例外処理のコーディング規約の理解のための共通性に基づく条件文選別手法の提案

M2016SE003 加藤大貴
指導教員：吉田敦

1　はじめに

開発者は，プログラムの実行時に異常が発生することを想定し，例外処理を記述する．高い品質のプログラムを作るためには，アプリケーションのコーディング規約(以下，規約) や API の副作用を考慮して記述すべきである．規約は例外処理の内容やその実現方法を定めたものである．例外処理が正しく記述されていることを確認するには起こりうる例外に対するテストケースを作成する必要があるが，例外を想定することは困難であり，正常時の処理に比べてテストケース漏れが生じやすい．さらに，規約自体はテストケースでは検査できないことが多い．

開発者が例外処理の規約を知らない場合，API のマニュアル等の文書や，既存のソースコードを参照する．しかし，文書には十分な情報がないことが多い．

本研究では，ソースコードからの規約の理解を支援するために，例外処理を記述した条件文の共通性に着目し，条件文の集合から例外処理である可能性が高いものを選別する方法を提案する．また，例外処理は数が多いことや，異なる規約に従って書かれたものがあるので，条件文を整理して提示することで，理解支援も行う．

対象とするプログラミング言語を，C 言語とする．OS や組み込みシステムなど多くのシステムで採用されているからである．また，一般に例外処理は if 文で記述されることから，対象とする条件文は if 文のみとする．ただし，switch 文など他の構文にも拡張は可能である．

開発者の規約の理解に適した例外処理を提示するには，2 つの課題がある．一つ目の課題は，ソースファイル中の条件文から例外処理を区別することが困難なことである．if 文は正常時の処理にも使われるので，例外時との区別が必要である．区別の方法として，強制終了関数や標準エラー出力など，例外処理の典型的な処理やデータを含むかどうかで判断する方法が考えられるが，この方法は確実ではない．例えば，cURL[1] では，正常時の処理である進捗表示に標準エラー出力を用いる．また，例外処理であってもそれらに典型的に用いられる関数を含まないものも存在する．例えば，アプリケーション独自の例外処理専用の関数を用意している場合である．

二つ目の課題は，1 つのアプリケーションに含まれる例外処理の数が非常に多く，それらを網羅的に見て規約を理解することが困難なことである．そのためには，if 文の数を絞って見せる必要があるが，規約は 1 つとは限らないので，規約に対して網羅的になるよう厳選する必要がある．

一つ目の課題に対しては，例外処理の共通性に着目する．一般に，正常時の処理内容はその目的に応じて異なることが多いが，例外処理は規約に従うので，エラーメッセージの出力，リソースの解放といった典型的な処理が記述される．これに着目し，多くの if 文に共通する字句を含む if 文を例外処理とみなす．それらの中には，例外処理でないものが存在するが，例外処理は特に高い共通性を持つという前提を置き，共通性の高いものから if 文を提示する．そのために，if 文と他の if 文の共通性の指標として条件文共通度を定義する．

二つ目の課題に対しては，クラスタリングを行って似た if 文をまとめ，それぞれのクラスタから 1 つずつ if 文を提示する．これにより，if 文が持つ規約の種類を保ちつつ見るべき if 文の数を減らす．

本稿では，第 2 章で関連研究について述べ，第 3 章で例外処理および本研究で対象とする if 文について述べる．第 4 章で提案手法について述べる．第 5 章で提案手法の評価と考察を述べ．第 6 章でまとめと今後の課題を述べる．

2　関連研究

C 言語の例外処理を特定し，仕様の抽出を行う研究もある [2][3]．これらの研究では実行経路が短い分岐経路を例外としているが，return などで明示的に経路が終わらないと判別できない．本手法は共通性に基づくので，実行の終了が関数に隠蔽されていても例外処理を見つけられる．ただし，規約に従ったものしか見つけられない．

ソースコード中に共通して現れる処理に着目する技術に，アスペクト指向プログラミング (以下，AOP)[4] がある．AOP では，複数のソースコードに散在する処理を横断的関心事と呼び，それらを 1 つのモジュールに分離することで保守性を高める．本研究は，AOP における横断的関心事の抽出を行っているといえる．AOP によって例外処理を記述する技術 [5] は存在するが，AOP の観点から例外処理を見つける研究は存在しない．

ソースコードの共通性に関連した技術として，コードクローン検出がある [6]．コードクローン関係にあるコード片の集合をコードクローンセットと呼ぶ．コードクローンセットを生成することと，本研究において if 文のクラスタを構築することは複数の if 文から似たものを集めるという点で類似している．ただし，コードクローン検出技術が 2 つのソースコード片に共通した記述に着目した技術であるのに対し，本研究では 2 つ以上のソースコード片に共通した記述に着目している．

3　例外処理

例外処理とは，機器の故障やユーザからの不正な入力といった処理の正常な実行に支障をきたす要因を引数や戻り値の確認によって検知し，処理が正常に実行されるよう変数の再設定などの回復を行う処理，もしくは，処理の実行を中断する処理のことを指す．例外処理は，速

やかに終了または復帰するために，複雑な制御は持たないので，複数の if 文がネストされた if 文が存在する場合，最も内側の if 文のみを例外処理の候補とし，外側の if 文を無視する．これは，一般的に例外処理の多くはネスト構造を持たない if 文で記述されるという前提に基づく．また，例外処理であるものが一部抜けても，全体として規約の理解は可能である．

4 共通的特徴に基づく例外処理の選別手法

4.1 選別手法の概要

入力として開発者から例外処理の候補となる if 文を含むソースファイル群を受け取り，入力されたソースファイル中の if 文とその条件文共通度を出力する．

選別手法の概要を図 1 に示す．具体的には，以下の手順で選別する．

1. ソースファイルを構文解析し，if 文を抽出
2. 抽出した if 文中の字句から，N-gram を生成
3. if 文ごとに N-gram とその出現回数の組を構成
4. 3 で得られた各組に対して，全 if 文中での出現回数に基づき重みを計算
5. 3 で得られた各組に対する 4 で求めた重みを合計することで if 文の条件文共通度を計算する
6. 条件文共通度の高い順に，if 文とその条件文共通度を開発者に提示する

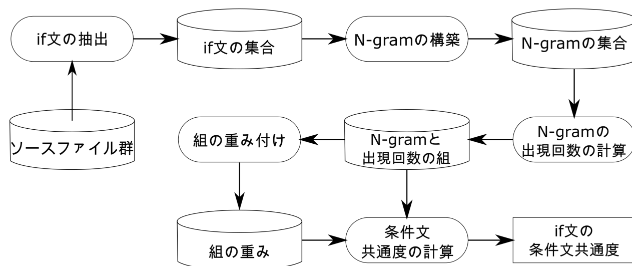

図 1 選別手法の概要

4.2 if 文の抽出

ソースコードを構文解析し，ネストの最も内側の if 文を取り出す．ただし，else 節がある場合は then 節と else 節をそれぞれ単体の if 文として分割して扱う．なお，どちらかは正常時の処理であるが，正常時と例外時の処理が混在していることを前提としているので問題ない．

4.3 条件文共通度の計算

抽出した if 文を選別する基準として，他の if 文との処理内容の共通性を表す指標,「条件文共通度」を定義する．まず，if 文に記述された処理の構成要素に対し，どれだけの if 文に共通して出現するかを表す「重み」を定義し，if 文ごとにその重みの合計を求めたものを「条件文共通度」とする．ここで，if 文の構成要素として字句を用いた場合，正常時，例外時を問わず多くの if 文に現れる字句の重みが極端に大きくなる．そこで，字句が出現する文脈を考慮することで同じ字句でも正常時と例外時に差が生じるようにする．具体的には字句の N-gram を用いる．ただし，N-gram を構成要素とした場合，同じ記述を繰り返す if 文の重みが大きくなる．そこで，重みを N-gram とその if 文中での出現回数の組に対して与える．N-gram を出現回数ごとに区別し，if 文の共通性に基づき高い条件文共通度を与えて優先的に提示する．

4.3.1 N-gram

共通度では，構成要素として if 文の then 節もしくは else 節中に連続して現れる字句を 1 つの組とした N-gram を用いる．条件式は異なる規約でも同じ書き方をするものが多く，個々の規約に特有な字句を抽出することが難しいので，条件式中の字句は構成要素として用いない．異なる規約で同じ書き方をする条件式の例に，NULL チェックがある．共通する処理を文脈として扱いつつ，細かなパラメータの差異を無視するために，対象とする字句を次のように定める．ただし，式文の終了は処理の区切りを表し，構文解析から得られる仮想的な字句である．

- 識別子: 関数名，変数名，マクロ名
- 予約語: continue，return，break，goto
- 式文の終了

4.3.2 N-gram とその出現回数の組に対する重み

「重み」は，N-gram とその if 文中での出現回数の組に対して与えられる．N-gram を出現回数によって区別することで同じ記述の繰り返しによる共通度の上昇を防ぐ．if 文 i から求められるすべての N-gram を集めた多重集合 G の要素 g に対して，g と等価の N-gram の出現回数 n を求め，組 (g, n) を得る関数 $pair(i, g)$ を以下の通り定義する．2 つの N-gram が等価であるとは，それぞれの N-gram が同じ字句によって構成され，その順序も等しいことをいう．

$$pair(i, g) = (g, \#\{g|g \in G\})$$

また，if 文 i 中のすべての組の集合を得る関数 $pairs(i)$ を，以下のように定義する．

$$pairs(i) = \{pair(i, g)|g \in G\}$$

また，組 $p = (g, n)$ に対する重みを $weight(p)$ と表し，入力のすべてのソースファイルから抽出した if 文の集合 I を用いて以下のように定義する．

$$weight(p) = \#\{i \in I|p \in pairs(i)\}$$

4.3.3 条件文共通度の定義

if 文 i に対する条件文共通度 S_i は，i の中の (N-gram, 出現回数) の組の重みを合計した値で，以下のように定義する．

$$S_i = \sum_{p \in pairs(i)} weight(p)$$

4.4 クラスタリング

if 文の数が多く，かつ複数の規約が混在するので似たものをまとめて提示することで規約の理解を支援する．そのために，クラスタリングにより，似たものをクラスタにまとめ，代表となるものを 1 つ提示する．各クラスタの最も高い条件文共通度をもつ if 文を代表，その条件文共通度をクラスタの条件文共通度とし，クラスタを条件文共通度順に提示することで，より典型的なものを確認できるようにする．

クラスタリングには，if 文間の類似度の定義が必要である．ここでは，例外処理の特徴に基づいて類似度を定義するために，if 文をそこから求められる N-gram に対応する成分を持つベクトルで表現する．ベクトルの各成分は N-gram とその if 文中での出現回数の組の重みである．成分を組ごとではなく N-gram ごとにすることでベクトルが細かく分かれることを防ぎ，開発者の見るべきクラスタ数を減らす．

クラスタ数は利用者が調整しなければならないが，Repeated Bisection 法 [7] にはクラスタの凝集度の閾値を指定する方法があり，適切な分割度を決めやすい．

4.5 選別の例

手法を coreutils[8] に適用したときの出力の 3 位から 5 位までを抜粋したものを図 2 に示す．なお，紙面の都合でインデント幅の調整，改行の追加をしている．図のようにクラスタを代表する if 文を条件文共通度の高い順に提示し，典型的な if 文を優先して提示する．

```
===  File: ./gnulib-tests/test-vprintf-posix/if-001.json,
     Cluster:1140 (size: 96)  (3/1287)
===  Score: 856.000
       (ave=841.333, min=322.000, max=856.000)
===  Similarity: 1.000
       (ave=0.989, min=0.607, max=1.000)
if (!(expr))                       \
{                                  \
  fprintf (stderr,
  "%s:%d: assertion failed\n",
    __FILE__, __LINE__);           \
  fflush (stderr);                 \
  abort ();                        \
}

===  File: ./src/touch/if-006.json,
     Cluster:1269 (size: 49)  (4/1287)
===  Score: 759.000
       (ave=674.878, min=532.000, max=759.000)
===  Similarity: 0.984
       (ave=0.979, min=0.908, max=0.994)
if (close (STDIN_FILENO) != 0)
{
  error (0, errno, _("closing %s"), quote (file));
  return false;
}

===  File: ./src/du/if-029.json
     Cluster:709 (size: 20)  (5/1287)
===  Score: 684.000
       (ave=410.750, min=337.000, max=684.000)
===  Similarity: 0.825
       (ave=0.967, min=0.825, max=0.995)
if (optind < argc)
{
  error (0, 0, _("extra operand %s"), quote (argv[optind]));
  fprintf (stderr, "%s\n",
_("file operands cannot be combined with --files0-from"));
  usage (EXIT_FAILURE);
}
```

図 2 共通性分析対象選択後の出力

4.6 共通性分析対象の選択のための拡張

第 4.1 節ではソースコード内のすべての if 文を対象としているが，開発者はアプリケーションや API 特有の規約を調べたい場合がある．そこで，特定の基準により共通性分析対象の if 文を選択し，限定的に if 文を分析する方法を提供する．選択の方法として，特定のアプリケーション内の if 文に限定する方法と，特定のソースファイル群内の if 文に限定する方法がある．

5 評価

5.1 条件文共通度による例外処理の判別の精度

例外処理である if 文に相対的に高い条件文共通度が与えられることを調べ，提案手法で例外処理と正常時の処理を選別できることを確認する．

実験対象は coreutils[8]，musicfs[9]，dfsch[10] とし．これらの条件文から経験的知識に基づき例外処理か否かを判別し，判別結果と条件文共通度の関係を調べた．例外処理を判別するために，対象のソースファイルから手作業によっていくつかの例外処理を抽出し，例外処理に多く現れる特徴を求めた．

musicfs のクラスタに与えられた条件文共通度とクラスタの順位の関係を図 3 に示す．例外処理のクラスタが上位になることが確認できた．また，上位 3 位のクラスタは所属する 8 個の if 文中 5 個が，4 位のクラスタは所属する 8 個の if 文中 7 個が例外処理であった．また，coreutils についても同様に例外処理が上位となった．dfsch については，正常時の処理のクラスタが上位となった．これは dfsch の return 文に N-gram の構成要素でない字句が多く現れ，return を含む N-gram とその出現回数の組の重みが大きくなったことが原因と考えられる．

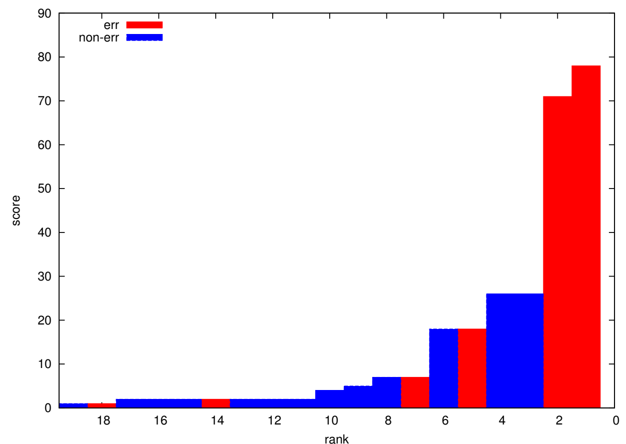

図 3 musicfs の条件文共通度と順位の関係

5.2 処理対象の選択方法についての評価

開発者が指定したディレクトリ内の if 文のみを共通性分析対象とすることで，特定の規約に従って書かれた if 文が上位に現れることを確認する．実験には coreutils[8] を用いる．coreutils のディレクトリのうち，gnulib-test にはテスト用ソースファイルが存在していることから，テ

スト用のマクロ ASSERT が多く用いられる．実験では，共通性分析対象を gnulib-test に限定することでそれらがより上位に提示されることを確認する．共通性分析対象限定後の出力の一部を図 4 に示す．なお，紙面の都合でインデント幅の調整，改行の追加をしている．共通性分析対象を選択しなかった場合の出力結果は図 2 である．共通性分析対象を選択しなかった場合，ASSERT が記述された if 文の中で最も順位が高いのは 49 位のクラスタであったが，選択により 4 番目となり，開発者はより確実にテストにおける規約を把握できる．また，coreutils の lib ディレクトリに見られる規約である errno の値を別の変数に保存し，return の直前で errno の値を復元する処理が見られるコードは共通性分析対象の選択によって 16 位から 1 位になった．

```
===  File: ./test-strtod/if-006.json,
     Cluster:31 (size: 5)  (4/65)
===  Score: 116.000
       (ave=107.600, min=103.000, max=116.000)
===  Similarity: 0.988
       (ave=0.988, min=0.983, max=0.992)
if (input)
{
  char *ptr;
  double result;
  input[0] = '-';
  input[1] = '0';
  input[2] = 'e';
  input[3] = '1';
  memset (input + 4, '0', m - 3);
  input[m] = '\0';
  errno = 0;
  result = strtod (input, &ptr);
  ASSERT (result == 0.0);
  ASSERT (!!signbit (result) ==
    !!signbit (-zero)); /* IRIX 6.5, OSF/1 4.0 */
  ASSERT (ptr == input + m);
  ASSERT (errno == 0);
}
```

図 4 共通性分析対象選択後の出力

5.3 考察

5.3.1 手法の妥当性

提案手法により，共通度に着目することで例外処理を提示できることがわかった．しかし，例外処理と同じクラスタに正常時の処理が所属し，クラスタの代表になることで，開発者が例外処理を確認できない可能性があった．また，dfsch を用いた実験ではリテラルを扱わないことが原因で戻り値を区別できず，正常時のクラスタのスコアが高くなった．これらのことから重みの定義をより洗練する必要がある．例えば，開発者から，提示された if 文が例外処理か否かの情報を受け取り，それをもとに正常時の if 文から得られる組の重みを下げることや例外処理に含まれる字句を新たに N-gram の構成要素に加えることなどが考えられる．

5.3.2 if 文以外の例外処理

本手法では C 言語における多くの例外処理が if で記述されることから，if 文による例外処理のみを対象とした．しかし，例外処理には while 文や switch 文を用いたものが存在する．例えば，プログラムがビジー状態にあるとき，while 文を用いてビジー状態が解消されるまで実行を試みる処理があげられる．また，switch 文を用いた例外処理の例として，複数のエラーコードを返す API のそれぞれの処理に対する case を記述したものがあげられる．よって，それらの構文に対して手法を拡張し，規約の理解に対する有効性を評価する必要がある．ただし，while 文は if 文と類似した構造を持っているので，手法の拡張は容易である．また，for 文は while 文に，switch 文は if 文に変換可能であるので，手法の適用は容易である．

6 おわりに

開発者の規約の理解を支援するために，if 文の選別手法と if 文を整理する手法を提案した．また，提案手法の有効性と問題点を実験によって確認した．今後の課題として，判別の精度の向上および，if 文以外での例外処理を推薦する方法の提案があげられる．

参考文献

[1] “cURL command line tool and library for transferring data with URLs.”, https://curl.haxx.se

[2] K. Yuan, B. Ray, and S. Jana. “APEx: Automated inference of error specifications for C APIs.”, Proc. of the 31st IEEE/ACM Inter. Conf. on Automated Software Engineering. ACM, 2016.

[3] T. Yuchi, and B. Ray. “Automatically diagnosing and repairing error handling bugs in C.”, Proc. of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, 2017.

[4] 千葉滋，“アスペクト指向入門”，技術評論社，2005

[5] F. C. Filho, A. Garcia, and C. M. F. Rubira. “Error handling as an aspect.”, In 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] 肥後芳樹, 楠本真二, 井上克郎. “コードクローン検出とその関連技術”, 電子情報通信学会論文誌 D 91.6 (2008): 1465-1481.

[7] Z. Ying, and G. Karypis. “Comparison of agglomerative and partitional document clustering algorithms.”, No. TR-02-014. MINNESOTA UNIV MINNEAPOLIS DEPT OF COMPUTER SCIENCE, 2002.

[8] “Coreutils - GNU core utilities.”, https://www.gnu.org/software/coreutils/coreutils.html

[9] “musicfs is a FUSE module implementing a media filesystem in userland.”, https://github.com/orbekk/musicfs

[10] “dfsch Scheme-style LISP language implementation with simple C interface.”, https://github.com/adh/dfsch