プログラム書換え環境における前処理の条件分岐への対応方法に関する研究

2008MI034 林 香織　　2008MI059 五十嵐 彩
指導教員　吉田 敦

1　はじめに

プログラム開発における仕様変更や，リファクタリングなどのソースプログラムの書換え作業の多くは手作業で行われている．手作業による書換えは，ソースプログラムの中から対象箇所を探し，書き換えるという単調な作業の繰返しであり，誤りが混入する可能性がある．そこで，自動書換えツールにより，誤りの混入を解決することが提案されている [3]．また，自動書換えツールを用いることで，作業効率の向上が期待できる．

C 言語のソースプログラムは前処理が利用されている．前処理の条件分岐はデバッグや，ハードウェア・OS 互換などに多く用いられる．プログラマが編集するソースプログラムは前処理前の状態である．よって，ソースプログラムの自動書換えを用いた開発支援をするためには，前処理前での正確な構文解析が必要となる．しかし，前処理前に前処理の条件分岐を考慮した構文解析を行うことは困難である．その原因は，前処理命令が任意のトークン間に記述できることにある．任意のトークン間での前処理命令の出現を想定した解析器の作成は困難である．

この問題の解決策として，前処理命令の無視によって近似的な構文解析を行う方法がある．しかし，実際には，構文が前処理の条件分岐にまたがって記述されていることで，解析できないソースプログラムがある．また，近似解を得る方法は解析においては多くの場合に有効であるが，前処理前での自動書換えでは正確な構文解析が必要となるので，近似解では不十分である．

本研究では，前処理前解析における前処理の条件分岐の問題を解決すること，および，前処理の条件分岐に対応した書換え環境を整えることを目的とする．そのために，前処理命令の無視による解析で正確な構文解析を行う方法として，前処理の条件分岐の組合せごとに異なるソースプログラムの生成をし，それぞれの解析後にマージする方法を考える．しかし，すべての分岐の組合せを導出することは現実的には不可能である．例えば，FreeBSD[1] 8.2 の ntpd.c には 112 の分岐があるので，組合せ数は単純に計算すると 2^{112} になり，膨大な計算時間と記憶空間が必要となる．また，組合せ数が少ない場合でも，マージの際に，各解析結果の間で同一の断片に対しての解析情報が異なることがあり，その不整合を解消する必要がある．そこで本研究では，ソースプログラムの複製を用いた解析および書換え方法，および，複製時の組合せを抑える方法を提案する．

2　前処理前解析について

前処理前解析を困難にしている原因は，ソースプログラム中の任意のトークン間に前処理命令を記述できることである．任意のトークン間での前処理の条件分岐命令の出現を想定した解析器の作成は困難である．ここで，前処理命令を無視して，近似の解析結果を得る方法がある．この方法は，次のような理由で，多くの場合に有効である．一般的に，前処理命令は構文要素の区切りに記述される．これは，ソースプログラムの可読性の維持や，Emacs[2] などのエディタでソースプログラムを編集する際に自動インデントで意図しないインデントが入ることを避けるためである．しかし実際には，構文要素の区切りではない箇所に前処理の条件分岐命令が記述されており，前処理命令を無視する方法では構文解析できないソースプログラムが存在する．そこで，構文解析できない原因の調査を行う．

前処理命令を無視した構文解析を行う解析器である TEBA[3] を利用し，FreeBSD 8.2 の src 以下のファイルを対象に調査を行った．調査の結果，前処理の条件分岐が原因で括弧の対応がとれず，実行エラーや解析結果に誤りが生じるソースプログラムが 98 あった．この問題について次に示す．

図 1 のように，左中括弧と右中括弧の数が異なる場合がある．この場合，前処理命令を無視して解析する方法では，条件分岐を考慮できないので，括弧の対応がとれず，正確に構文解析できない．また，右中括弧が左中括弧より多い場合も同様の問題が起こる．さらに，左括弧の数が右括弧の数より多い場合と，右括弧の数が左括弧の数より多い場合が同時に起こる場合がある．このような場合では，ソースプログラム中の左括弧と右括弧の数が等しくなることで解析は可能になるが，前処理の条件分岐を考慮していないので，正しい括弧の対応がとれない．また，中括弧のない制御文や，括弧が丸括弧の場合

FreeBSD src/contrib/amd/libamu/wire.c 389行目～

```
#ifdef HAVE_STRUCT_IFADDRS_IFA_NEXT
   ......
  for (ifap = ifaddrs; ifap != NULL; ifap = ifap->ifa_next) {
#else /* not HAVE_STRUCT_IFADDRS_IFA_NEXT */
   ......
  for (i = 0,ifap = ifaddrs; i < count; ifap++, i++) {
#endif /* HAVE_STRUCT_IFADDRS_IFA_NEXT */
   ......
 }
```

図 1　括弧の対応の例

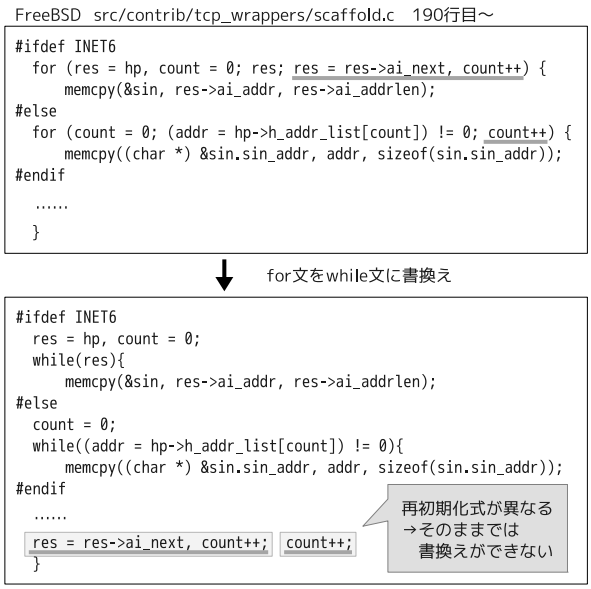

図 2　そのままの形では書換えすることができない例

でも，同様の問題が起こる．

前述の問題より，前処理の条件分岐を考慮した解析・書換えが必要であると考える．

3　ソースプログラム複製による前処理前解析

本研究では，前処理前プログラムの解析・書換えにおける問題を解決することを目的とし，ソースプログラムの複製における，組合せ爆発，および，前処理命令を含んだ書換え候補箇所の判別の問題に対処する．そこで，TEBA を用いてソースプログラム複製による解析・書換えの実装を行った．ここでのソースプログラムの複製とは，前処理の条件分岐における分岐の組合せに基づいて，異なるソースプログラムの生成を行うことである．TEBA は，ソースプログラムを解析し，属性付き字句系列に変換する．また，字句に対する属性値として，構文の開始と終了や括弧の対応を同一番号の ID で表現する．書換え時には，書換えパターンに対して構文解析を行い，解析結果を基に生成した正規表現を用いてパターンマッチをし，書換えを行う．

3.1　解析方法について

前処理の条件分岐に対応した解析方法として，書換え時の問題と，既存の解析器自体に大きく手を加えることがないという理由から，本研究ではソースプログラムの複製により解析・書換えを行う．書換え時の問題とは，書換えの際に変形が必要なソースプログラムが存在することである．図 2 に示すソースプログラムの for 文を while 文に書き換える場合は，for 文の再初期化式が異なるので，そのままの形で書換えできない．そこで，整合性を保った書換えを行うために，書換え箇所ごとにソースプログラムの複製を行い，書換え後に比較・変形を行う必要がある．書換え時の問題は，マージの際に整合をとる方法と同一の方法で解決できると考える．

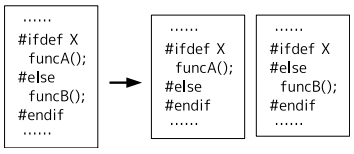

図 3　ソースプログラムの複製

ソースプログラムの複製を用いた方法では，前処理の条件分岐の組合せを導出し，それを基に複製を行う (図 3)．異なる複製をしたソースプログラムそれぞれに対して構文解析を行い，構文解析後に解析情報をマージする．書換えの際は，マージ前に，複数の構文解析情報それぞれに対して書換えを行う．

解析情報のマージの際には，どの分岐を通っても同じ解析情報を持つトークンと，異なる解析情報を持つ可能性があるトークンが存在する．この場合，前処理の条件分岐外のトークンは，どの分岐を通っても同一の字句情報を持つが，括弧やステートメントの対応といった構文情報は分岐ごとに異なる．そこで，マージの際に情報の整合をとる必要がある．本研究では，前処理の条件分岐外で異なっている箇所のみを新たに作成した前処理の条件分岐に入れる方法を考えた．しかし，新たに作成する前処理の条件分岐の数によってはソースプログラムの可読性が低くなるので，別の方法をとる必要がある．

3.2　解析・書換え手順

ソースプログラム複製による解析・書換えは次の処理で構成される．

1. 字句解析・構文解析 (近似)
2. 組合せ対象の判別
3. 書換え候補箇所判別
4. 複製時の組合せ導出
5. 字句解析結果の複製
6. 構文解析
7. 書換え
8. 解析情報のマージ
9. 解析結果をソースプログラムへ逆変換

3.3　ソースプログラム複製

ソースプログラムの複製では，前処理の条件分岐における分岐の組合せを導出し，それを基に字句解析結果の複製を行う．

実装したツールを用いて，前処理の条件分岐の数が 20 個程度のソースプログラムに対してすべての組合せを導出しようとしたところ，計算量が膨大となり，メモリ不足に陥った．そこで，組合せ数を低く抑える方法として 2 つの方法をとる．

1 つ目の方法は，条件分岐内の記述に応じて組合せの対象とするかを判別する方法である．前処理の条件分岐内で構文の開始と終了の対応がとれており，かつ，書換え候補箇所を含んでいない場合は，条件分岐を考慮せず

FreeBSD src/contrib/ntp/ntpd/ntpd.c

5行目〜

```
#ifdef HAVE_CONFIG_H
# include <config.h>
#endif
```

532行目〜

```
        if (HAVE_OPT( INTERFACE )) {
#if 0
                int     ifacect = STACKCT_OPT( INTERFACE );
                char**        ifaces  = STACKLST_OPT( INTERFACE );

                /* malloc space for the array of names */
                while (ifacect-- > 0) {
                        next_iface = *ifaces++;
                }
#else
                specific_interface = OPT_ARG( INTERFACE );
#endif
        }
```

図 4　後の解析情報に影響を与えない前処理の条件分岐

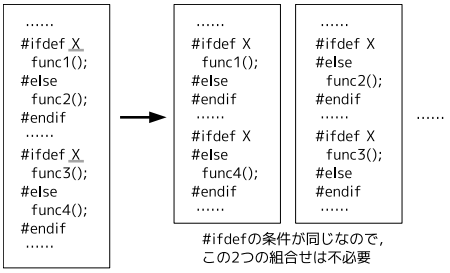

図 5　不必要な組合せの例

```
// Program Rewriting: for to while
% before

for ( ${init:EXPR} ; ${cond:EXPR} ; ${succ:EXPR} )
  ${stmt:STMT}$;

% after

${init};
while (${cond}) {
  ${stmt}$;
  ${succ};
}
% end
```

図 6　書換えパターンの例

FreeBSD src/contrib/ntp/ntpd/ntpd.c 460行目〜

```
            int optct = optionProcess(
#ifdef SIM
                                &ntpdsimOptions
#else
                                &ntpdOptions
#endif
                                , argc, argv);
```

図 7　式文の途中に前処理の条件分岐命令が含まれている例

に解析をしても問題がない (図 4)．以降では，このような前処理の条件分岐を，後の解析情報に影響を与えない前処理の条件分岐と呼ぶ．後の解析情報に影響を与えない前処理の条件分岐であっても，式文や制御文の範囲による問題に対処するために，書換え時には組合せの対象とする必要がある場合がある．この方法を用いて組合せを導出したところ，多くのソースプログラムで組合せ数を低く抑えることができた．したがって，この方法は妥当であると考える．しかし，これだけでは，組合せを低く抑えることができないものも実際には存在する．

2 つ目の方法は，前処理の条件分岐命令の入れ子関係や条件の関係から，実行されない分岐を組合せから外す方法である．例えば，同じ条件を持つ ♯ifdef が複数存在した場合，♯ifdef と ♯else という実行されない分岐を組合せの対象としない (図 5)．この方法では，前処理の条件分岐に使われる条件がすべて別の条件である場合など，組合せ数が減らない場合がある．

1 つ目の方法で組合せ数を抑えて組合せを導出した後，2 つ目の方法で組合せ数をさらに減らし，残った組合せを基にソースプログラムの複製を行う．

3.4　書換え候補箇所の判別

本研究で実装に用いる TEBA は，書換えパターンによってプログラムを書き換えるフィルタ型のツールで構築されている．例として，for 文を while 文に書き換えるための書換えパターン例を図 6 に示す．%before の後に記述されている書換え前パターンにマッチする箇所があれば，%after の後に記述されている書換え後パターンに書き換える．

式文や制御文の途中に前処理の条件分岐命令が記述されており，後の解析情報に影響を与えないので組合せの対象にしないが，書換えを考慮した場合，組合せの対象にする必要がある前処理の条件分岐が存在する．図 7 のソースプログラムは実質 2 通りの optionProcess の関数呼出しである．しかし，前処理命令を無視して解析する方法では，1 つの関数呼出し optionProcess(&ntpdsimOptions&ntpdOptions, argc, argv); として解析される．この解析結果は構文的には誤りではないが，正しい解析結果ではないので，一方の分岐だけを想定した書換えができないという問題が起こる．

このような前処理の条件分岐を組合せの対象とするために，書換え候補箇所の判別を行う．書換え候補箇所を探す際には，書換え前パターンを用いる．ソースプログラム複製時の組合せの対象となるかを判別するので，複製した状態ではなく，元のソースプログラムを判別する必要がある．しかし，書換え対象箇所のトークン間に前処理命令が記述されているので，そのままの形でパターンマッチによる判別はできない．

本研究では，書換え前パターンをいくつかに分解する方法を用いて，書換え候補箇所の判別を行う．この方法では，図 8 のように決まった数のトークンごとに書換え前パターンを分解し，トークンのかたまりごとにパターンマッチを行う．マッチした箇所の直前もしくは直後に前処理の条件分岐命令があるとき，該当する前処理の条件分岐を組合せの対象とする．

この方法の問題として，パターンマッチを行うトークンのかたまりの数によって，取りこぼしが出たり，書き換えの対象箇所とは関係ない前処理の条件分岐が組合せの対象となることがあげられる．また，この方法では，トークンのかたまりの数によっては計算時間が膨大になる可能性がある．実際に書換え前パターンを正規表現に

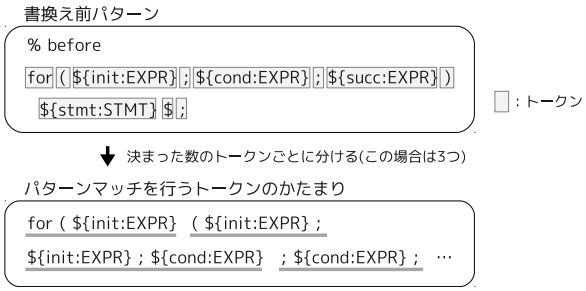

図 8　書換え前パターンの分解方法

変換したものを用いてパターンマッチを行ったところ，書換えとは関係ない前処理の条件分岐が多数組合せの対象となった．そこで，トークンのかたまりの数を増やすことで対処を試みたが，組合せの対象とすべき前処理の条件分岐を取りこぼす結果となった．よって，今後の課題として，パターンマッチを行うトークンのかたまりの数を少なめに設定し，複数のトークンのかたまりがマッチしたとき，書換え前パターンにおいて連続したものであるかを判別することが考えられる．

4　評価と考察

ソースプログラム複製による解析・書換えツールの評価には，FreeBSD 8.2 を用いた．評価の対象は，調査で見つかった前処理の条件分岐が原因で構文解析できない 98 個のソースプログラムとした．評価を行った前処理の条件分岐数は 2704 個である．

4.1　評価方法

本研究で提案した，ソースプログラム複製による解析方法を実装したツールに対する評価として，ツールでの判別結果と目視での判断結果を比較して評価を行う．また，ソースプログラム複製による構文解析ツール全体の実行時間について評価を行う．

4.2　評価結果

前述した 98 個のソースプログラムを用いて解析の動作検証を行った結果は次のようになった．

- 組合せの対象となる前処理の条件分岐を判別するツールの網羅性である再現率が 100%，精度である適合率が 90.2%，これら 2 つの調和平均である F 値が，0.94 と高い数値になった
- 96.0% のソースプログラムに対して組合せを正確に導出できた
- 実行されない分岐がすべて正確に判別された
- 組合せを正確に導出できたソースプログラムを正確に複製できた
- 組合せ数を抑えられた 81 個のソースプログラムの平均実行時間は 25.2 秒となった

4.3　考察

評価結果より，実装したツールは妥当であると考える．組合せ数を抑えられたソースプログラムの平均実行時間は 25.2 秒であるので，現実的な実行時間であると考える．検討すべき問題は，異なる複製の中の同じ断片に対して同一の解析を繰り返すことで計算量が増え，組合せ数が増えるにしたがって計算時間が膨大になることである．例えば，組合せ対象が 10 個のソースプログラムに対して複製を用いて構文解析を行う場合，組合せを基にした字句解析の複製と，それぞれの複製に対する構文解析に 40 分程度の時間がかかる．これは，組合せ数が低いソースプログラムの平均実行時間 25.2 秒の約 95 倍である．そこで，解決策として 2 つの方法を考える．1 つ目は，解析時に解析情報のキャッシュを用いて，複製したソースプログラム内の重複箇所の解析の計算量を減らす方法である．2 つ目は，ソースプログラムの複製後，前処理の条件分岐が含まれていない関数を解析前にマージし，複製したソースプログラムの重複箇所を減らす方法である．

評価の結果，組合せ数を抑えられた 81 個のソースプログラムから，現実的な実行時間内で構文解析できるソースプログラムは，前処理の条件分岐の組合せ数が約 10 通り以下のものであるということがわかった．組合せ数を抑えずにすべての組合せでソースプログラムの複製を行うと，現実的な実行時間内で構文解析できるソースプログラムは，38 個であり，全体の 13.3% となる．それに対し，組合せ数を抑えると，81 個の 82.7% になる．組合せ数を抑える方法では，現実的な実行時間で構文解析できるソースプログラムの割合が 69.4 ポイント上昇する．したがって，本研究で提案した，組合せ数を抑えてソースプログラムの複製を行う方法は妥当である．

5　おわりに

本研究では，前処理の条件分岐命令の条件や，条件分岐内の構文の開始と終了を調べることで，組合せ数を抑えたソースプログラムの複製による解析を行った．これにより，組合せ爆発になる問題を解決することができた．

今後の課題としては，書換え候補箇所の判別や，組合せを減らす場合の ♯ifdef，♯ifndef の関係のような特定の組合せの判別に対する対策が挙げられる．また，書換え時に変形が必要な問題や，重複した記述に対して同一の解析をすることで計算時間が増える問題，マージ方法についても検討が必要である．

参考文献

[1] The FreeBSD Project, “*FreeBSD*,” http://www.freebsd.org/ja/, 2011.

[2] GNU Project, “*GNU Emacs*,” http://www.gnu.org/software/emacs/, 2011.

[3] 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の試作，” ソフトウェアエンジニアリング最前線 (ソフトウェア・エンジニアリング・シンポジウム 2010 予稿集)，pp.119-126，Aug. 2010.