字句列の共通性に基づく例外処理条件文の抽出手法の提案

吉田 敦[1,a)]　加藤 大貴[2]　蜂巣 吉成[1,b)]　桑原 寛明[1,c)]　阿草 清滋[3,d)]

概要：高い品質のプログラムを作成するうえで例外処理は不可欠である．ここでの例外処理は，何らかの異常により処理を中断したり，回復を試みる処理である．もしソースコード中から，自動的に例外処理部を区別できれば，例外処理の記述忘れや必要な処理が欠けた不適切な例外処理の発見などに応用できる．また，ソースコードを読んで主要な流れを把握したいときに，例外処理部を一時的に隠すことができれば，ソースコードの理解が容易になる．本論文では，実用的なアプリケーションのソースコードでは，例外処理の記述に統一性があることに着目し，字句列の共通性に基づいて例外処理部を抽出する手法を提案する．実験では，オープンソースプログラムの 1 つ Coreutils-8.29 に適用し，約 8 割の例外処理部が上位に連続し，約 7 割が非例外処理部より上位に位置することを確認した．また，手法の限界を確認するために，Postfix-3.2.4 にも適用し，提案手法が適用しにくいものがあることも確認した．

Extraction of Conditional Statements for Exception Handling based on Commonality of Token Sequences

1. はじめに

高い品質のソフトウェアを開発するには，通常処理だけでなく，例外処理についても正確に記述する必要がある．ここでの例外処理とは，API などの関数（手続き）や処理中のデータの異常値などで，処理が継続できないときに，エラーログの出力やプログラムの中断，あるいは，エラー状態からの回復のための措置などの処理を指す．もしソースコード中から，自動的に例外処理部を区別できれば，例外処理の記述忘れや必要な処理が欠けた不適切な例外処理の発見などに応用できる．また，ソースコードを読んで主要な流れを把握したいときに，例外処理部を一時的に隠すことができれば，ソースコードの理解が容易になる．

一般に，実用的なプログラムのソースコードでは，例外処理に対して規約あるいはそれに類するものが存在し，統一的な記述になる．これは，例外処理時に，エラーログの出力やリソースの解放，プログラムの停止などの処理が必要であり，それらの専用の関数の呼び出すようにして保守性を高めているからである．本論文が対象とする例外処理部の例を**図 1** に示す．図 1 では，ハッシュ用のデータの解放や，エラーログの出力，エラー状態用の変数の設定などが含まれている．このような例外処理部を見つける方法として，標準エラー出力の参照（stderr）やエラー出力用関数（error）などの条件に基づいた検索が考えられる．しかし，そもそもどのような字句とすべきかは，アプリケーションに依存し，その決定は容易ではない．既存研究 [3] では，実行経路を調べ，分岐後の経路が短い条件文を例外処理部と判定しているが，条件文の中でライブラリ関数等，外部の関数を呼び出していると判定できない．また，再試行を行うような例外処理の場合には，実行経路は長くなるので，適用できない．

もし例外処理部が規約などにより，統一的に書かれるのであれば，それらが持つ字句の共通性に着目することで，探索の条件となる字句も自動的に同定できるはずである．ただし，共通する字句を単純に求めると，printf 関数のように，正常処理にも頻出する字句が上位となり，例外処理とは区別ができない．一方，N-gram のように，字句の並びを単位として考えると，正常処理のコードでは同じ並びは出現しにくい．正常処理のコードでも，同じ処理を繰り

[1] 南山大学
Nanzan Univ., Showa, Nagoya, 466–8673, Japan
[2] 株式会社バッファロー
Buffalo Inc., Naka, Nagoya, 460–8315, Japan
[3] 京都高度技術研究所
ASTEM, Shimogyou, Kyoto, 600-8813, Japan
[a)] atsu@nanzan-u.ac.jp
[b)] hachisu@se.nanzan-u.ac.jp
[c)] kuwabara@nanzan-u.ac.jp
[d)] agusa@astem.or.jp

```
            if (ev->wd == f[i].parent_wd)
              {
                hash_free (wd_to_name);
                error (0, 0,
                    _("directory containing watched file was removed"));
                errno = 0;  /* we've already diagnosed enough errno detail. */
                return true;
              }
```

図 1　例外処理部の例: Coreutils-8.29/src/tail.c
Fig. 1　An example of exception handling: Coreutils-8.29/src/tail.c

返すことは想定できるが，関数にまとめたり，繰り返し文で記述することで解消していることが多く，例外処理に比べれば可能性は低い．

本論文では，字句列の共通性が高い条件文は例外処理部であるという仮定のもと，例外処理を抽出する方法を提案する．この方法では，条件文が持つ字句の並び，すなわち字句列がより多くの条件文に出現するかどうかで共通性を評価し，共通性の高い条件文を例外処理部として抽出する．また，仮定の妥当性を検証するために，オープンソースコードへの適用実験により，字句列の共通性の高さと例外処理部の関係を示す．ここで，字句列の共通性が高いとは，直感的には，条件文を構成する字句列が，他の多数の条件文にも出現していることを意味する．また，対象言語は，広く利用されている C 言語とする．C 言語の場合，例外処理を記述するための専用の構文要素が存在せず，一般的に，例外処理部は条件文で記述される．C 言語を対象とすれば，他の言語への適用は容易となる．

以下では，2 章で関連研究を議論したあと，3 章で共通性の評価指標とその求め方を示す．4 章では，オープンソースプログラムへの適用結果とそれに対する評価を示す．

2. 関連研究

Kang ら [3] は，エラー処理の実行経路は通常時に比べ，分岐点，文，関数呼び出しが少ないことに着目し，ソースコード中の例外処理に該当する実行経路を特定する手法とツール APEx を提案している．また，Baishakhi ら [4] は APEx によって推定された例外処理の仕様を用いて自動的に例外処理のバグを修正するツールを提案している．APEx の手法は，実験結果から有効性は示されているが，例外処理部から呼び出される関数まで解析しないと正確には求められない．また，正常処理部の方が短い可能性もあり，その場合には対応できない．

例外処理部を統一的に保守する方法として，アスペクト指向プログラミング環境を利用する方法がある．Fernando ら [5] は例外処理部を正常時の処理と分離することは保守性および再利用性の観点で重要であると主張している．また，パターン “Error Handling Aspect” により，例外処理部を正常時の処理と分離し，正常時の処理の保守性，および，例外処理部の再利用性を高められると主張している．本論文の場合は，アスペクト指向言語ではなく，C 言語を対象としているが，C 言語のソースコードに対する例外処理アスペクトのマイニングと捉えることができる．ただし，例外処理部をアスペクトとして抽出する研究は存在しない．

同じ字句列を含む条件文を求めるという意味では，コードクローン技術とも関係するが，本論文は，数個の字句列の連続に着目しており，連続する文の列を対象とはしていない．また，複数の条件文に出現する字句列に基づいて共通性の高さを定義するので，この点でもコードクローン検出とは異なる．

3. 字句列の共通性に基づく例外処理の抽出

3.1 手法の概要

本論文では，共通性の高い記述を持つ条件文は例外処理部であるという仮定をおく．この仮定のもとでは，条件文ごとの共通性の高さを評価し，共通性の高さの降順に並べ替えることで，上位には例外処理部が並ぶ．例外処理部を抽出するには，上位の条件文を取り出せばよい．そのためには，共通性を評価する指標が必要となる．以下では，その共通性を評価する指標として「共通性スコア (commonality score)」を定義する．

共通性スコアを求める全体の流れを**図 2** に示す．全体の処理の概要は以下の通りである．

(1) 対象となるソースコードを構成するソースファイル群を与えると，各ソースファイルに対し，構文解析をしたうえで，例外処理部候補となる if 文を抽出する．
(2) 各 if 文について，それを構成する字句列とその出現数を求める．
(3) すべての if 文の「字句列とその出現数の組」を集め，「字句列とその出現数の組」の出現数を求める．この出現数を字句に対するスコアとする．
(4) 各 if 文について，その if 文が持つ「字句列とその出現数の組」ごとの出現数，すなわち字句列に対するスコアを求め，それらの総和を求める．この総和が，if 文に対する共通性のスコアとなる．

次節以降では，例外処理部候補なる if 文や評価に用いる字句の定義や，字句に対するスコアや共通性のスコアの詳細について述べる．

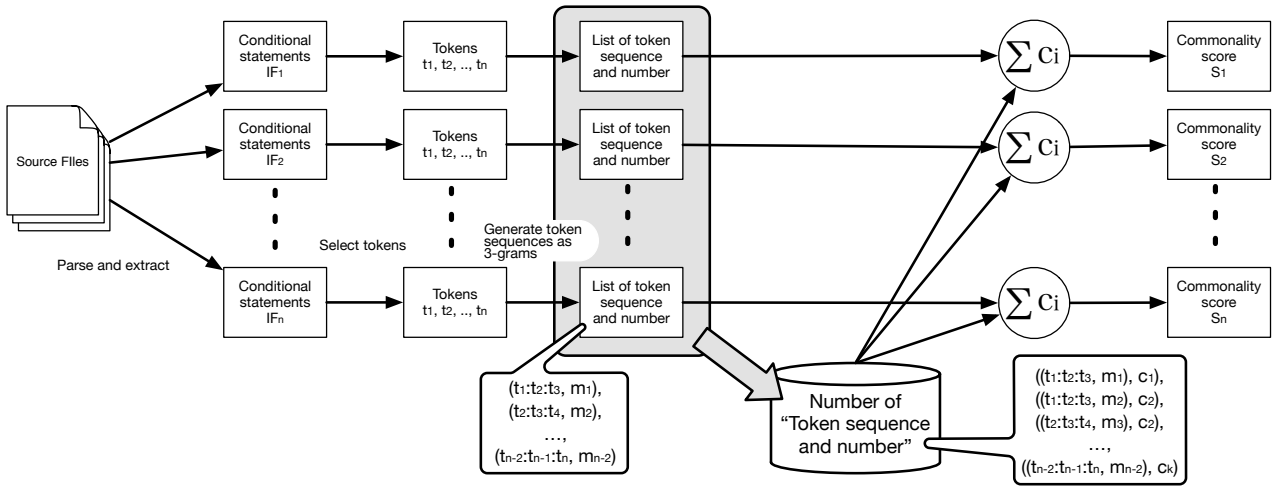

図 2 共通性スコアの算出の流れ
Fig. 2 The calculation flow of commonality score

3.2 条件文と字句列

本手法では，前提条件として，条件文のうち，内部に条件文を持たない最内のもののみを対象とする．これは，Kang ら [3] と同じく，例外処理は分岐が少ないという判断に基づく．ただし，最内ではない条件文への拡張する場合も，まず最内のものから判定し，構文木の根に向って候補を広げていく方法が考えられ，本手法が不可欠となる．また，条件文は if 文のみとし，then 節と else 節は分離したうえで，2 つの条件文があるもとして扱う．なお，エラーの状態により処理を変えるために内部に switch 文など，if 文以外の制御文を持つ if 文もあるが，稀であるので，その if 文も最内として扱う．

対象とする字句列は，条件文を構成する構文要素のうち，条件式の真偽に従って実行される文から取り出すものとする．条件式を含めない理由は，条件式には正常処理と同じ字句が含まれやすく，統一性がないことである．C 言語では，代入式が記述できるので，正常処理自体が条件式に含まれることがある．

文を構成する字句は，関数や変数，マクロなどの識別子だけでなく，予約語や演算子，括弧，コメントなど，複数の種類がある．例外処理の書き方は，規約等で統一されているが，細部では異なっているので，あらゆる字句を対象とすると共通性を見出せない．例外処理の場合，複雑な式は出現する可能性が低く，関数や変数など，意味に基づいた名前を持つ要素が共通していると考えられる．また，配列や構造体は，それ自体の参照は様々な条件文に出現すると考えられるが，添字やメンバ名は変動しやすいと考えられる．これらのことから，配列の添字内の式は無視し，それ以外の構文要素を構成する字句のうち，以下の字句を対象とする．

- 変数識別子，関数識別子，マクロ (変数名また関数名として出現するもの)
- 代入演算子 （=, += など）
- リテラルの種類 （数値, 文字, 文字列）
- ジャンプ命令の予約語 (return, break, continue, break)
- 式文の終り

ここでは前提として，前処理前のソースコードは構文解析をしたうえで，識別子については，変数，型，メンバについて区別できているものとする．

リテラルは 3 種類に分け，その種類名を字句とする．これは，例外処理によってリテラルの値は変わりやすいが，関数の特定の引数に用いられるなど，リテラルの出現の位置には共通性があるからである．

演算子のうち代入演算子だけを対象とする理由は，代入のみが変数の値を更新する副作用を持つからであり，関数と同じように処理の中で大きな意味を持つ．

ジャンプ命令は，例外処理に使われる典型の予約語であることから対象としている．

「文の終り」は，本来，字句としては存在しないが，式文の終りを表す特殊な字句が存在するものとして扱う．これは，文の並び順は，必ずしも重要ではなく，入れ替わる可能性があることから，字句列が前後の文との関係に強く依存することを避けるためである．なお，一般的には，式文の終りにはセミコロンが存在するので，セミコロンを対象とする方法もあるが，前処理前のソースコードにおいて，マクロの定義内にセミコロンが存在することでセミコロンが存在しない場合には対応できない[*1]．

字句列は，上記に示した種類の字句を条件文から取り出し，出現順に並べたうえで，前から 1 つずつずらしながら N 個を組み合わせたものとする．これは，自然言語処理などで用いられる N-gram と同じ考えである．なお，N が小さいと正常処理部との区別がつきにくくなり，N が大きい

[*1] 評価実験で示すように，前処理前のソースコードを構文解析することを前提とし，このような場合も文を識別できるものとする．

```
1 GRAM      <hash_free:wd_to_name:;>
1 GRAM      <wd_to_name:;:error>
1 GRAM      <;:error:.LIN.>
1 GRAM      <error:.LIN.:.LIN.>
1 GRAM      <.LIN.:.LIN.:_>
1 GRAM      <.LIN.:_:.LIS.>
1 GRAM      <_:.LIS.:;>
1 GRAM      <.LIS.:;:errno>
1 GRAM      <;:errno:=>
1 GRAM      <errno:=:.LIN.>
1 GRAM      <=:.LIN.:;>
1 GRAM      <.LIN.:;:return>
1 GRAM      <;:return:true>
1 GRAM      <return:true:;>
```

図 3 図 1 に対する字句列とその出現数
Fig. 3 Token seqeunces and the numbers of Fig.1

と，共通性が高い字句列が減りやすい．N の値について，明確な解は存在しないが，経験的に 3 のときに期待する結果が出やすいことから，本論文では $N = 3$ とし，字句列にはトライグラム（3-gram）を用いるものとする．

図 3 に，図 1 に対する字句列を示す．これは，提案手法を実装したツールが出力したものを，字句の出現順に並べ替えたものである．左から 3 列目の < と > に囲まれた部分が字句列で，コロンは字句を分けるメタ記号である．セミコロンは「文の終り」を表す特別な記号で，.LIN. は整数リテラル，.LIS. は文字列リテラルを意味する．なお，図 3 の 1 列目は後述する字句列の出現数であり，2 列目はツール内で他の情報と区別するためのタグである．条件式は含まないので，内部の 4 つの文が持つ 24 個の字句のうち，カンマや括弧などを除く 16 個の字句を対象として 14 個の字句列を出力している．同じ字句列が 2 つ以上出現していないので，1 列目はすべて 1 となっている．

3.3 字句列に基づく条件文の共通性

字句列（トライグラム）に基づく条件文の共通性は，多くの条件文に出現する字句列を多く含むものほど高くなるように定義する．そのためには，まず，字句列の出現数を求める必要がある．ここで，例外処理の典型的な書き方を考えたときに，エラーログを 1 つの出力命令で出力する場合と，複数の出力命令で出す場合に，後者の方が字句の出現数が多くなる．しかし，多くの例外処理部が 1 つの出力命令しか使っていないとしたら，前者の方が共通性が高く，後者の方は低くなるべきである．そこで，各条件文について，まず字句列とその出現数の組を求め，次に，ソースコードを構成するソースファイル群全体において，その組の出現数を数える．

例えば，fprintf 文で標準エラー（stderr）に出力する文が 2 つ存在する条件文の場合には

(<fprintf:stderr:.LIS.>, 2)

という組が求まる．さらに，この組を持つ条件文がソースファイル群全体に 5 個存在するなら，ソースコード全体の「字句列とその出現数の組」とその出現数を以下のように求める．

((<fprintf:stderr:.LIS.>, 2), 5)

なお，ここで求める「『字句列とその出現数の組』の出現数」は表現としてわかりにくいので，以下では，便宜上，「字句列スコア」と呼ぶ．これにより，例えば，fprintf 文で標準エラー（stderr）に出力する文を 1 つのみ持つ条件文では同じ字句列が出現するが，2 つ存在する場合とは区別される．

次に，各条件文について，字句列スコアの総和を求め，これを「共通性スコア」と呼ぶ．具体的には，各条件文ごとに，その条件文から求まるすべての「字句列とその出現数の組」に対し，ソースファイル群全体での出現数，すなわち字句列スコアを求め，さらにそれらの合計を求める．例えば，前述の fprintf 文で標準エラー（stderr）に出力する文が 2 つ存在する条件文の場合には，求まる字句列に <fprintf:stderr:.LIS.> が存在し，2 つ出現することから，5 というスコアが得られる．このスコアを総和を求めるとに用いる．

図 4 は，同じ関数が 2 回呼び出されている例である．これに対する字句列とその出現数を**図 5** に示す．1 列目が出現数で，3 列目が字句列である．なお，字句列の出現順に並べているが，error 関数の第 1 および第 2 引数が共通しているので，字句列 <error:.LIN.:.LIN.> の出現数が 2 となり，最初の出現の方に集約している．この例の場合は，argv の添字に式 “optind + 1” が存在するが，添字内は対象としないので，字句列にはこれらの字句は含まれない．

図 1 と図 4 の 2 つの条件文を例として，これらから共通スコアを求める過程を説明する．まず，それぞれの条件文から求まる図 3 と図 5 を組み合わせて字句列スコアを求めると，**図 6** になる．1 列目が，字句列スコアで，2 列目は条件文内での字句列の出現数，4 列名が字句列である．これは，図 3 や図 5 を結合したものを並べ替えてから，連続して出現する重複行を 1 つにまとめ，各行の先頭に重複した数を挿入したものである．例えば，図 1 と図 4 のどちらも，error 関数の式文が，別の文の後に出現している箇所が 1 つあるので，図 3 と図 5 には “1 GRAM <;error:.LIN.>” が 1 つずつ含まれ，合わせて 2 個となる．よって，図 6 では，出現数の 2 が字句列スコアとなる．

共通性スコアは，各条件文の字句列とその出現数の組に対し，図 6 の 2 列目から 4 列目が一致する行を求め，その 1 列目の字句列スコアを合計したものである．図 3 に対する共通性スコアは，14 個の字句列が存在し，うち 4 個の字句列スコアが 2 で，残りの 10 個は 1 なので，合計 18 となる．図 5 は字句列が 15 個存在し，同様に計算して 19 となる．

4. 評価と考察

4.1 評価方法

共通性スコアが高いものほど例外処理である可能性が高

```
    if (traditional && 1 < n_files)
      {
        error (0, 0, _("extra operand %s"), quote (argv[optind + 1]));
        error (0, 0, "%s",
               _("compatibility mode supports at most one file"));
        usage (EXIT_FAILURE);
      }
```

図 4　例外処理部の例: Coreutils-8.29/src/od.c
Fig. 4　An example of exception handling: Coreutils-8.29/src/od.c

```
  2 GRAM       <error:.LIN.:.LIN.>
  1 GRAM       <.LIN.:.LIN.:_>
  1 GRAM       <.LIN.:_:.LIS.>
  1 GRAM       <_:.LIS.:quote>
  1 GRAM       <.LIS.:quote:argv>
  1 GRAM       <quote:argv:;>
  1 GRAM       <argv:;:error>
  1 GRAM       <;:error:.LIN.>
  1 GRAM       <.LIN.:.LIN.:.LIS.>
  1 GRAM       <.LIN.:.LIS.:_>
  1 GRAM       <.LIS.:_:.LIS.>
  1 GRAM       <_:.LIS.:;>
  1 GRAM       <.LIS.:;:usage>
  1 GRAM       <;:usage:EXIT_FAILURE>
  1 GRAM       <usage:EXIT_FAILURE:;>
```

図 5　図 4 に対する字句列とその出現数
Fig. 5　Token sequences and the numbers of Fig.4

```
  1    1 GRAM  <.LIN.:.LIN.:.LIS.>
  2    1 GRAM  <.LIN.:.LIN.:_>
  1    1 GRAM  <.LIN.:.LIS.:_>
  1    1 GRAM  <.LIN.:;:return>
  2    1 GRAM  <.LIN.:_:.LIS.>
  1    1 GRAM  <.LIS.:;:errno>
  1    1 GRAM  <.LIS.:;:usage>
  1    1 GRAM  <.LIS.:_:.LIS.>
  1    1 GRAM  <.LIS.:quote:argv>
  1    1 GRAM  <;:errno:=>
  2    1 GRAM  <;:error:.LIN.>
  1    1 GRAM  <;:return:true>
  1    1 GRAM  <;:usage:EXIT_FAILURE>
  1    1 GRAM  <=:.LIN.:;>
  2    1 GRAM  <_:.LIS.:;>
  1    1 GRAM  <_:.LIS.:quote>
  1    1 GRAM  <argv:;:error>
  1    1 GRAM  <errno:=:.LIN.>
  1    1 GRAM  <error:.LIN.:.LIN.>
  1    1 GRAM  <hash_free:wd_to_name:;>
  1    1 GRAM  <quote:argv:;>
  1    1 GRAM  <return:true:;>
  1    1 GRAM  <usage:EXIT_FAILURE:;>
  1    1 GRAM  <wd_to_name:;:error>
  1    2 GRAM  <error:.LIN.:.LIN.>
```

図 6　図 1 と図 4 に対する字句列スコア (1 列目)
Fig. 6　Tokens sequeneces and the number of FIg.1 and Fig.4

いことを調べるために，次のように実験を行った．まず，提案手法に基づいて，対象となる条件文について，共通性スコアを求める．さらに，条件文を共通性スコアの降順に並び換える．

次に，実験対象のソースコードを精査し，最内の条件文のうち例外処理であることを特徴付ける字句の条件を定める．これは，実際に目で見て，できるだけ正確に判定できるように条件を定義する．この条件に基づいて，例外処理部となる条件文に印を付ける．

共通性スコアの降順に並べた条件文のリストの前の方に例外処理部と印を付けたものが並び，その後に例外処理部の印がつかないものが並べば，抽出できることがわかる．

表 1　例外処理部の判定条件
Table 1　Judgement condition of exception handling

文字列リテラル内の単語:	can, could, could't, could not, unable, invalid, fail, faild, err, error, exception
識別子の名前:	exit, abort, warning, error, usage, errno, stderr, EXIT_FAILURE
識別子の名前の末尾:	_die, _fatal

4.2　評価実験

Coreutils-8.29[1] を対象に実験を行なった．Coreutils を対象とした理由は，GNU のプロジェクトであり，例外処理が統一的に書かれていることが期待できることと，ツールセットになっていることから，擬似的に複数のアプリケーションを対象としたときの問題が見えてくる可能性があることなどである．なお，Coretutils には，テスト用のコードなども含まれるが，例外処理に統一性が出るように src ディレクトリのみを対象とする．src ディレクトリに含まれる最内の条件文は 4458 個である．

構文解析には TEBA[6] を利用した．TEBA は，前処理前のソースコードに対して構文解析を行う．前処理前のソースコードの場合，必ずしも正確に構文解析ができるとは限らないが，ヒューリスティックなルールで，字句を補完しつつ，構文と合わない箇所も近似的に解析する．また，識別子もその出現文脈に従って，型や変数，関数，メンバの区別を行う．

Coreutils-8.29 に対し，例外処理部かどうかの判定は，文の中に**表 1** の条件に合致する字句を持つかどうかで判定し，例外処理部以外のものが混合しないかは，目視で確認をしつつ，調整した．なお，単純に return のみなど，この条件で判定できないものは，例外処理部とはしなかった．

表 1 に基づいて例外処理部と判定した条件文と，例外処理部ではない，すなわち非例外処理部と判定した条件文について，それぞれの共通性スコアの分布を**図 7**，**図 8** に示す．また，これらでは，コマンドの使い方（usage）に関する処理の条件文を除いており，**図 9** に独立して示す．分離した理由は後述する．

それぞれの共通性スコアの最大値，最小値，平均値，および各処理部に該当した条件文の数を**表 2** に示す．条件文を共通性スコアの降順で並べたときに，非例外処理部は上

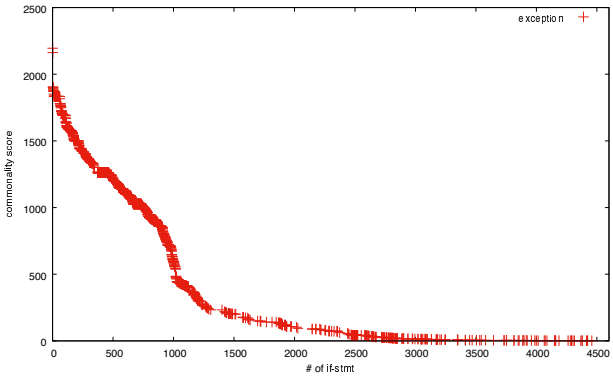

図 7　例外処理部の共通性スコアの分布 (Coreutils)
Fig. 7　Commonality scores of error handlings (Coreutils)

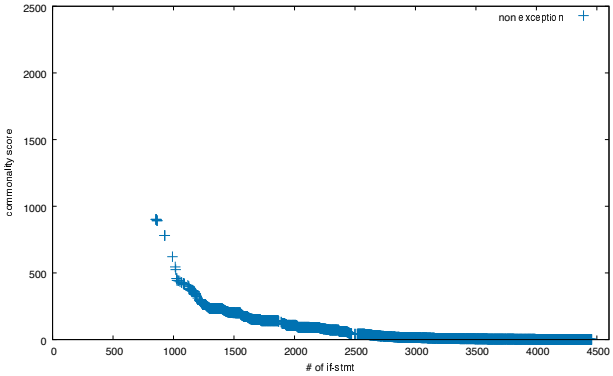

図 8　非例外処理部の共通性スコアの分布 (Coreutils)
Fig. 8　Commonality scores of non error handlings (Coreutils)

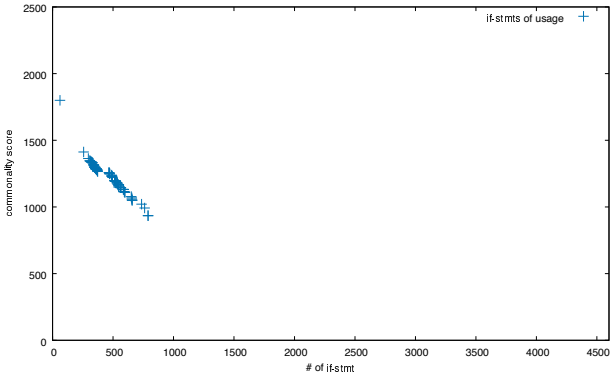

図 9　usage に関する条件文の共通性スコアの分布 (Coreutils)
Fig. 9　Commonality scores of usage (Coreutils)

表 2　例外処理部と非例外処理部の共通性スコア (Coreutils)
Table 2　Commonality scores (Coreutils)

種類	例外処理部	非例外処理部	usage
最大値	2195	901	1800
最小値	1	1	934
平均	874.9.8	71.1	1231.0
条件文の個数	1345	3020	93

位には出現していない．最大値は 901 であるが，上位の 11 個がやや外れて存在しており，12 番目（共通性スコア 622, 全体で 991 番目）から，ほぼ連続的に出現している．

例外処理部と判定した条件文（1345 個）については，上位の 1085 番目（共通性スコア 233, すべての条件文に対して 1301 番目）まではほぼ連続的に出現しており，例外処理部と判定した条件文全体の約 80.7%を占める．最初の非例外処理部より共通性スコアが大きいものに限定すると，789 個で全体の約 60.6%を占める．また，非例外処理の 12 番目より大きなものは 914 個で，全体の約 68.0%である．

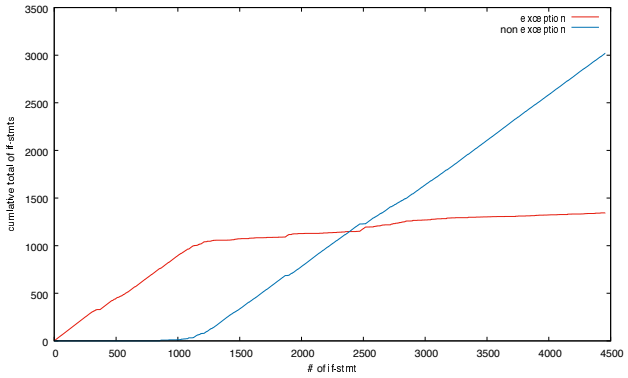

図 10　例外処理部と非例外処理部の出現累計 (Coreutils)
Fig. 10　Cumlative total of conditional statements (Coreutils)

共通性スコアの降順に並べた条件文について，上位から個数を累積したときのグラフを**図 10** に示す．おおよそ 1000 番目を越えるあたりまでは例外処理部が連続して増えているのに対し，非例外処理部は増えない．また，それ以降は例外処理部は増えず，非例外処理部が増えている．

これらのことから，上位には非例外処理部が出現せず，例外処理部の約 8 割が連続して上位を占めていることがわかる．また，例外処理部の約 7 割については，非例外処理部が連続するスコアより上位に位置している．完全な区別はできないが，典型的な例外処理部の書き方を確認したり，一時的に例外処理部を隠してプログラムの読む量を減らすといった用途には応用可能である．

使い方（usage）に関する条件文を分離した理由は，Coreutils がツールセットを構成しているので，usage に関する共通した処理が各ファイルに記述されており，それらが上位に出現するからである．図 9 では上位に固まって出現しているが，これは usage の書き方に統一性があるからである．当初，これらの条件文は非例外処理部に含まれており，例外処理部と非例外処理部の重なりを精査して見つけた．本論文では，例外処理部と非例外処理部の 2 種類に分離することを目的としているが，実際には，複数の処理に分類することが必要である．このような状況は，複数のアプリケーションを対象に共通性を評価した場合にも起こりうると想定される．

図 11 に，字句列スコアの上位のものを示す．この中には，error や errno といった例外処理に関わる字句が多く含まれている．一方，例外処理とは必ずしも関係しないと思われる字句も多い．例えば，識別子 “_” は，gettext 関数を呼び出すマクロであり，例外処理かどうかに関係なく

```
515    1 GRAM  <.LIN.:_:.LIS.>
358    1 GRAM  <_:.LIS.:;>
327    1 GRAM  <errno:_:.LIS.>
326    1 GRAM  <error:.LIN.:.LIN.>
303    1 GRAM  <.LIN.:.LIN.:_>
234    1 GRAM  <_:.LIS.:quoteaf>
233    1 GRAM  <return:false:;>
231    1 GRAM  <error:.LIN.:errno>
211    1 GRAM  <die:EXIT_FAILURE:errno>
199    1 GRAM  <=:.LIN.:;>
196    1 GRAM  <die:EXIT_FAILURE:.LIN.>
192    1 GRAM  <EXIT_FAILURE:.LIN.:_>
183    1 GRAM  <_:.LIS.:quote>
176    1 GRAM  <.LIN.:errno:_>
172    1 GRAM  <=:false:;>
148    1 GRAM  <=:true:;>
141    1 GRAM  <break:;>
134    1 GRAM  <;:return:false>
133    1 GRAM  <usage:EXIT_FAILURE:;>
131    1 GRAM  <EXIT_FAILURE:errno:_>
116    1 GRAM  <;:usage:EXIT_FAILURE>
114    1 GRAM  <errno:.LIS.:quotef>
108    1 GRAM  <return:true:;>
```

図 11　上位の字句列とその出現数
Fig. 11　Top list of token sequences and the numbers

文字列の出力で利用される．単純に，字句列ではなく，字句の出現数を基準に共通性を求めると，このような字句によって，例外処理と無関係な条件文の共通性スコアが高くなる．一方，最も値が大きい字句列は，出現数 1 の字句列 “.LIN.:_:.LIS.” で，字句列スコアが 515 である．図 8 からわかるように，非例外処理部については共通性スコアが 500 以下のものが大半であり，この字句列は含まれない．よって，この字句列は例外処理に特有と判断できる．このことから，例外処理に特有な識別子だけでなく，全体的に広く使われる字句やリテラルなどであっても，その組み合わせに例外処理に特有なものが存在することがわかる．

4.3　共通性スコアと文の数の比較

Kang ら [3] のアイデアに従うと，例外処理部となる条件文は短く，そうでないものは長くなる傾向にあることが期待でき，その場合は，例外処理部の抽出に利用できる．共通性スコアが上位の条件文は例外処理部であるので，上位と下位で傾向が異なるか，文の数を調べた．文の数には，式文だけでなく，制御文の出現数も含めている．**図 12** は，横軸を共通性スコアとし，その共通性スコアの条件文が持つ文の数をすべてプロットしたグラフである．なお，実験では，446 個の文を持つ条件文（共通性スコア 1800）が 1 つ存在したが，グラフを見やすくするために，外れ値として除外している．

共通性スコアの順位で 1000 番目の条件文の共通性スコアは 594 であり，約 600 前後を基準としても，例外処理部の方が短いという傾向は読み取れない．また，文の数が多いと，相対的に字句数が増えるので，共通性スコアの値を高くする悪影響を与える可能性がある．しかし，グラフからは，そのような傾向は読めず，文の数の影響はないと考えられる．例外は外れ値とした除外した条件文で，多数の前処理指令の分岐を含み，それぞれに fputs() の呼び出しがあるので，文の数に応じて共通性スコアが高くなっていた．

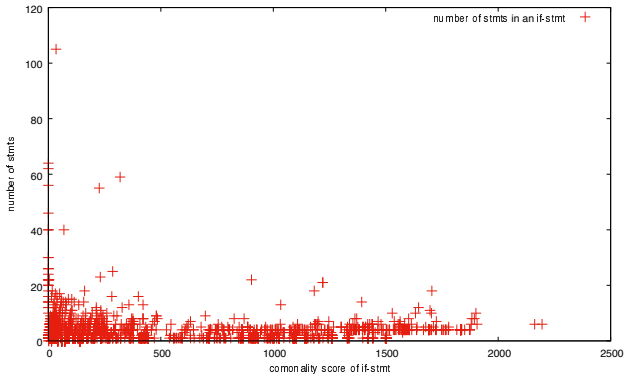

図 12　共通性スコアと文の数
Fig. 12　Commonality scores and numbers of statements

4.4　類似性に基づく例外処理部の分類

本論文では，条件文のうち最内の条件文のみに限定しているが，実用規模のソースコードでは，それでも数は非常に多い．例えば，Coreutils-8.29 の src ディレクトリだけで，4458 個存在し，Coreutils-8.29 全体では 9562 個存在する．ソースコード全体で，どのように例外処理が書かれているか，概要を知りたくても，1 つ 1 つ見ていくことは容易ではない．そこで，字句列に基づいた特徴ベクトルでクラスタリングすることで，類似した条件文のグループを作り，代表となる条件文を見ることで，全体の概要を理解できるようにした．

クラスタリングには，ツール bayon[7] を用いた．特徴ベクトルは，各条件文ごとに，出現数が 2 以上の字句列を次元とし，各次元の大きさはその字句列の出現数として定義した．また，適切なクラスタ数を決めることは難しいので，bayon の機能で分割の限界となる基準値（オプション limit 値）を 2.0 に設定した．これにより，4458 個の条件文が 225 クラスタに分割された．図 1 は共通性スコアが 1984 で，113 個の条件文のグループに，図 4 は共通性スコアが 1880 で，17 個の条件文のグループの代表となったものである．

クラスタを，それに含まれる条件文の共通性スコアのうち最大値を用いて降順に並べ，各クラスタごとに最大値と最小値を表すバーを表示したグラフを**図 13** に示す．先頭から 50 番目ぐらいのクラスタを見れば，例外処理の可能性が高い条件文を網羅できる．最大値と最小値に開きのあるクラスタがあるが，このようなクラスタには少数ではあるが類似性の低いものが混ざっている．クラスタリングツールの精度や分割数に依存しており，改善には至っていない．

このようなツールを使うことで，条件文の数が多くても，効率良く例外処理部を確認できる．このツールを用いたことで，使い方（usage）に関する条件文が存在することを発

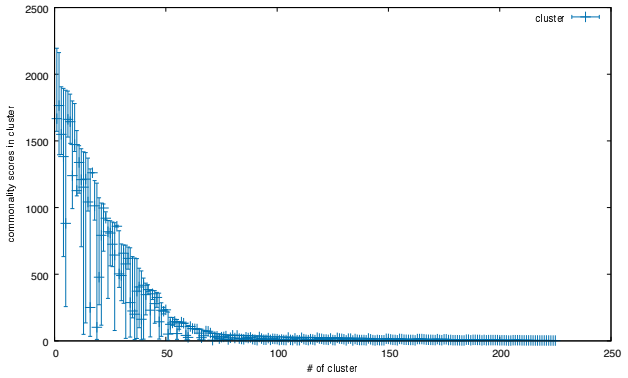

図 13 クラスタごとの共通性スコア
Fig. 13 Commonality scores of clusters

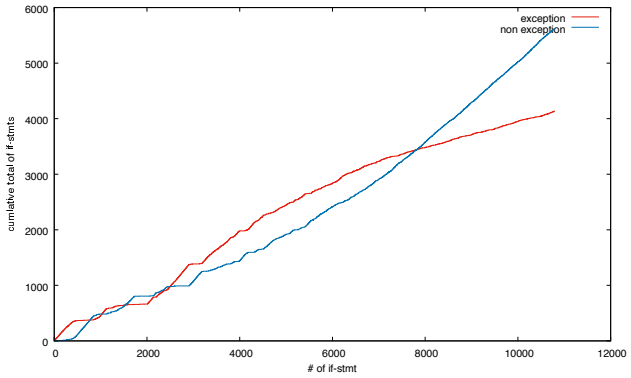

図 14 例外処理部と非例外処理部の出現累計 (Postfix)
Fig. 14 Cumlative total of conditional statements (Postfix)

見した．さらに，該当する条件文が 3 つのグループに集約されたので，それぞれ分離できるかも検討できた．また，条件文だけを見ても例外処理部かどうかわからないことがあるので，条件文が出現する前後の文脈も表示する機能も実装して利用した．

4.5 提案手法が適用しにくいソースコード

提案手法により，例外処理部が見つかる理由は，例外処理には何らかの規約が存在し，それに従って記述しているからである．したがって，学生が演習で記述するようなプログラムの場合には，統一的に記述されていない可能性が高く，適用できない．

一方，実用的なソースコードにおいても，提案手法が適用しにくい場合があると想定される．その一つは，処理全体がプロトコルに従って動作する状態機械になっている場合で，そもそも条件文が例外処理部と呼べるかどうかの判断も難しい．例えば，プロトコルに合わない入力に対する処理は，例外処理とすることもできるが，それ自体もプロトコルに対する仕様を実装したものとも捉えられる．

状態機械を実装した典型例のアプリケーションとして，Postfix 3.2.4[2] を対象として同様の実験を行なった．対象ディレクトリは src に限定し，対象となる条件文は 10795 個である．すでに述べたように，そもそも例外処理かどうかの判定が難しいが，Coreutils と同様にエラーに関する字句等を含む条件文を例外処理部として扱い，共通性スコアとの関係を調べた．**図 14** に示すように，共通性スコアの上位は例外処理部であったが，下位にも多く含まれており，coreutils ほど明確な区別はできなかった．共通性スコアが高いもので，例外処理部ではないものを調べると，状態を表現する変数への代入が多く見られ，状態遷移に関わる処理が共通性スコアを高めていた．

Coreutils と同様に，例外処理部以外で繰り返し出現するものとして，動作状況の出力（verbose モード）がある．Postfix の場合は，msg_info 関数が用いられることから，図 14 では，これを含む条件文は分離している．各条件文の共通性スコアの分布を**図 15**，**図 16**，**図 17** に示す．

Coreutils の場合は，個々のツールにおいて，オプションや環境が正常動作と異なる場合に例外処理を行うので，例外処理部が明確に区別される．よって，提案手法により，直感と合う結果を得られたと考えられる．

4.6 妥当性への脅威

実験では，共通性スコアの高い条件文が例外処理かどうかを評価するために，例外処理に特有な字句を持つかどうかで判定している．著者ら自身が，実際にソースコードを読んで誤りがないことは確認しているが，数が多いので見落しなどにより，本来例外処理部ではないものを例外処理部と扱っている可能性がある．また，逆に，例外処理部とすべきものを非例外処理部と扱っているものは存在する．これは意味的に判断が難しいものや，return 文だけなど，自動的に判定することが困難なものがあるからである．

共通性を求めるときに対象とする字句の種類は実験対象に依存しないよう一般性のあるものを選んでいる．しかし，検討するときに，Coreutils の実験結果を参考にしており，Coreutils への依存を完全には否定できない．

実験は，再現性を重視し，本論文で提示しているグラフも含め，すべての処理を自動化している．また，その結果については，細かく確認をしているが，ツールの不具合を見落している可能性はある．

5. おわりに

本研究では，ソースコード内の例外処理部を抽出するために，字句列に基づいて条件文間の共通性の高さに着目する方法を提案した．字句列は，最内の条件文から選択した字句から構成されるトライグラムであり，より多く出現する字句列を持つ条件文ほど，共通性が高いと定義した．Coreutils に対する適用実験により，共通性が高い条件文は例外処理部であり，また，非例外処理部は低くなることを確認した．一方，提案手法の限界として，ヘルプの出力といった，例外処理とは異なる処理のグループが上位に来

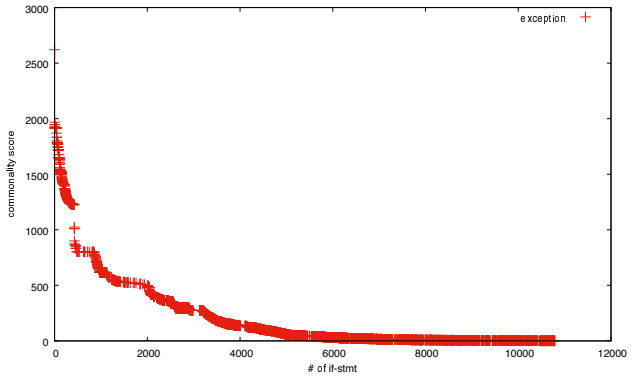

図 15　例外処理部の共通性スコアの分布 (Postfix)
Fig. 15　Commonality scores of exception handling (Postfix)

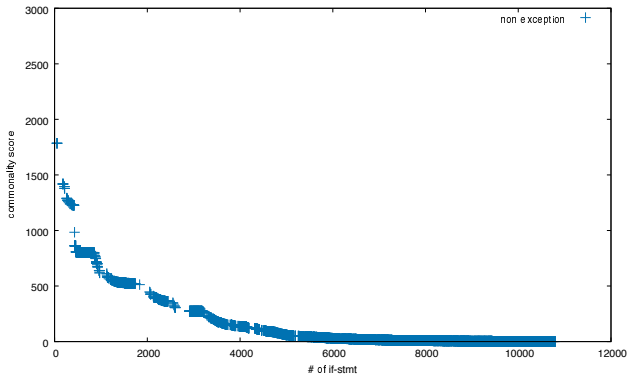

図 16　非例外処理部の共通性スコアの分布 (Postfix)
Fig. 16　Commonality scores of non exception handling(Postfix)

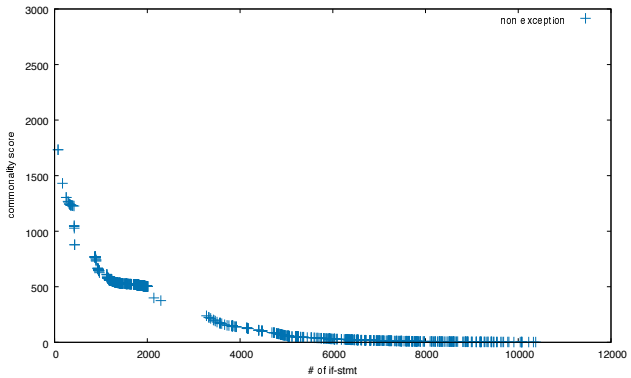

図 17　動作確認出力の共通性スコアの分布 (Postfix)
Fig. 17　Comonality scores of verbose mode (Postfix)

る可能性があること，Postfix のように状態遷移の条件文など，同じような非例外処理が出現するアプリケーションが存在し，そのソースコードにはうまく適用できないことを確認した．

今後の課題として，提案手法が有効となるアプリケーションの範囲の明確化と，クラスタリングなどを利用し，例外処理とそれ以外という 2 つではなく，複数のグループへの分割方法の検討が挙げられる．また，共通性の定義を改善して精度を上げることも必要である．本論文では，条件文の字句の並びからトライグラムを構成したが，その並びには構文木の構造が反映されていない．構文木の親子関係を表現するような字句列を採用するといった検討が必要である．また，共通性の評価には，条件文の文の字句のみを使用したが，条件式や前後の文脈などにも対象を広げることも検討が必要である．

謝辞

本研究の一部は，JSPS 科研費 17K00114，17K01154，17K12666，18K11241，2018 年度南山大学パッヘ奨励金 I-A-2 の助成を受けた．

参考文献

[1] “Coreutils - GNU core utilities.”, https://www.gnu.org/software/coreutils/coreutils.html
[2] “The Postfix Home Page”, http://www.postfix.org.
[3] Kang, Yuan, Baishakhi Ray, and Suman Jana, “APEx: Automated inference of error specifications for C APIs,” Proc. of the 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, 2016.
[4] Tian, Yuchi, and Baishakhi Ray, “Automatically diagnosing and repairing error handling bugs in C,” Proc. of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, 2017.
[5] Fernando Castor Filho, Alessandro Garcia, and Cecília Mary F. Rubira, “Error handling as an aspect,” Proc of the 2nd workshop on Best practices in applying aspect-oriented software development (BPAOSD ’07), ACM, 2007, New York, NY, USA, Article 1, DOI=http://dx.doi.org/10.1145/1229485.1229486.
[6] 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.
[7] “軽量データクラスタリングツール bayon”，http://alpha.mixi.co.jp/entry/2009/10714/