# パターン変換系における前処理条件分岐指令の維持手法の提案

M2011MM065 曽我展世
指導教員：吉田敦

## 1　はじめに

ソースコードを仕様変更や可読性の向上のために修正するときに，同じ作業を複数の箇所に適用することがある．例えば，コードクローンや特定のコード規約に従った複数箇所の記述に対する修正がその典型である．同じ作業をすべて正確に繰り返すためには，プログラムのパターン変換の利用が有効な手段の一つである．パターン変換はユーザから書換え前後の字句の並びのパターンに従いプログラムを書換える．プログラムのパターン変換を利用することにより，手作業と比べ作業時間を短縮でき，ミス混入の可能性を減らせる．

プログラムのパターン変換を実現するときに，複数の言語を含むプログラム記述への対応が問題となる．複数の言語を含むプログラム記述とは，C 言語前処理 [1] や各言語に対する前処理系 [2, 3, 4] である．これら複数の言語を含むプログラム記述には，どちらかの記述のみを扱うパターン変換系は存在するが，複数の言語を同時に扱う書換え系は存在していない．

C 言語においては前処理条件分岐によって処理や定義が異なるコード片を混在して記述でき，複数の実行環境への対応等に利用される．C 言語本来のコード片に前処理条件分岐指令の構文的な制約を加えることで，コンパイルされるコード片を切り替え，複数の環境を同一のソースコード上に表現できる．前処理条件分岐内にはそれぞれの環境独自のコード片が，前処理条件分岐外にはすべての環境共通のコード片が記述される．

C 言語では前処理条件分岐やマクロ参照により，一つのソースコード内に異なる記述を織り込んで記述でき，いわばソースコードは多重化された状態 [5] とみなせる．通常，ソースコードの書換えは抽象構文木に対する操作として実現される．しかし，多重化したソースコードは，部分的に構成が異なる複数の抽象構文木を含んでおり，それらの整合性を保ちつつ同時に書換える方法は明らかでない．前処理記述と C 言語の記述を同時に扱う書換え系では，多重化した記述と多重化していない記述の境界が問題となる．書換え時に字句が境界を越えて移動すると前処理条件分岐指令が記述された意図を保持できない．本研究では前処理条件分岐指令が記述された意図のことを前処理条件分岐の「振舞い」と呼ぶ．

本研究は前処理記述に対応した書換えツールの実現を目指し，前処理条件分岐を含んだ箇所を各条件の振舞いを維持したまま書換えを行えるパターン変換手法を提案する．

操作可能な記述へと変形する方法として，各条件毎の記述を生成し，各個別の記述を操作してから統合する方法が考えられる．しかし，前処理条件分岐の条件数 $n$ に対し，最悪 $O(2^n)$ 個の記述が生成され，現実には実行不可能である．条件毎の記述を生成しない場合には，書換えルールが前処理条件分岐指令にまたがってマッチした箇所を正しく書換える仕組みが必要である．しかし，前処理前のソースコードに対する解析や書換えによく利用される戦略である，前処理の無視や前処理後に処理を行なう方法では各条件の振舞いを保てない．

実用的な時間で処理を行なうために，パターン変換が適用される箇所のみに対して各条件毎の記述を求める方法を用いる．実際の書換えでは，書換え対象となる分岐数は少ないと想定され、実用的な時間で実行できると期待できる．前処理条件分岐の振舞いを保持したまま書換える方法として，条件分岐指令を書換えの対象範囲の外側へ移動させる方法を採用する．これは前処理条件分岐指令外に記述される記述を前処理条件分岐指令内に配置するのみであり，前処理条件分岐の振舞いは変更されない．

これらの手法を実装し，条件分岐を多く含むソースコードを対象に評価することで，本手法が前処理条件分岐を含む記述に対し，どの程度の計算時間の削減の効果や，その実用性を評価する．

## 2　パターン変換における前処理記述

### 2.1　前処理記述に対応したパターン変換

図 1 に前処理記述に対応したパターン変換の例をあげる．書換え前に記したソースコード内には前処理条件分岐が記述され，マクロ `DEBUG` の有効・無効によって `then` 節，`else` 節の記述を切り替えられる．書換え前のソースコードに対し，書換えルールに従ってパターン変換を適用すると，書換え後に記すように `then` 節，`else` 節双方の振舞いを保ったまま書換えが行われることが理想である．

前処理前のソースコードに対する解析や書換えによく利用される戦略として前処理の無視や前処理後に処理を行なうものがある．前処理を無視すると，`then` 節，`else` 節の記述（図 1: 80, 81, 83, 85 行目）がすべて有効となり，書換えルールに一致しない．また，前処理後に処理を行なうと，書換えは行えるが，この例では `then` 節（図 1: 80〜85 行目）の振舞いを保つことができない．

### 2.2　コードクローンセットに含まれる前処理指令

予備実験として実際のオープンソースソフトウェアに前処理指令を含むコードクローンがどの程度あるか調査を行なった．コードクローンとは「ソースコード内に含まれるコピー＆ペーストによって作られた（あるいは，作られたように見える）コードの断片」[6] であり，コードクローンの修正には同じ作業を複数の箇所に適用しなければならない．コードクローンを検出することで，そのソースコード内にパターン変換を利用できる箇所がどの程度あるか把握できる．

コードクローン検出ツールである simian-2.3.33[7], NiCad-3.4[8] を用いて，オープンソースである coreutils-

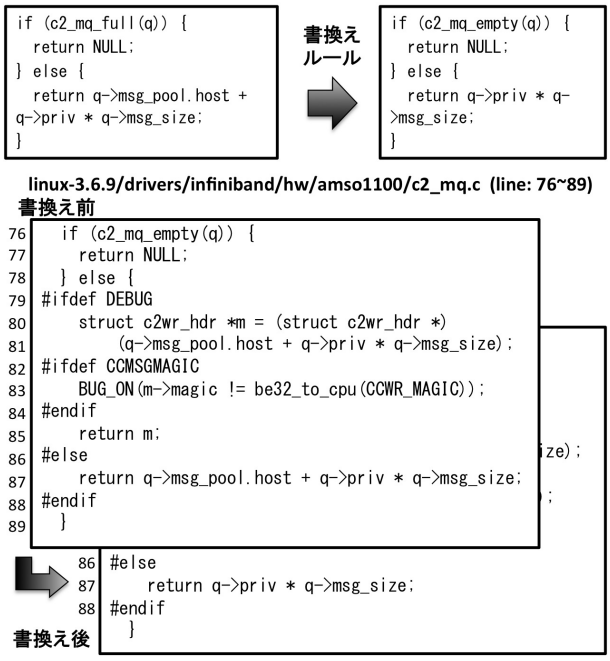


図 1 書換えルール

8.20[9], emacs-24.2[10], linux-3.6.9[11] を対象に調査を行なった．検出したコードクローンセットには前処理指令を含むものが coreutils-8.20, emacs-24.2 で約 20 %，linux-3.6.9 で約 5 %あり，それらの前処理指令のうち約 90 %が前処理条件分岐指令であった．以上よりパターン変換では現実的に前処理指令を無視できない．

## 3 組合せ爆発と書換え対象箇所内の前処理条件分岐指令

前処理条件分岐を含む箇所にパターン変換を適用するには組合せ爆発と書換え対象箇所内の前処理条件分岐指令が問題となる．この二つの問題と解決の方針を示す．

### 3.1 組合せ爆発

前処理前のソースコードに対しパターン変換を行なう際に考慮すべき問題として組合せ爆発がある．例えば，emacs-23.4/src/process.c には 329 箇所の条件分岐が含まれ，条件分岐の入れ子や連続も考慮すると最悪 $2^{329}(\simeq 1.75\times10^{86})$ の組合せが発生する．

前処理条件分岐に対応したパターン変換を実現する簡易な方法としては，書換え対象のソースコードに対しすべての条件の組合せを考え，その組み合わせごとに前処理後の記述を生成し，生成した記述に対しパターン変換を行なう方法がある．しかし，前処理条件分岐が多く含まれるソースコードでは条件分岐数 $n$ に対し，最悪 $O(2^n)$ の前処理後の記述が発生し，さらにこれらすべての記述に対し探索と書換えが行われる．現実的な時間で処理するには生成される記述の数を $O(2^n)$ よりできるだけ下回る必要がある．

生成される記述の数を抑える方法として，パターンに合致する範囲に含まれる前処理条件分岐指令の組合せのみを生成する方法を考える．これにより，適合範囲に含まれない分岐を省け，書換えルールの書き方によって組合せ爆発を抑えられる．探索の計算量が最悪となるケースは書換えルールの最後の字句で適合せず，ファイル内の最初の分岐から最後の分岐まで探索する場合である．

探索対象のソースコードの行数を $l$，適合した箇所の行数を $m$ とし，前処理条件分岐指令 $n$ が均等に分散していると仮定すると，対象となる分岐数は $n\times m/l$ と見積もれ，ソースコード内のある箇所から適合する箇所を調べる計算量は $O(2^{n\times m/l})$ となる．ソースコードの行数と条件分岐数には比例関係があると仮定すると，前述の分岐が均等に分散しているという仮定から $n=kl(k$ は定数$)$ となり

$$O(2^{n\times\frac{m}{l}})=O(2^{n\times\frac{km}{n}})=O(2^{km})$$

と整理できる．これはソースコード内のある箇所からの計算量なので，ソースコード内すべての箇所から探索を行なうと計算量は $O(2^{km}\times l)$ となる．全体の計算量が指数関数的であるのは避けられないが，対象となる分岐数 $km$ はユーザが記述するパターンに依存するので，パターンの書き方によって組合せ爆発を抑えられる．

### 3.2 書換え対象箇所内の前処理条件分岐指令

#### 3.2.1 前処理条件分岐指令にまたがったパターンマッチ

本研究では前処理条件分岐指令の振舞いを変えずに書換えを行なうために，書換え対象箇所から前処理条件分岐指令退避する方法を採用する．これは，対象となる前処理条件分岐指令外の記述を前処理条件分岐指令内に取り込むことで実現でき，各条件で有効となるコードは変わらず前処理条件分岐指令の振舞いは変わらない．

#### 3.2.2 冗長な記述

前処理命令を外側へ移動させると書換え対象箇所に冗長な記述が生成される．冗長な記述とは前処理条件分岐指令内すべてに共通して記述される字句である．冗長な記述には，前処理条件分岐指令を退避させることによって生じるものと，もともと前処理条件分岐指令内に記述されていたものがある．前者は書換え後に排除すべきものであるが，後者は可読性を高めるためなど，意図的に書かれた可能性があるので残すべきである．よって，これらを区別するよう，各字句にどの条件分岐に属したか紐付けする．書換えた後にこの紐付けに基づき，書換えで一時的に生じた冗長な記述を排除する．

## 4 前処理条件分岐に対応したパターン変換方法

図 2 に前処理条件分岐に対応したパターン変換系の概要を記す．当変換系ではユーザからソースコードと書換えルールを受け取り，パターン変換済みのソースコードを出力する．ソースコードの構文解析器には TEBA[12] を用いる．前処理前のソースコードに対応していることと，ソースコードの断片においても構文解析可能なことがその理由である．

前処理条件分岐に対応したパターン変換系では，前処理条件分岐指令の退避操作・復元操作が必要になる．前処理条件分岐指令の退避操作は書換え直前に行ない，書

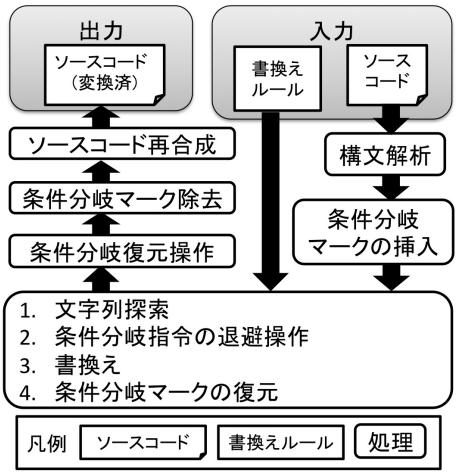


図 2 前処理条件分岐に対応したパターン変換系

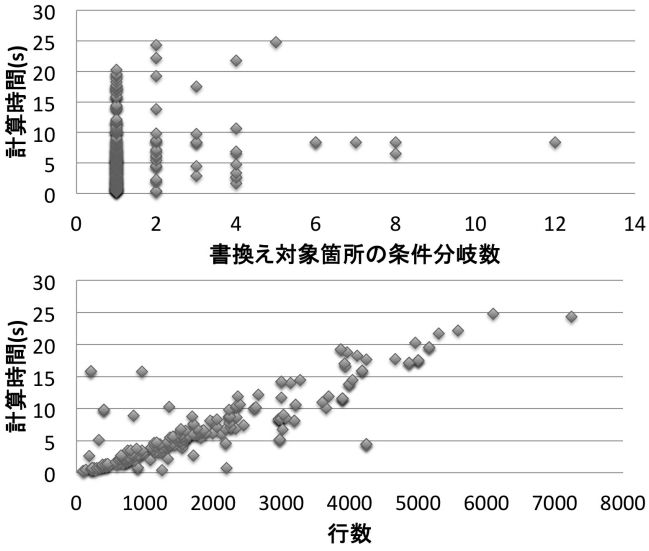


図 3 評価結果

換え対象範囲内の前処理条件分岐指令を退避する．前処理条件分岐指令の復元操作はソースコード内のすべての書換えが終わった後に，ソースコード全体から復元可能な前処理条件分岐指令を探索し復元する．

### 4.1 前処理条件分岐指令の退避操作・復元操作

前処理条件分岐指令の退避操作では書換え対象箇所内の前処理条件分岐指令前後の対象箇所を条件分岐内に引きこむ．else 節がない前処理条件分岐指令については else 節を追加する等，計 7 種類の書換えが必要である．前処理条件分岐指令が連続して現れた場合には退避操作を外側の前処理条件分岐指令から繰り返し適用することで矛盾なく処理できる．

条件分岐の復元操作時には残すべき冗長な記述と，そうではないものを区別する必要がある．そこで，すべての字句に対して，各字句が所属する条件分岐の識別記号を付ける．この条件分岐の識別記号を本研究では「条件分岐マーク」とよぶ．条件分岐マークが表す条件から外れない範囲で共通部分を条件分岐外へ移動し，残す必要のない冗長な記述を前処理条件分岐指令内から排除する．

### 4.2 探索アルゴリズムの拡張

組み合わせ爆発を回避するために，前処理条件分岐指令を含んだ状態で適合範囲を正確に求める必要がある．そこで，文字列探索アルゴリズム内で書換えルールに適合する可能性がある箇所のみ，すべての経路を探索する．

本研究ではシンプルな文字列探索アルゴリズムである brute-force search を選択し，拡張した．この拡張したアルゴリズムでは，ある字句からパターンが適合を始め，完全に適合する，もしくは適合しないと判定できるまでに分岐に出会ったら各分岐経路について再帰的に適合の判定を行なう．ひとつの分岐経路で適合したら，その範囲に含まれる前処理条件分岐指令の退避を行ない，書換えを実施する．適合を開始した字句は記録し，条件分岐マーク復元後には記録した字句の次の字句から再度探索を行なう．

## 5 評価と考察

本研究の提案手法の有効性を示すために，提案手法の精度と時間の評価を行なった．本研究の有効性については 2.2 節に記した予備実験をもって評価する．

### 5.1 評価方法

提案手法の精度の評価では，手作業により前処理命令による記述の多重化をすべて含む書換え対象箇所を用意し，パターン変換ツールでそれらを正しく書換えられるか確認する．これにより，パターン変換ツールが正しく前処理前の記述を処理できるかが解る．また，書換えにかかった時間を記録し，現実的な時間で処理できるかを確認する．評価対象は予備実験において検出した，linux-3.6.9/driver/以下のコードクローンを対象とし，各コードクローン毎に手作業で作業した．

### 5.2 評価結果

すべてのコードクローンにおいて正しくパターン変換が行えた．パターン変換のときに生じる問題として空白の変化があるが，これは本研究室にて提案されている手法 [13] において補正が可能であり，正しくパターン変換が行えたものとした．

書換えの時間は平均で 2.66s だった．最長は linux-3.6.9/drivers/scsi/osst.c[1]で，平均 39.36s かかった．osst.c は評価対象の中で最長の行数であり，最長の実行時間になったと考えられる．図 3 に評価結果のグラフを示す．書換え対象箇所の条件分岐数と計算時間のグラフより，実際の書換えにおいては書換え対象箇所の条件分岐数が低く抑えられ，分割箇所は少なく抑えられたことがわかる．また，全体の計算時間は行数に応じて多項式的に伸びる結果となった．

[1]行数:6095 条件分岐数:213 書換え対象箇所に含まれる条件分岐: 5

### 5.3 考察

計算量推定時に用いた仮定，探索アルゴリズムの妥当性について考察を行なう．本研究では組み合わせ数に下記 2 つの仮定を用いた．

- 前処理条件分岐指令はソースコード内に均等に分布
- 条件分岐数とソースコード行数は相関関係

これらの仮定については emacs-23.4 のソースファイルで，この傾向にあることを確認している．しかし，行数の短いソースコードほど前処理条件分岐指令の分布が偏っている．emacs-23.4 では前処理命令が比較的多く利用されているが [15]，オープンソースソフトウェアであり，他のソフトウェアにおいても上記の仮定が成り立つかや，変則的なソースコードが多く含まれるかは，今後の調査が必要である．

本研究では文字列探索に brute-force search を用いたので必ずしも効率は良くないが，実用的な時間で処理できた．これは書換え対象箇所の条件分岐数が現実的にも少なかったことが原因である．前処理条件分岐指令を多く含む書換えを多く行なう場合には，未だ組み合わせ爆発の可能性があり，現実の開発においてそのような書換えを行なう可能性は否定できず，どのような書換えが実際に行われるか調査が必要である．

全体の計算時間が行数に応じて伸びていることから，今後さらに大規模なソフトウェアに適用するには，より効率の良い探索アルゴリズムを採用する必要がある．そのためには，条件分岐に対応可能な他の効率のよいアルゴリズムを見つける必要がある．本研究の評価実験においては，実用上問題ない性能を出せたが，さらに大規模なソフトウェアにおいて実用時間内に処理できない場合には，前処理条件分岐に対応したパターン変換系と通常のパターン変換とを使い分けていく必要がある．

## 6 関連研究

関連研究として前処理系解析 [5] や前処理のリファクタリング [14]，そしてコードクローンに関する研究 [6, 16] がある．しかし，我々が知りうる範囲において，本研究が提案する前処理記述に対応したパターン変換系に関する研究は存在しなかった．

## 7 おわりに

本研究は前処理記述に対応した書換えツールの実現を目指し，前処理条件分岐を含んだ箇所を各条件の振舞いを維持したまま書換えを行えるパターン変換手法を提案した．実用的な時間で処理を行なうために，パターン変換が適用される箇所のみに対して個別の記述を求めることで，組み合わせ爆発を回避しつつパターン変換が行えることを示した．また，これらの手法を実装し，オープンソースソフトウェアに対して正しく，かつ，実用時間内で書換えられることを確認した．今後の課題としては，さらに大規模なソフトウェアを対象に検証を行なうことと，brute-force search 以外の文字列探索アルゴリズムの中で，本研究に適用可能なものを用いる必要がある．

## 参考文献


[1] ISO, *ISO/IEC 9899-1999: Programming Languages —C*, 1999.

[2] S. S. Some and T. C. Lethbridge, “Parsing minimization when extracting information from code in the presence of conditional compilation,” In *Proc. IWPC*, pp. 118-125. IEEE Computer Society, 1998.

[3] E. Figueiredo et al., “Evolving software product lines with aspects: An empirical study on design stability,” In *Proc. ICSE*, pp. 261-270. ACM Press, 2008.

[4] BigLevel Software, Inc., Austin, TX., *BigLever Software Gears: User’s Guide*, version 5.5.2 edition, 2008.

[5] C. Kastner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann and T. Berger, “Variability-aware parsing in the presence of lexical macros and conditional compilation,” *Proc. of the 2011 ACM inter. conf. on OOPSLA*, pp. 805-824, 2011.

[6] 神谷年洋，肥後芳樹，吉田則裕, “コードクローン検出技術の展開，” コンピュータソフトウェア， Vol. 28, No. 3, pp. 29-42, 2011.

[7] Simon Harris, “Simian - Similarity Analyser,” http://www.harukizaemon.com/simian/, 2011.

[8] J.R. Cordy and C.K. Roy, “The NiCad Clone Detector,” in *Proc. of the Tool Demo Track of the ICPC 2011*, IEEE Press, pp. 219-220, 2011.

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

[10] Free Software Foundation, Inc., “GNU Emacs,” http://www.gnu.org/software/emacs/, 2012.

[11] Linux Kernel Organization, Inc., “The Linux Kernel Archives,” http://www.kernel.org/, 2012.

[12] 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プログラム書換え支援環境，” 情報処理学会論文誌， Vol. 53, No. 7, pp. 1832-1849, 2012.

[13] 大島誠也，“プログラムのパターン変換系における空白字句保存手法に関する研究，” 南山大学数理情報学部 *2013* 年度卒業論文要旨集, 2013.

[14] A. Garrido and R. Johnson, “Challenges of refactoring C programs,” *Proc. of the IWPSE*, pp. 6-14, 2002.

[15] M.D. Ernst, G.J. Badros, D. Notkin, “An Empirical Analysis of C Preprocessor Use,” *IEEE Trans. on Software Eng.*, pp. 1146-1170, 2002.

[16] E. Duala-Ekoko and M. P. Robillard, “Clone region descriptors: Representing and tracking duplication in source code,” *ACM Trans. on Software Eng. and Method.*, Vol. 20, pp. 1-31, 2010.