では,「対象」の字句のみ「書換え対象」属性を取り除いて，書換えを行う．これは，TEBA の書換え系が追加された属性に対応していないので，書換えの対象外の箇所に意図的に属性値を残すことで，TEBA の書換え系を拡張することなく，実現できるためである．例として while 文にマッチするためのパターン記述を図 3 に示す．実装上の処理の流れを図 4 に示し,「書換え対象」属性を付加する流れを図 5 に示す．

```
${inti:EXPR}
while (${cond:EXPR}) {
      ${stmt:STMT*};
      ${succ:EXPR}$;
}
```

図 3 ターゲット記述

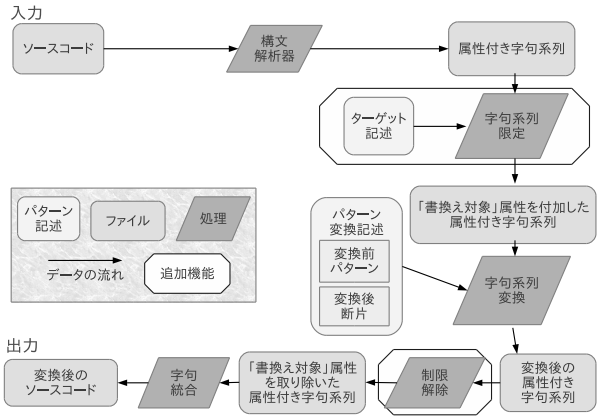


図 4 処理の流れ

## 3.5 字句単位での書換え箇所の肯定的な限定方法

### 3.5.1 パターン変数に制約条件を付与する方法の提案

パターン変換記述において，正規表現を用いてパターン変数に制約条件を付与することで表現する．しかし，変換

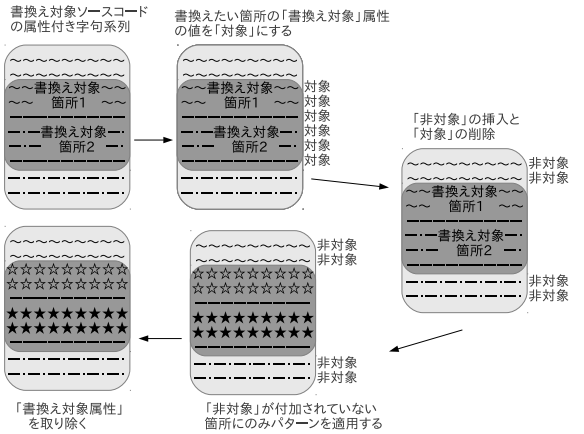


図 5 ターゲット機能の流れ

表 1 適用させる修飾子とその役割

| 修飾子 | 役割 |
|---|---|
| i | アルファベットの大文字と小文字の区別をしない |
| g | すべてのマッチング箇所を探す |
| e | 置換文字列を式として実行 |

前パターンと変換後断片では記述方法が異なるので，分けて説明する．

**変換前パターン**　字句を構成する文字列に対し正規表現を用いて制約条件を付与するために，パターン変数を

```
${名前:種別 /正規表現/修飾子}
```
と記述する．パターン変数に対応する字句が正規表現の条件を満たす場合のみ書換えが適用される．

**変換後断片**　変換後断片では，字句に対する書換えを表現するために，正規表現を用いて

```
${名前 /正規表現/置換文字列/修飾子}
```
と記述する．パターン変数に対応する字句に対し，正規表現の記述にマッチングした箇所が置換文字列に書き換わる．正規表現には，TEBA の実装言語に合わせ Perl の表記を用いる．

**正規表現に適用させる修飾子**　変換前パターン・変換後断片の正規表現において，表 1 の修飾子を記述できる．変換前パターンの正規表現では修飾子 i，変換後断片の正規表現では修飾子 g，e，i を適用させ制約条件の表現力を向上させる．

### 3.5.2 パターン変換記述における正規表現の適用方法

変換前パターンにおける制約条件は，内部で字句に適当するパターンとして単純に文字列を使用する代わりに，正規表現の記述をそのまま使うことで実現する．変換後断片における制約条件は，適用処理を行う字句を対象に，正規表現でマッチングした箇所を置換文字列に置き換える書換え処理を行うことで，適用する．

## 4　評価・考察

各制約条件に対する処理が実現できるか，実装を行い，2 章で示した事例に基づいて書換えを行なった．その結果，いずれも書き換えられたことが確認できた．

例えば，構文単位の否定に対しては，字句単位の否定で示した制約条件の記述方法と構文単位の否定で示した制約条件の記述方法を組み合わせたパターン変換記述を作成し，2.1 節の事例に対し書換えを行なった．その結果，void 型の関数定義と正しく記述した return 文を含む関数定義以外が書き換えられていることを確認した．これにより，それぞれの限定方法を相互利用した書換えが行なえることが確認できた．使用したパターン変換記述を図 6 に示す．

構文単位の肯定に対しては，Coreutils 8.17[4] のディレクトリ src に含まれる拡張子が.c のファイル (116 個）に対し，書換える範囲を指定して書換え可能であることを確認した．このことから現実の中規模のオープンソースに対してもエラーを起こすことなく書換えられることを確認した．また，応用として，ターゲット記述で指定した箇所を否定条件とする機能も実装し，動作を確認した．以下では，これを「エスケープ機能」と呼び，処理の流れを図 7 に示す．これらの機能を用いて構文単位で書換える範囲を限定することにより，適用するパターン変換記述は書換え内容を表現したものだけになり，パターン変換記述の再利用性の向上が期待できる．

構文単位の否定では，別の事例を用いて実験したところ，パターン変数が存在しないことを条件として記述できないことが判明した．例えば,「fopen により取得したファイルポインタを返す return が関数定義内に存在しない」という条件などである．これに対処するには，条件となる字句にパターン変数を記述できるように拡張する方法もあるが，エスケープ機能を用いて「fopen により取得したファイルポインタを返す return が存在する関数」を非書換え対象とすることで，この技術的課題を解決できる．エスケープ機能を用いた場合は，複数のターゲット記述を用いて書換え箇所の否定的な限定が可能である．一方，制約条件を付与したパターンを用いた書換えの場合は，1 つのパターン変換記述内で「複数の字句が存在しない」という条件を記述できる．したがって，それぞれの方法を使い分けることが重要である．

構文単位の肯定では，複数のターゲット記述を組み合わせるときに指定範囲が排他的でなければ正しく動作しなかった．これは実装上の都合であり，柔軟に制約条件を組み合わせられる仕組みが必要である．

## 5　おわりに

本研究では，TEBA のパターン変換記述における表現力を向上を図るために，パターン変換記述において「構文要素が存在しないという条件の付与」,「特定の箇所に字句が存在しないという条件の付与」「あらかじめ書換え対象箇所を特定し，パターン変換記述により書換える枠組み」,「字句を構成する文字列に対する制約条件の付与」の 4 点を拡張することで実現した．これにより，限定的なパターン変換を行うことが可能になった．

```
% before
${decr:DECR!/void/} ${func:ID_FUNC}( ${int:DECL})
{
     ${stmts:STMT*}$;
}
% after
${decr} ${func}( ${int})
{
     ${stmts}$;
      Not_Exist_Begin('return':RESERVE)
      return (${decr})NEED_RETURN_VALUE;
      Not_Exist_End
}
% end
```

図 6　組み合わせたパターン変換記述

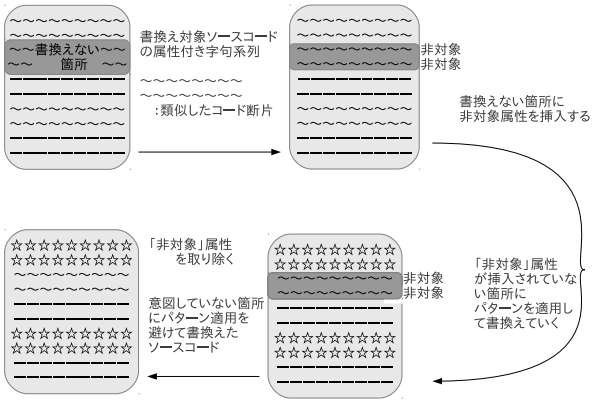


図 7　エスケープ機能の流れ

今後の課題として，パターン変換記述の表現力をより向上させるために，上記の 4 つの限定的なパターン変換記述の併用を可能にする TEBA の拡張が必要となる．

## 参考文献


[1] James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider, “Source Transformation in Software Engineering Using the TXL Transformation System,” Journal of Information and Software Technology, vol. 44, no. 13, pp. 827-837, 2002.

[2] 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, 2012.

[3] Y. Padioleau, “Parsing C/C++ Code without Preprocessing,” Proceedings of the 18th International Conference on Compiler Construction, pp. 109-125, 2009.

[4] Free Software Foundation, Inc., “*Coreutils - GNU core utilities*,” http://www.gnu.org/software/coreutils/, 2012.