修士学位論文

題目

識別子の共起関係を用いた類似コード検索手法の提案と実現

指導教員
井上 克郎 教授

報告者
服部 剛之

平成 20 年 2 月 8 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 19 年度 修士学位論文

識別子の共起関係を用いた類似コード検索手法の提案と実現

服部 剛之

内容梗概

ソフトウェア保守を困難にする要因の 1 つとして類似コード（コードクローン）が指摘されている．類似コードは，コード片（ソースコードの一部）のコピーとペーストや定型処理等が原因で生成される．修正するコード片に類似コードが存在していると，全ての類似コードに対して同様の修正を検討する必要がある．しかし，人手で類似コードを探す作業はコストが大きく困難な作業である．

類似コードの修正作業を支援するために，grep 等のキーワード検索ツールを用いて類似コードを自動的に検索する方法が考えられる．しかし，grep には，キーワードをクエリ（検索質問）として考える必要があるという問題点がある．また，著者が所属する研究グループでは，以前に，字句解析後に得られる連続したトークン列の等価性に基づきコード片を検索するツール Libra を提案している．Libra はクエリとしてコード片を与えることで検索が可能であり，キーワードを考える必要がないという利点がある．しかし，コード片に例外処理等が挿入されていると検索結果に含まれなくなるため，検出漏れが多いという問題点がある．

本稿では，これらの問題点を解決するために，識別子の共起関係を用いて類似コードを検索する手法を提案し，実装を行った．本手法では，コード片からキーワードを自動的に抽出するため，キーワードを考える必要がない．また，識別子に着目して類似コードを検索するため，コード片に挿入等が行われていても検索に含めることができる．さらに，提案手法の有効性を評価するため，クエリに欠陥を含むコード片を与え，同じ内容の欠陥を含む類似コードを検索する実験を行った．その結果，提案手法は，クエリと同じ内容の欠陥を含むコード片の多くを検索結果に含めることができた．

主な用語

ソフトウェア保守
類似コード
情報検索
欠陥検出

目 次

1 まえがき

近年，ソフトウェアの大規模化・複雑化に伴い，ソフトウェアの保守作業を効率化することが重要な課題となってきている．ソフトウェアの保守とは，“納入後，ソフトウェア・プロダクトに対して加えられる，フォールト修正，性能または他の性質改善，変更された環境に対するプロダクトの適応のための改訂”であると定義されている [8].

ソフトウェア保守を困難にする要因の 1 つとして類似コード（コードクローン）が指摘されている [1][2][11][13]．コードクローンは，ソースコードの一部である**コード片**のコピーとペーストによる再利用や，定型処理の実装などによって作成される [12]．ソフトウェア保守を行う際に，複数個所に同様の修正を加える必要が生じることがある [7]．例えば，ソースコード中に欠陥が見つかった場合，その欠陥を含むコード片の類似コードを探し，検査する必要がある [7][10][16]．また，ある機能を追加するために，複数の類似コードを修正する必要が生じることがある．しかし，ソフトウェア中の類似コードを人手で探すことはコストが大きい．例えば，同一処理を実装したコード片であっても表現上の差異があることが多く，全ての類似コードを人手で探すことは困難である．特に，大規模ソフトウェアが対象になると，ソースコードの量が膨大となり，全ての類似コードを探すことはより困難となる．

このため，一般のソフトウェア開発者は grep[6] などを利用したキーワード検索を用いて，類似コードを検索すると考えられる．具体的には，対象となる類似コードに含まれると予想されるキーワードを考え，検索を行う．しかし，適切なキーワードを発見することは難しく，予想したキーワードによっては，全ての類似コードを検出することができないという問題点がある．また，著者が所属する研究グループが以前に提案したコードクローン検索ツール Libra[17] を用いた方法がある．Libra は，コードクローン検出ツール CCFinder[11] が検出する**コードクローン関係**に基づく検索を行うツールである．しかし，検索したい類似コードが，コードクローン関係になければ検索対象にならないという問題点がある．例えば，あるコード片に対して，コードの追加や削除が行われた類似コードは，CCFinder ではコードクローンとして検出することができない．そのため，Libra の検索対象に含まれない．

本稿では，キーワードを自動的に発見し，類似コードの検索を行う方法として，コード片を入力として，識別子の共起関係に基づいて類似コードを検索する手法を提案する．具体的には，まず類似コードをクエリ（検索質問）として選択する．次に，クエリから頻繁に出現している識別子（特徴語）を自動的に抽出する．そして，抽出した特徴語を含むコード片を類似コードとしてソフトウェア中から検出する．本手法では，コード片からキーワードとなる特徴語を自動的に抽出するため，キーワードを考える必要がない．また，識別子に着目して類似コードの検出を行うため，コード片の追加，削除が存在していても検出結果に含めることができる．

適用実験として，提案手法のクエリに欠陥を含むコード片を与えることで，ソフトウェア中に存在する同じ内容となっている欠陥の検出を行った．実験対象は，C 言語で開発されたソフトウェア 2 つに対して行った．実験を行った結果，クエリと同じ内容となっている欠陥をほぼ漏れなく検出することができた．欠陥を漏れなく検出するという観点では，提案手法は有用であることが期待される．

以降，2 節では，ソフトウェア保守の課題と類似コードが引き起こす問題に関して述べ，既存の類似コード検索における問題点を挙げる．3 節では，識別子の共起関係を利用した類似コード検索法を提案し，4 節では，提案手法を実装したシステムについて述べる．5 節では，2 つのソフトウェアを対象として，ツールを欠陥検出に適用した実験結果について述べる．6 節では，5 節の実験結果，及び提案手法の利用法について考察を行い，提案手法の有用性について評価を行う．最後に 7 節では，まとめと今後の課題について述べる．

2　背景

2.1　ソフトウェア保守と類似コード

2.1.1　ソフトウェア保守

ソフトウェア保守は目的毎に 4 つのカテゴリに分けられている [8].

修正を目的とした保守 (Correction)
: 発見された問題を修正するために，納入後に実施されるソフトウェア・プロダクトの対処的な改変

適応を目的とした保守 (Adaption)
: 変化した，または変化しつつある環境において，ソフトウェア・プロダクトを引き続き使用可能な状態を維持するために，納入後に実施されるソフトウェア・プロダクトの改変

完全化を目的とした保守 (Perfection)
: 性能または保守性を改善するために，納入後に実施されるソフトウェア・プロダクトの改変

予防を目的とした保守 (Prevention)
: ソフトウェア・プロダクト中に存在する潜在的なフォールトが，効果的なフォールトに転じる前に，それを検出し，修正するために，納入後に実施されるソフトウェア・プロダクトの改変

ソフトウェア保守は，ソフトウェアライフサイクルコストの大きな部分を占めている [5]. ライフサイクルを考えるとき，その費用と労力から見て保守は非常に大切な工程である．しかし，ソフトウェア保守における問題は多くあり，Dorfman らは，保守に関しては投資効果が明らかにならないので，常に資源を獲得するための戦いが起きると述べている [4]．資源を巡って争うということが常に存在し，次のソフトウェア・プロダクトに対するソースコードを作成しながら，将来の納入計画を決め，現在のソフトウェア・プロダクトに対して緊急的なパッチを施すということは難題である．また，ソフトウェアを開発したチームは，ソフトウェアが運用に入ると必ずしも保守を担当するとは限らない．自分の開発したものではない，大規模なソースコードに潜む欠陥を発見しなければならないということは，保守者にとっては非常に難題である．

多くの場合，独立したチームが，ソフトウェアが適切に運用され，ユーザの変化するニーズを満たすように進化させられていることを確実にするために雇用されているが，保守のア

ウトソーシング (社外調達) も，主要な産業になりつつある．大企業は，非公開としたいビジネス中核となるソフトウェアを除いては，ソフトウェア保守を含めて運用をアウトソーシングする．このような背景のために，開発者は通常，ソースコードを説明しなければならない場には居合わせないことが多く，変更も文書化されていないことも多い．したがって，保守者は，ソフトウェアに関して制限された理解しか得ることができず，ソースコードを自分で読まねばならない．Pigoski は，保守作業のおおよそ 40% から 60% は，改変すべきソフトウェアの理解に費やされていると指摘している [15]．したがって，保守作業の効率を高めること，さらにプログラムの理解容易性を高めるということは，ソフトウェア工学における重要な課題の 1 つとなっている．

2.1.2 類似コードが引き起こす問題

類似コードは，ソフトウェア保守を困難にする要因の 1 つして指摘されている．プログラム中に類似コードが存在すると，そのプログラムに対して修正を行うことが難しくなる．例えば，あるコードを修正する際に類似コードが存在していると，すべての類似コードに対しても修正の検討を行う必要がある．

特に大規模ソフトウェアを対象としたソフトウェア保守では，大量のソースコード中から類似コードを探し出し，その 1 つ 1 つに対して修正の検討を行うことは，大きなコストがかかる作業である．更にアウトソーシングなどにより，ソフトウェア開発者とソフトウェア保守者が異なる場合は，類似コードを探し出す作業がより大きな負担となる．このようなコストは，投資対効果の見積もりが難しいソフトウェア保守工程では，非常に大きな問題となる可能性が高い．

類似コードがソフトウェア中に作りこまれる，若しくは発生する原因として次のような場合がある [2][11][12].

既存コードのコピーとペーストによる再利用

近年のソフトウェア設計手法を利用すれば，構造化や再利用可能な設計が可能である．しかし，コードの再利用が容易になったために，現実にはコピーとペーストによる場当たり的な既存コードの再利用が多く行われるようになった．

定型処理

定義上簡単で頻繁に用いられる処理は，類似したコードが用いられている．例えば，給与税の計算や，キューの挿入処理，データ構造アクセス処理などである．

プログラミング言語に適切な機能の欠如

抽象データ型や，ローカル変数を用いられない場合には，同じようなアルゴリズムを

File: linux-2.6.19/drivers/scsi/arm/eesox.c **Code1**

```
407: if (length >= 9 && strncmp(buffer, "EESOXSCSI", 9) == 0) {
408:     buffer += 9;
409:     length -= 9;
410:
411:     if (length >= 5 && strncmp(buffer, "term=", 5) == 0) {
             ••••
418:     } else
419:         ret = -EINVAL;
420: } else
421:     ret = -EINVAL;
```

File: linux-2.6.19/drivers/scsi/cumana_2.c **Code2**

```
322: if (length >= 11 && strcmp(buffer, "CUMANASCSI2") == 0) {
323:     buffer += 11;
324:     length -= 11;
325:
326:     if (length >= 5 && strncmp(buffer, "term=", 5) == 0) {
             ••••
333:     } else
334:         ret = -EINVAL;
335: } else
336:     ret = -EINVAL;
```

図 1: 類似コードの例

持った処理を繰り返し書かなくてはならないことがある．

パフォーマンス改善

リアルタイムシステムなど時間制約のあるシステムにおいて，インライン展開などの機能が提供されていない場合に，特定のコード片を意図的に繰り返し書くことによってパフォーマンスの改善を図ることがある．

コード生成ツールの生成コード

コード生成ツールにおいて，類似した処理を目的としたコードの生成には，識別子名等の違いはあろうとも，あらかじめ決められたコードをベースにして自動的に生成されるため，類似したコードが生成される．

```
                キーワード     検索対象
% grep -n Request.type wconvert.c

759:     int cxnum = Request.type2.context;
781:     int cxnum = Request.type2.context, stat = -1;
          ••••
1022:    int mode = Request.type10.mode;
1023:    int cxnum = Request.type10.context, len, i, stat = -1;
1066:    int stat = -1, cxnum = Request.type6.context, retval;
          ••••
2319:    buf += HEADER_SIZE; Request.type2.context = S2TOS(buf);
2320:    ir_debug( Dmsg(10, "req->context =%d¥n", Request.type2.context) );
2331:    buf += HEADER_SIZE; Request.type3.context = S2TOS(buf);
2332:    buf += SIZEOFSHORT; Request.type3.buflen = S2TOS(buf);
2334:    ir_debug( Dmsg(10, "req->context =%d¥n", Request.type3.context) );
2334:    ir_debug( Dmsg(10, "req->buflen =%d¥n", Request.type3.buflen) );
          ••••
```

図 2: grep の実行例

2.2 類似コード検索

本稿では，類似コード検索を “ソースコードの集合から，クエリ（検索質問）として与えられたコード片（ソースコードの一部）と一致もしくは類似したコード片を探すこと” と定義する．なお，コード片は 5 項組 (ファイル番号, 開始行, 開始桁, 終了行, 終了桁) で定義する．

類似コードの例を図 1 に示す．図は，Linux の Version2.6.19 における類似コードである．2 つのコード片は，同じ処理内容となっているが，Code1 の 407 行目では **strncmp** が用いられているが，Code2 の 322 行目では **strcmp** が用いられている [10].

2.3 既存の類似コード検索法と問題点

2.3.1 キーワード検索に基づく手法

多くのソフトウェア開発者が利用すると考えられる類似コード検索法として，キーワード検索に基づく手法が挙げられる．キーワード検索を行う手法の例として，grep[6] が挙げられる．

grep の実行例を図 2 に示す．図では，キーワード Request.type が，ファイル wconvert.c 中に出現している箇所を検索した結果の一部を表している．出力結果は，行番号とその行のコードを出力するようにした．

次に，grep を用いた類似コード検索手順を示す．

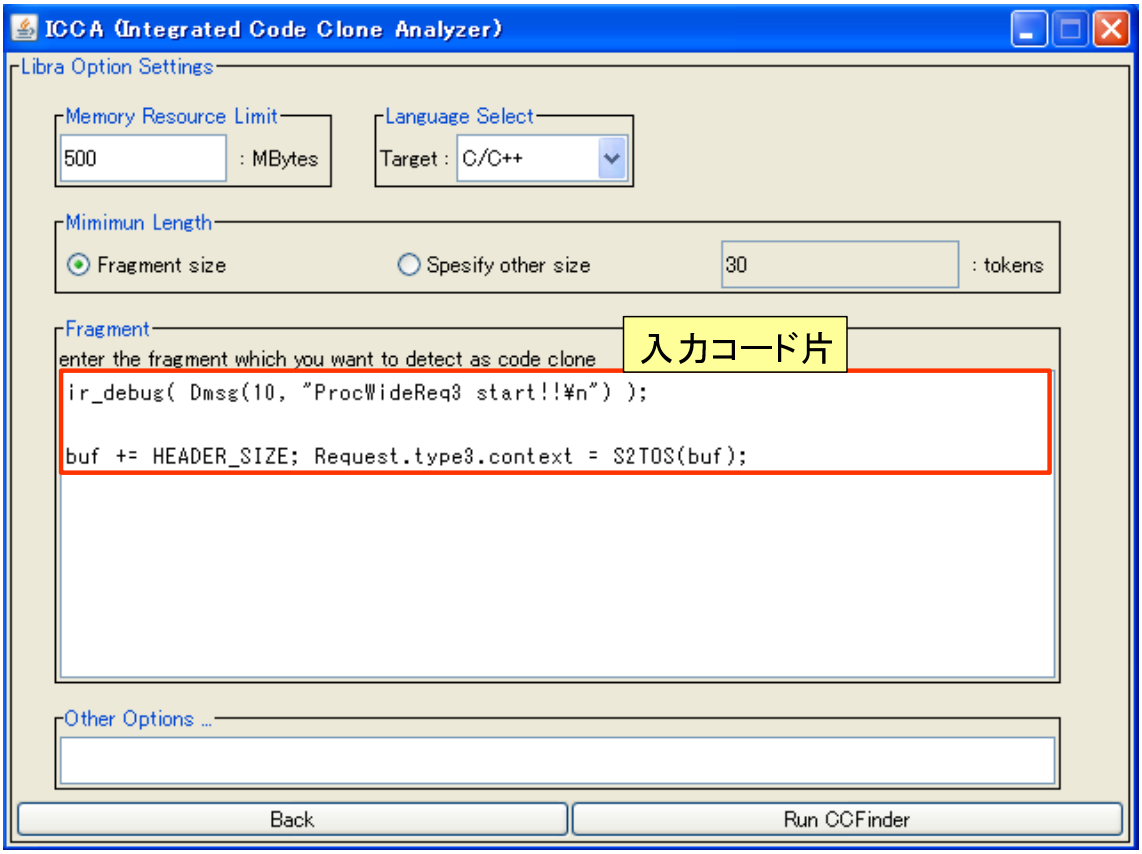


図 3: Libra の入力例

1. あるコード片から重要と思われるキーワード（識別子，式など）を発見する．
2. grep を用いてそのキーワードが出現する箇所を検索する．
3. grep の出力結果を基に，キーワードを含むコード片を特定する．

この手順にしたがう開発者は，同様の修正を検討すべきコード片間で，識別子や式などのキーワードが共起しているであろうという予想に基づいて類似コード検索を行っていると考えられる．しかし，識別子の同義語をはじめ，キーワードには様々な変化形が存在するため，単一のキーワードで同様の修正を検討すべきコードを漏れなく検索結果に含めることは困難である．

また，保守対象のソフトウェアを十分に理解している開発者でなければ，適切なキーワードを発見することは困難である．必ずしも本人が開発したソフトウェアの保守を担当するとは限らず，そのような場合には，キーワード検索を用いること自体が困難である．

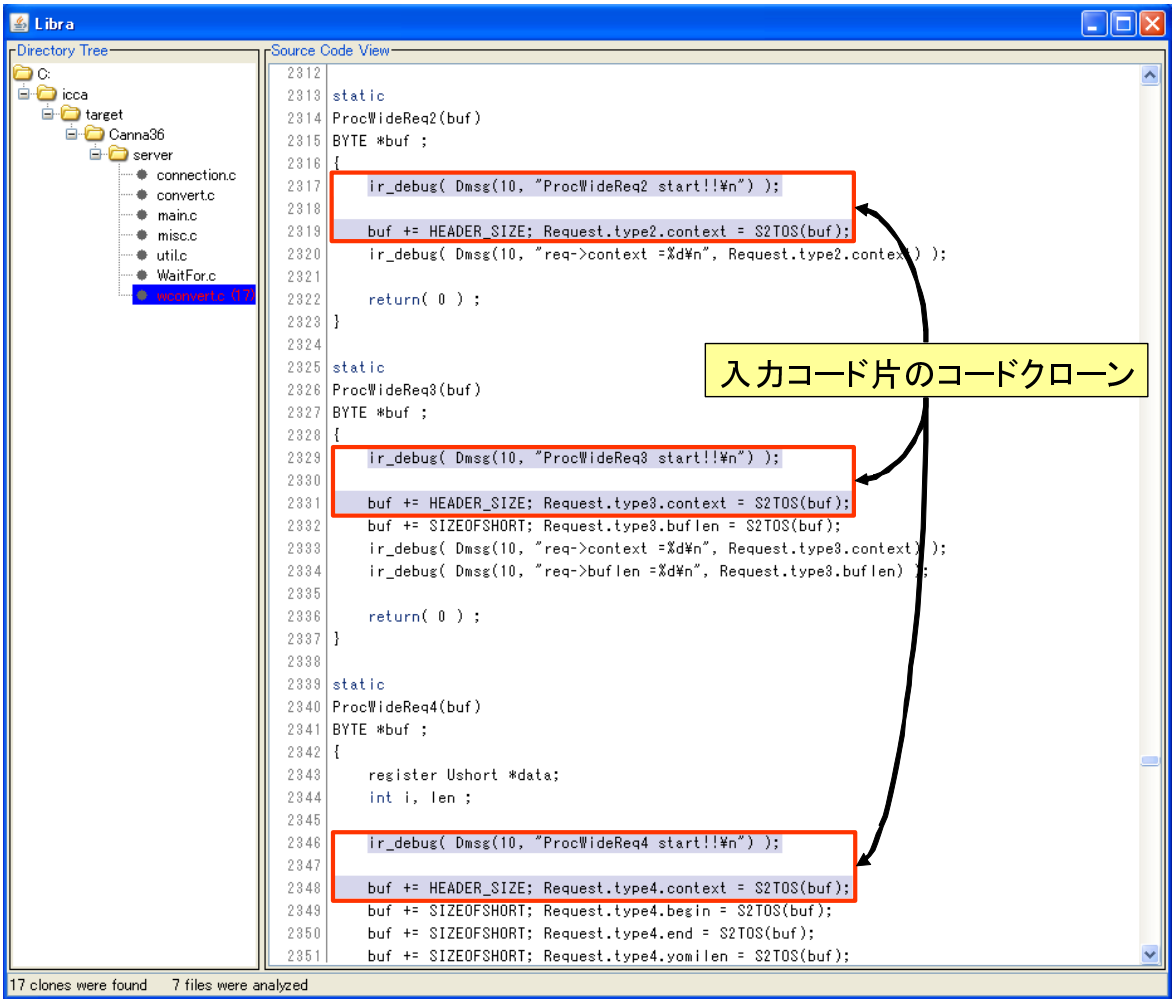


図 4: Libra の出力例

2.3.2 コードクローン検出法を用いた手法

類似コード検索法として，キーワード検索に基づく手法以外にコードクローン検出法を用いた手法が挙げられる．コードクローンとは，ソースコード中に含まれる同一，もしくは類似したコードのことであり，いわゆる “重複したコード” のことである [18].

コードクローン検出法を用いた手法の一例として，我々の研究グループで以前に提案しているコードクローン検索ツール Libra[17] について述べる．Libra は，ソースコードの集合から，クエリとして与えられたコード片のコードクローン [1][2][11][13] を検索するツールである．Libra はコードクローンの検出に CCFinder[11] を用いているため，Libra は CCFinder と同じく，連続して一致するトークン列をコードクローンとする．

そのため，クエリとして与えられたコード片と比べてトークン列上の差異があるコード片

は，検索結果に含まれない．例えば，クエリとして与えられたコード片に対してログ出力文や例外処理を追加したコード片は，検索結果に含まれない．

Libra の実行例を図 3，図 4 に示す．Libra は，検索対象のソースコード集合を指定した後で，図 3 のように入力コード片を指定することでコードクローンの検索を行う．まず，内部でコードクローン検出ツール CCFinder を起動してソースコード集合から，コードクローンの検出を行う．次に，入力コード片のコードクローンを検出されたコードクローンを対象に検索する．図 3 の実行結果は図 4 となる．入力コード片のコードクローンが，ハイライトで示されている．

Libra ではコード片をクエリとして与えられるため，開発者がキーワードやパターンを考えなくて良いという利点がある．しかし，前述のとおり，トークンが連続して一致するコード片でなければ，検索結果に含めることができないという欠点がある．また，識別子の情報（変数名など）を用いていないため，同様の修正をすべきコード片間で識別子が共起している場合は，キーワード検索に基づいた類似コード検索の方が有利である．

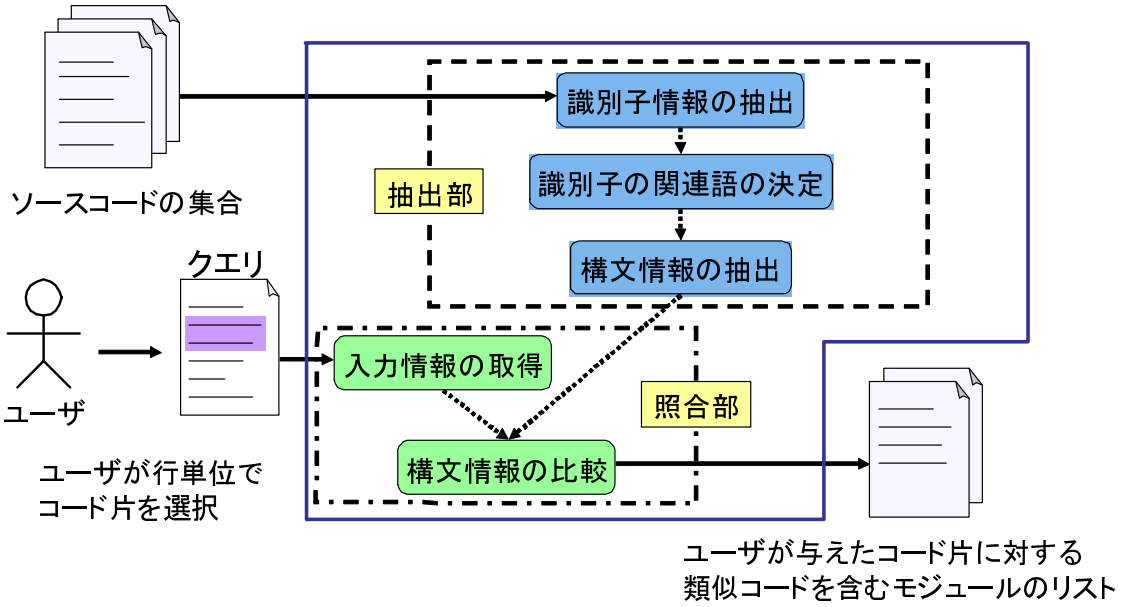


図 5: 手法の概略図

3　提案手法

3.1　提案手法の概要

本稿が提案する手法では，入力としてソースコードの集合とクエリとなる類似コード検索対象のコード片を与える．そして，入力から識別子の情報を抽出し，頻繁に出現している識別子（特徴語）を求める．次に，クエリと同じ特徴語を含むコード片を検索することで類似コードを含むモジュール（C 言語における関数，Java 言語におけるメソッドなど）を出力する．提案手法の概略図を図 5 に示す．この手法により，自動的に複数のキーワードを発見し，類似コードの検索を行うことができる．

提案手法は大きく分けてソースコード集合から識別子等の情報を抽出する抽出部と入力コード片，ソースコード集合の抽出された情報を比較する照合部の 2 つで構成されている．なお，特徴語と特徴語を含むコードの関係を図 6 に示す．

- **抽出部**

  1. ソースコードの集合から各ソースコードをモジュール単位で分割して識別子を抽出する
  2. 抽出された識別子からモジュール毎に特徴語を求め，特徴語を含むコードの集合を抽出する

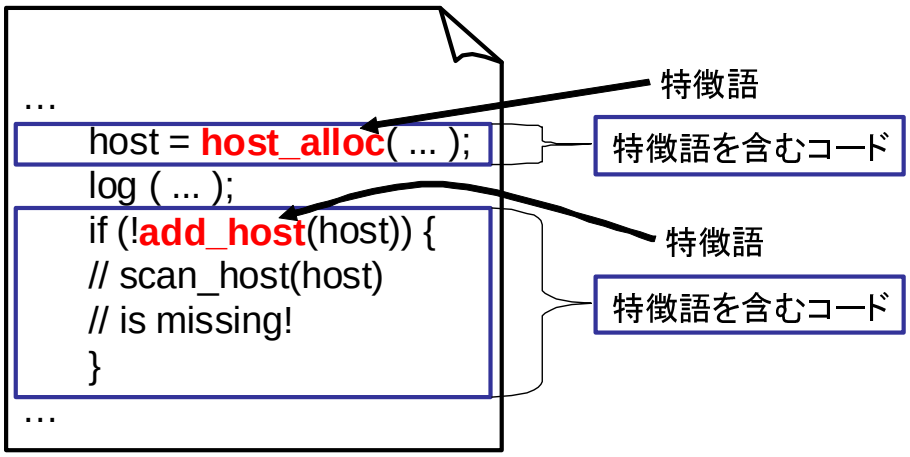


図 6: 特徴語と特徴語を含むコードの関係

- **照合部**

  1. クエリとしてコード片を与え，特徴語，及び特徴語を含むコードを抽出する
  2. 特徴語と特徴語を含むコードが表す構文の組が，クエリと同じモジュールを検出する

3.2 準備

3.2.1 提案手法が検出する類似コードの定義

提案手法では，特徴語を含むコード集合同士を比較し，以下の項目が成り立つ場合，比較したコード集合を類似コードと定義する．

- 出現する特徴語が一致，もしくは関連語の関係にある
- 構文に関する情報が一致するコード片が存在する

3.2.2 特徴語の定義

コード片で頻出する識別子を特徴語と定義する．具体的には，あるモジュールにおいて，設定した閾値を越える出現回数を持つ識別子をそのモジュールの特徴語と定義する．閾値は，まずソースコード集合から固有の値として基準値を求め，モジュール毎に正規化を行うことによって与えられる．基準値，閾値の定義は次のように設定した．

$$\text{基準値} = \frac{\text{ソースコード集合で識別子が出現した回数の総和}}{\text{各モジュールで出現した識別子の種類の総和}} \tag{1}$$

$$\text{平均出現回数} = \frac{\text{ソースコード集合で識別子が出現した回数の総和}}{\text{ソースコード集合中に存在するモジュールの数}} \tag{2}$$

$$\text{閾値} = \text{基準値} \times \frac{\text{対象モジュールで識別子が出現した回数の総和}}{\text{平均出現回数}} \tag{3}$$

3.2.3 関連語の定義

共起回数の分布が類似している識別子の集合を関連語と定義する．提案手法では，共起回数を識別子が同時に出現しているモジュールの数とした．また，共起回数の分布が類似していることを表す尺度として，自然言語処理において文書中の語の類似性を計測するために用いられている Jensen-Shannon divergence[3][23] を採用した．

ある分布を基準として別の分布との違いを計測する尺度として Kullback-Leibler divergence がある [3][14]．Jensen-Shannon divergence とは，非対称である Kullback-Leibler divergence を対称にした値である．提案手法では，この値を識別子間の距離として用いている．つまり，共起回数の分布が類似していると，識別子間の Jensen-Shannon divergence は小さくなる．

3.2.4 構文情報の定義

対象のコード片において，出現している特徴語の集合と文の種類の組を構文情報として定義する．文の種類は以下のように分類した．

- 変数宣言などを表す宣言文
- 条件文，繰り返し文などの条件節
- break 文，return 文を表す補助制御文
- 上記の 3 つのどれにも当てはまらない文

3.3 類似コード検出手順

3.3.1 抽出部

この節では，類似コード検出の前段階として，識別子から情報の抽出を行う抽出部の流れについて説明する．

手順 1　識別子情報の抽出

まず，ソースコードの集合からすべてのモジュールを抽出する．次に，モジュール中に存在する識別子を抽出し，各識別子の出現回数を計算する．このとき予約語，型名は除外している．また，抽出した識別子が複数の語から成り立つ場合は，分解し個々の語を識別子として扱う．

手順 2　識別子の関連語の決定

まず，ソースコード集合中に出現する識別子に対して，識別子間の距離を求める．次に，クラスタリングを行うことで共起回数の分布が類似した識別子をグループ化する．それぞれのグループを関連語集合とする．

クラスタリングには，全ての識別子がそれぞれ 1 つのクラスタという状態から始めて，クラスタ間の距離が最小となるクラスタの組から順次結合していく形式を採用した．なお，クラスタ間距離については，2 つのクラスタに属する要素間の距離の平均をクラスタ間距離とする群平均法（平均距離法）[19, p. 50][21, p. 136] を用いた．アルゴリズムは以下で表される．C はクラスタの集合，$distance(c_\alpha, c_\beta)$ はクラスタ間距離を表している．

手順 2.1

初期状態 $C = \{c_\alpha \mid 1 \leq \alpha \leq |I|\}, c_\alpha = \{i_\alpha\}$

手順 2.2

$\forall\alpha\forall\beta\ distance(c_p, c_q) \leq distance(c_\alpha, c_\beta)$ となる p，q を探索

手順 2.3

$c_p = c_p \cup c_q$，C から c_q を取り除く

手順 2.4

これを $|C| = 1$ となる，もしくは終了条件 $\forall\alpha\forall\beta\ s \geq distance(c_\alpha, c_\beta)$，($s$ は与えられた閾値) を満たすまで繰り返す．

手順 3　構文情報の抽出

モジュール毎に特徴語を求め，特徴語を含むコードを特定する．それぞれの特徴語を含むコードに対して，構文情報を抽出する．その結果，モジュール毎に構文情報の集合が生成される．

3.3.2　照合部

この節では，抽出部で得られた情報を基に類似コード検出を行う照合部の流れについて説明する．

手順 A　入力情報の取得

クエリとして与えられたコード片から特徴語を含むコードを特定し，構文情報を抽出する．特徴語を含むコードの特定は，クエリを 1 つのモジュールと見なして抽出部で述べた方法で行う．抽出された構文情報の集合を，類似コード検索の入力として用いる．

手順 B　構文情報の比較

クエリの構文情報が各モジュール構文情報と対応するか比較を行う．構文情報の比較の条件については次のように設定した．

手順 B.1

クエリ側の構文情報を { 特徴語の集合 A, 文の種類 A}，比較対象の構文情報を { 特徴語の集合 B, 文の種類 B} とする

手順 B.2

特徴語の判定を行う

手順 B.2.1

特徴語の集合 A と 特徴語の集合 B について，特徴語をその特徴語が属する関連語集合に変換する

手順 B.2.2

特徴語の集合 A 中の関連語集合すべてが，特徴語の集合 B に存在する場合，特徴語の集合 A は 特徴語の集合 B と対応した判定する

手順 B.3

特徴語が対応し，文の種類が一致した場合，クエリ側の構文情報が比較対象の構文情報と対応したと判定する

クエリの構文情報すべてが，モジュール中のいずれかの構文情報と対応する場合，そのモジュールをクエリの類似コードを持つモジュールとする．比較の計算時間コストを削減するため，特徴語が，クエリの特徴語，または関連語となっているモジュールのみを比較対象とした．

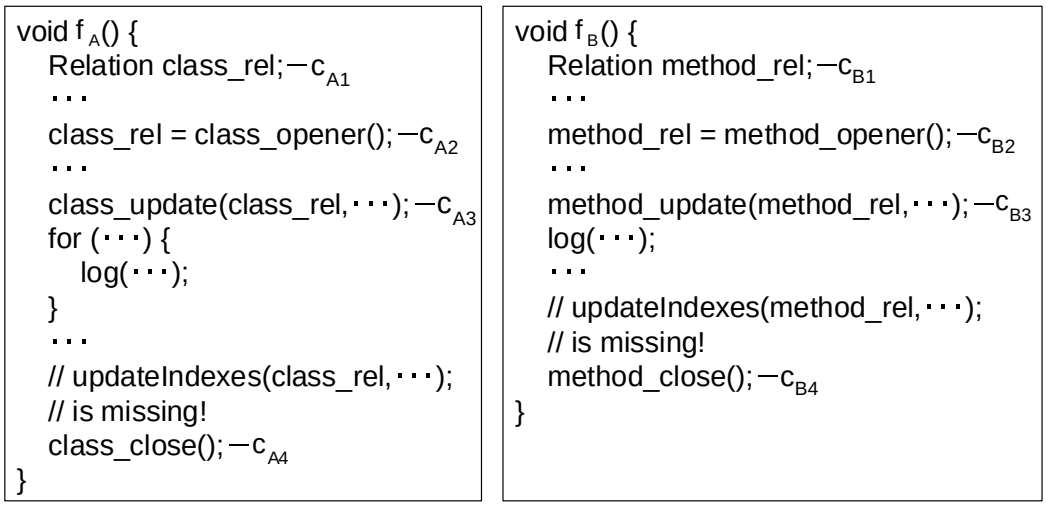


図 7: 入力コード片と比較対象のモジュール

$$
\begin{array}{c} \\ f_A \\ f_B \\ \\ \end{array}
\begin{array}{c}
\begin{array}{ccccccccc} \text{class} & \text{rel} & \text{update} & \text{log} & \text{opener} & \text{close} & \text{method} & & \end{array} \\
\begin{pmatrix}
\cdots & 6 & 3 & 1 & 1 & 1 & 1 & 0 & \cdots \\
\cdots & 0 & 3 & 1 & 1 & 1 & 1 & 6 & \cdots \\
 & & & & & & & & \ddots
\end{pmatrix}
\end{array}
$$

図 8: 識別子の出現回数を表す行列

3.4 例題

提案手法の流れを例題を用いて説明する．図 7 から図 12 は，入力コード片である関数 f_A と比較対象である関数 f_B について，構文情報を比較する流れを示している．

図 7 では，関数 f_A，関数 f_B のコードの一部を表している．これらの関数では，関数呼び出し opener，関数呼び出し update，関数呼び出し close を順に呼び出しているが，関数呼び出し update の後に関数呼び出し updateIndexes が欠落しているという欠陥が存在している．

関連語を求めるまでの流れは，次のようになる．

手順 A.1　識別子の出現回数の算出

関数ごとに識別子が何回出現しているかを計算し，行列を作成する（図 8）．

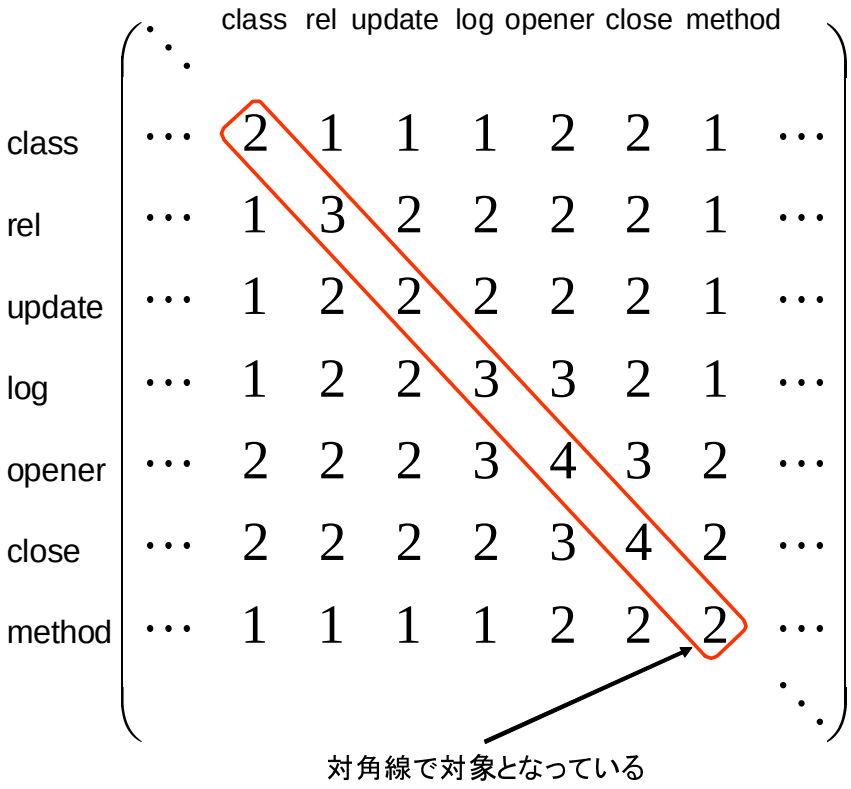


図 9: 識別子の共起回数を表す行列

手順 A.2　識別子間の共起回数の算出

識別子の共起回数を求め，行列の各要素が識別子間の共起回数を行列を作成する（図 9）．なお，行と列が表す識別子が同じ要素の場合は，その識別子が出現している関数の数となる．共起行列は，対角線で対象となっている．

手順 A.3　識別子間の距離の算出

共起行列に基づいて，Jensen-Shannon divergence を求める．その結果から識別子間の距離を表す行列を作成する（図 10）．この行列は，対角線で対象となっている．

手順 A.4　関連語のクラスタリング

クラスタリングの閾値に基づいて，識別子間の距離が小さい識別子をクラスタリングする．図 11 は，クラスタリングの閾値を 0.00 とした場合のクラスタリング結果である．この場合では，識別子 class と識別子 method が関連語となっている．

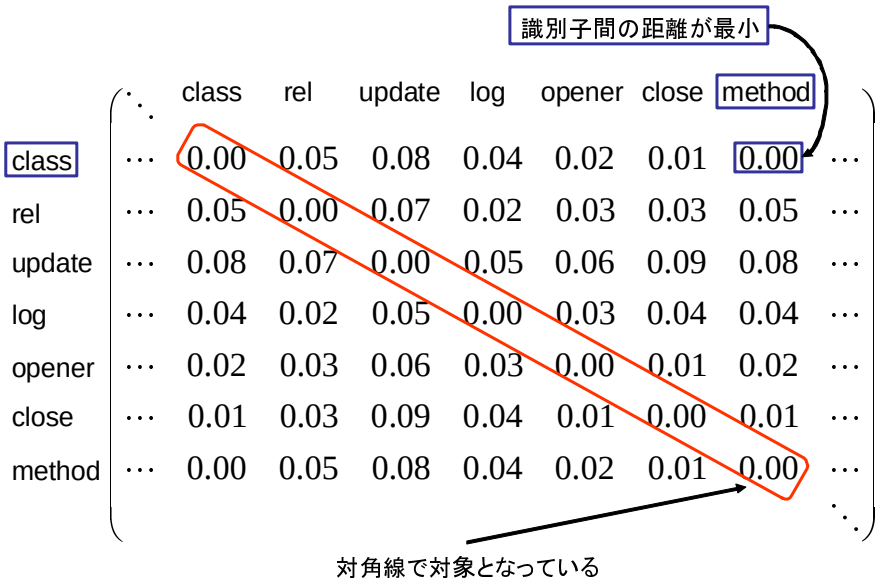


図 10: 識別子間距離を表す行列

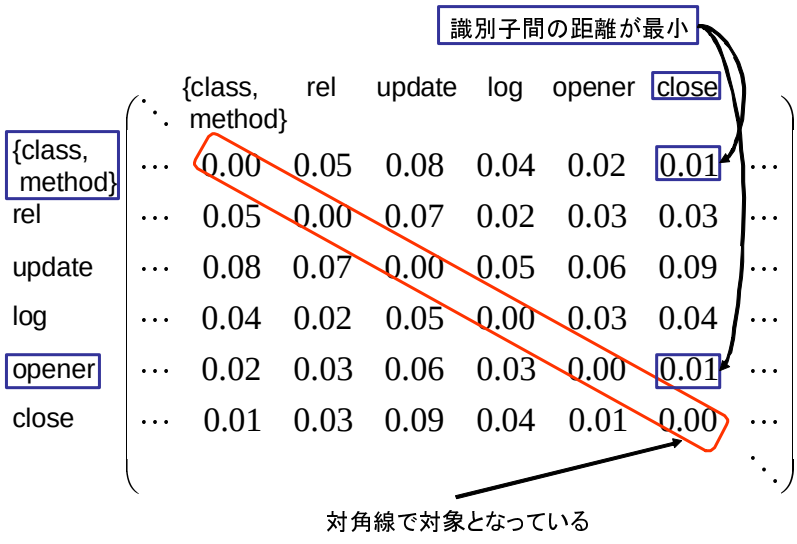


図 11: クラスタリング後の行列

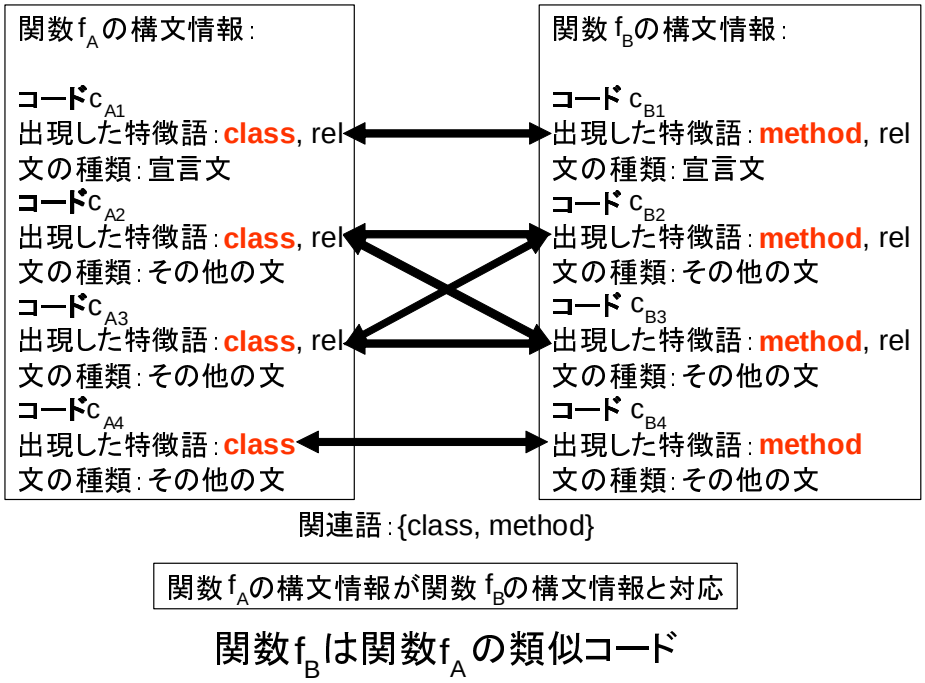


図 12: 構文情報の比較

図 12 では，構文情報を比較する流れを表している．具体的な手順は，次のようになる．

手順 B.1　特徴語の抽出

例題では，関数 f_A の特徴語が識別子 class と識別子 rel，関数 f_B の特徴語が識別子 method と識別子 rel であったとしている．

手順 B.2　特徴語の判定

識別子 class と識別子 method は関連語となっており，また残りの特徴語 rel は共通であることから，関数 f_A の特徴語全てを関数 f_B が含んでいることになる．従って，関数 f_A と関数 f_B の構文情報の比較を行う．

手順 B.3　構文情報の抽出

関数 f_A の特徴語を含むコードは，コード c_{A1}，コード c_{A2}，コード c_{A3}，コード c_{A4} となる．同様に，関数 f_B の特徴語を含むコードはコード c_{B1}，コード c_{B2}，コード c_{B3}，コード c_{B4} となる（図 7）．それぞれのコード毎の構文情報は，図 12 となる．

手順 B.4　構文情報の比較

関数 f_A の各コードにおいて，構文情報が対応する関数 f_B のコードを検出する．例題の場合では，コード c_{A1} がコード c_{B1} に対応し，コード c_{A2} はコード c_{B2} 及びコード

c_{B3} に対応している．同様にコード c_{A3} もコード c_{B2} 及びコード c_{B3} に対応している．そして，コード c_{A4} はコード c_{B4} と対応している．その結果，入力コード片である関数 f_A の構文情報全てが対応する組を持っているので，関数 f_B は関数 f_A の類似コードと判定される．

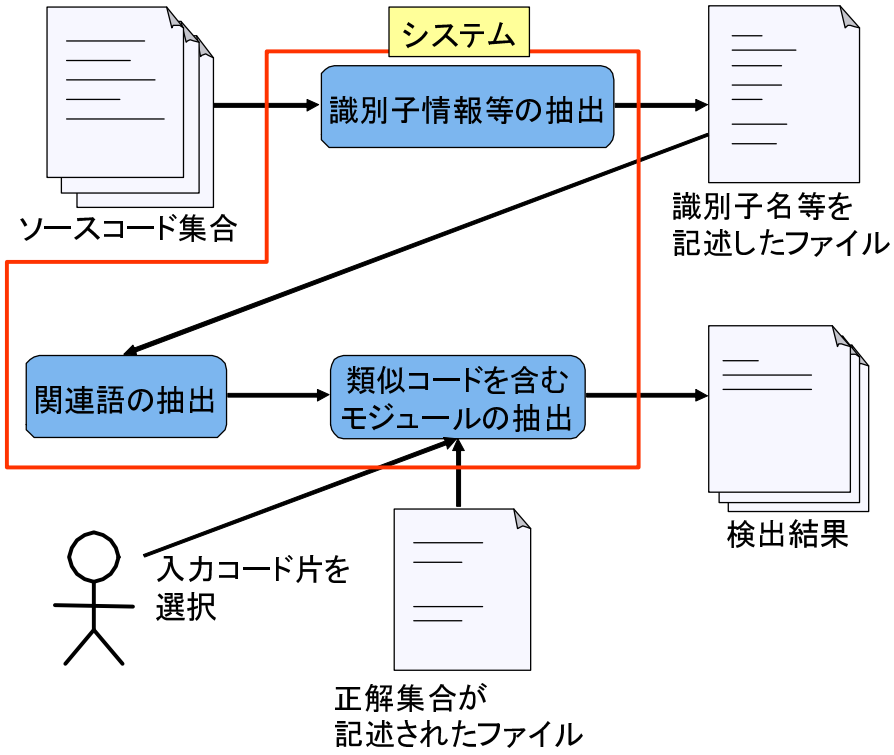


図 13: 類似コード検索システム SCRetriever の概略図

4　類似コード検索システム：SCRetriever

4.1　システムの概要

提案手法の有効性を評価するために作成した類似コード検索システムについて述べる．本システムは，Java 言語で記述されており，C 言語と Java 言語のソフトウェアを対象とする．図 13 に作成したシステムの概略を示す．

本システムでは，対象のソースコード集合から特徴語，関連語を求めるために必要な情報を一旦ファイルに出力した後，そのファイルを用いて類似コード検索を行う．以降，それぞれの処理について説明する．

4.2　識別子情報等の抽出

対象のソースコード集合から，類似コード検索に必要な情報の抽出を行う．抽出された情報は，以降の処理で用いるためファイルに出力される．抽出する内容は，次の 5 つの項目となっている．

- 記述されているプログラミング言語
- ソースコード集合中に存在するモジュールのリスト
- ソースコード集合中に存在する識別子のリスト
- 各モジュール中に出現する識別子毎の出現回数
- 各識別子における識別子間の距離

なお，モジュールや識別子のリストを取得するために，各プログラミング言語毎のパーサを実装した．パーサの実装には，JavaCC[9] を用いた．JavaCC とは，オープンソースの Java 言語向けパーサ生成ツールである．字句規則と構文規則を記述することによって，Java 言語で記述されたパーサのソースコードを自動的に生成する．また，C 言語のパーサについては，実装の都合上，プリプロセッサ処理を行った後のソースコードを対象として実装した．

4.3 関連語の抽出

識別子情報等が記述されたファイルを基に，関連語集合を生成する．なお，クラスタリングを行う際に，クラスタリングの閾値を設定する割合を設けている．具体的には，初期状態のクラスタ間距離の最大値，つまり識別子間の距離が最大となる値に対する割合としている．例えば，設定した割合が 0% の場合では，共起回数が完全に等しい識別子のみが関連語集合として生成される．設定した割合が大きくなるについて，関連の弱い語も関連語集合に含まれるようになる．

4.4 類似コードを含むモジュールの抽出

クエリを選択し，類似コードを含むモジュールの検出を行う．この処理を行っている画面は，図 14 である．本システムでは，クエリをモジュールから選択するようになっている．そのため，入力コード片を記述したモジュールを作成してクエリとすることで，擬似的に入力コード片をシステムに与えるようにしている．なお，入力コード片を記述したモジュールは，本来のソースコード集合には存在しないモジュールであるため，検出結果から除外する．

また，適用実験の際に検出されるべきモジュールを指定する必要があるため，そのモジュール名を記述したファイルを与えることで，指定するようにしている．なお，検出結果は csv ファイルで出力される．出力される内容は，次の 3 つである．

- 各入力コード片毎の適合率
- 各入力コード片毎の再現率

- 各入力コード片毎の検出モジュール数と検出できた正解モジュール数

適合率とは，検出ノイズの少なさを表す尺度であり，再現率とは，検出漏れの少なさを表す尺度である [20, pp. 17–19][22, pp. 73–77]．適合率，再現率の詳細については，5.1.3 節で述べる．

図 15 は，出力ファイルの一例である．設定した割合における入力コード片毎の検出結果を，並べて出力するようになっている．

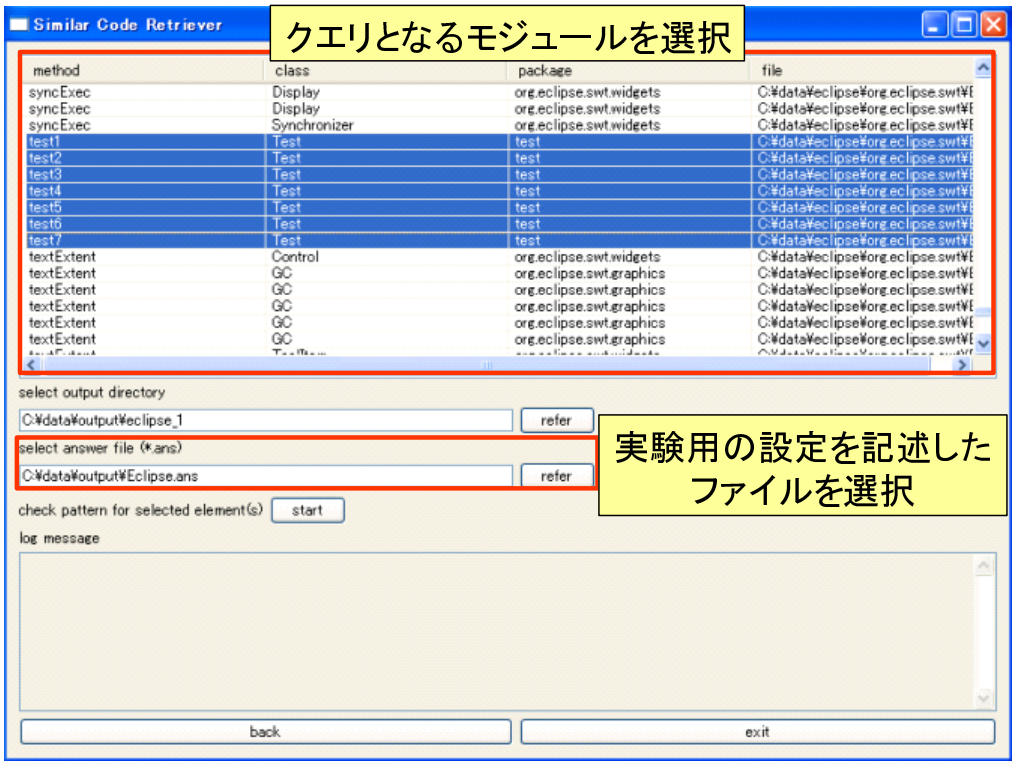


図 14: 類似コード検索画面

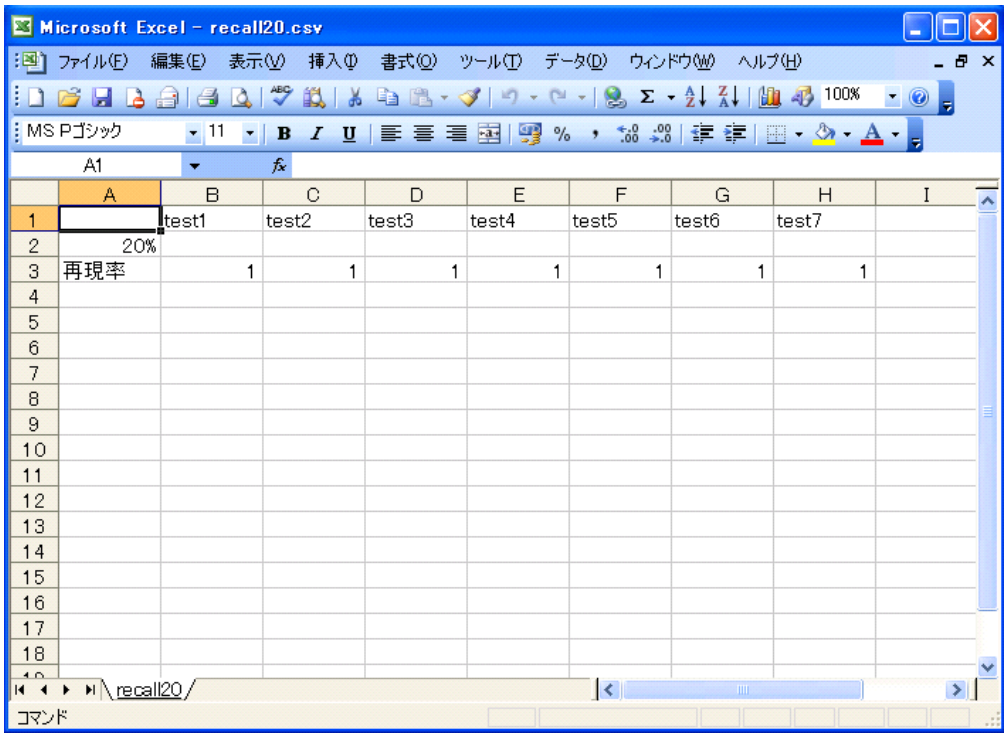


図 15: 出力ファイル例

5　欠陥検出を目的とした実験

5.1　実験概要

5.1.1　実験対象

提案手法の評価を行うため，4 節で述べたシステムを C 言語のソフトウェアについて適用し，実験を行った．実験対象は，オープンソースソフトウェアの日本語入力システム “かんな”[25] と我々の研究グループで開発しているソフトウェア部品検索システム SPARS-J[24] を用いた．なお，実験対象が C 言語のソフトウェアであるため，検出を行う単位は関数単位とした．実験対象のデータは表 1 となっている．欠陥の数とは，修正が行われた箇所の数を表している．以降，それぞれのソフトウェアについて行った実験について説明する．

かんなはクライアント・サーバ方式の日本語入力システムであり，かな漢字変換をサーバと接続して実行している．かんなでは，バージョン 3.6 とバージョン 3.6p1 間でバッファのオーバーフローを検知するコードが追加された [17]．修正された箇所の例は図 16 である．バッファのオーバーフローを検知するコードが追加された箇所は，サーバの処理を行っているコードであった．サーバ関連の欠陥に対する修正と考え，サーバ関連の処理を行っている server ディレクトリ以下のファイルを実験対象とした．図 16 で示した修正は 19 個の関数で行われており，これらの関数を正解集合とする．入力コード片は，修正前バージョンから修正が行われる箇所と前後の行，合計 3 行とした．修正は 21 箇所存在していたので，21 個の入力コード片に対して実験を行った．

SPARS-J では，複数の関数において型キャストの追加が同時に行われる修正があった．修正された箇所の例は図 17 である．修正が行われた関数は様々なディレクトリに存在していたため，ソフトウェア全体を実験対象とした．実験に用いたソースコード集合は，修正が行われる前の最新の状態を用いた．図 17 で示した修正は 52 個の関数で行われており，これらの関数を正解集合とする．かんなの場合と同様に，入力コード片は，修正前のソースコードから修正が行われる箇所と前後の行，合計 3 行とした．修正は 75 箇所存在していたので，75 個の入力コード片に対して実験を行った．

表 1: 実験対象のデータ

ソフトウェア	総行数	ファイル数	欠陥の数	欠陥を含む関数の数
かんな（server ディレクトリ以下）	7,613 行	7 ファイル	21 個	19 個
SPARS-J	35,744 行	171 ファイル	75 個	52 個

```
static
ProcWideReq2(buf)
BYTE *buf ;
{
    ir_debug( Dmsg(10, "ProcWideReq2 start!!¥n") );

    if (Request.type2.datalen != SIZEOFSHORT)
                return( -1 );

    buf += HEADER_SIZE; Request.type2.context = S2TOS(buf);
    ir_debug( Dmsg(10, "req->context =%d¥n", Request.type2.context) );

    return( 0 ) ;
}
```

追加されたコード

(a)

```
ir_debug( Dmsg(10, "req->context =%d¥n", Request.type10.context) );
ir_debug( Dmsg(10, "req->number =%d¥n", Request.type10.number) );
ir_debug( Dmsg(10, "req->mode =%d¥n", Request.type10.mode) );

if (rest != Request.type10.number * SIZEOFSHORT)
    return( -1 );

buf += SIZEOFINT; Request.type10.kouho = (short *)buf;
for (i = 0; i < Request.type10.number; i++) {
    Request.type10.kouho[i] = S2TOS(buf); buf += SIZEOFSHORT;
    ir_debug(Dmsg(10, "req->kouho =%d¥n", Request.type10.kouho[i]));
}
•••
```

追加されたコード

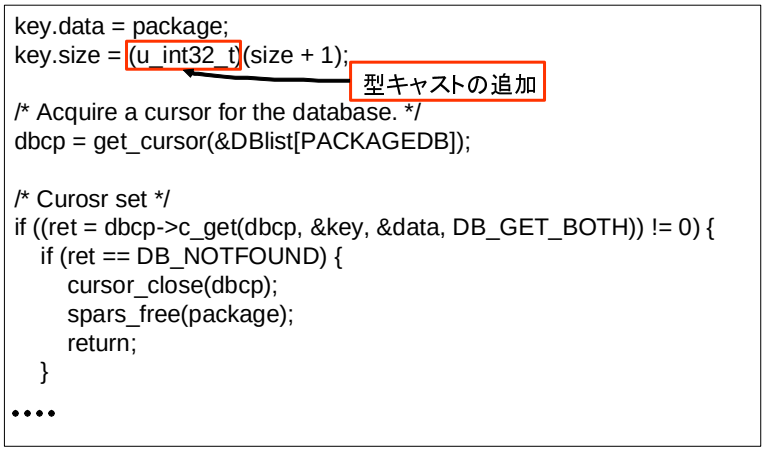
(b)

```
buf += SIZEOFSHORT; Request.type13.kouhosize = S2TOS(buf);
buf += SIZEOFSHORT; Request.type13.hinshisize = S2TOS(buf);

if (Request.type13.yomilen != len - 1)
    return( -1 );

ir_debug( Dmsg(10, "req->context =%d¥n", Request.type13.context) );
ir_debug( Dmsg(10, "req->dicname =%s¥n", Request.type13.dicname) );
•••
```

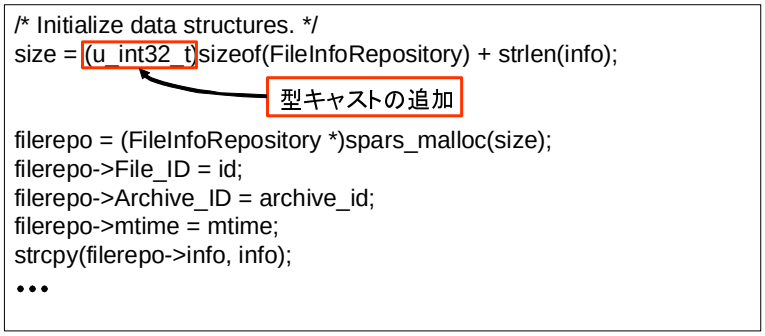
追加されたコード

(c)

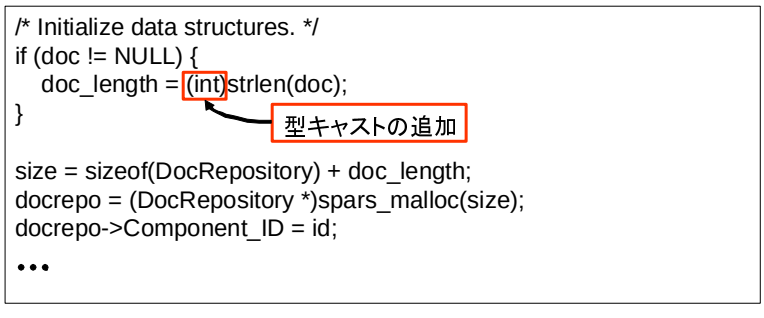
図 16: かんなの修正事例



(a)



(b)



(c)

図 17: SPARS-J の修正事例

5.1.2 実験で確認する項目

実験で次の 4 つの項目を確認する．

クラスタリングの閾値が与える影響

関連語の範囲を広げることによって，正解をどの程度検出できるようになるか調べる．今回は，クラスタリングの閾値として設定する割合を 0% から 100% まで 10% 刻みで変化させて実験を行った．

特徴語の閾値が与える影響

特徴語の閾値を変更することで，検出できる正解の数を調べる．今回は，3.2.2 節で述べた閾値に基づいて特徴語を定義する場合とモジュール中に出現する識別子を全て特徴語と定義する場合の 2 つについて実験を行った．なお，以降では，WOF(Word Occurrence Frequency) フィルタと表記する．

構文情報が与える影響

構文情報の使用の有無によって，検出する関数の数がどの程度変化するか調べる．実験では，クエリの特徴語が属する関連語集合を全て含むモジュールの数とそのモジュールに対して構文情報を用いた比較を行った結果について調べた．なお，以降では，SI(Syntax Information) フィルタと表記する．

既存ツールとの比較

提案手法の比較実験として，Libra でも同様の実験を行う．ソースコード集合は，本システムと同様にプリプロセッサ処理を行った後のソースコードを用いた．

5.1.3 評価基準

実験結果の評価は，情報検索の分野でよく用いられている適合率，再現率 [20, pp. 17–19][22, pp. 73–77] で評価する．適合率とは，検索された集合の中で適合するものの割合であり，検索ノイズの少なさを示す尺度である．再現率とは，検索対象中の適合する集合の中で実際に検索されたものの割合であり，検索漏れの少なさを示す尺度である．適合率，再現率はトレードオフの関係となっている．提案手法の実験におけるそれぞれの定義は次のように定義した．

$$\text{適合率} = \frac{\text{検出された正解集合の数}}{\text{検出された関数の数}} \times 100(\%) \tag{4}$$

$$\text{再現率} = \frac{\text{検出された正解集合の数}}{\text{正解集合の総数}} \times 100(\%) \tag{5}$$

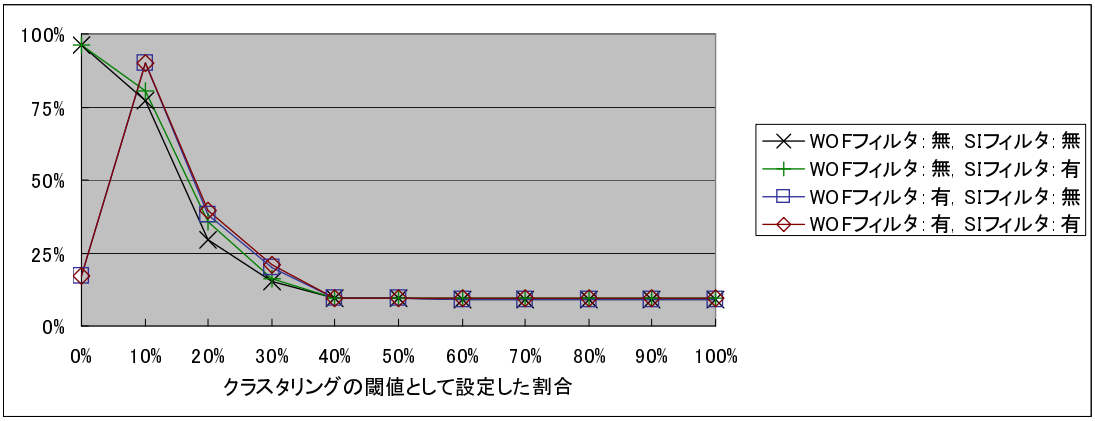


図 18: かんなの server ディレクトリ以下における適合率

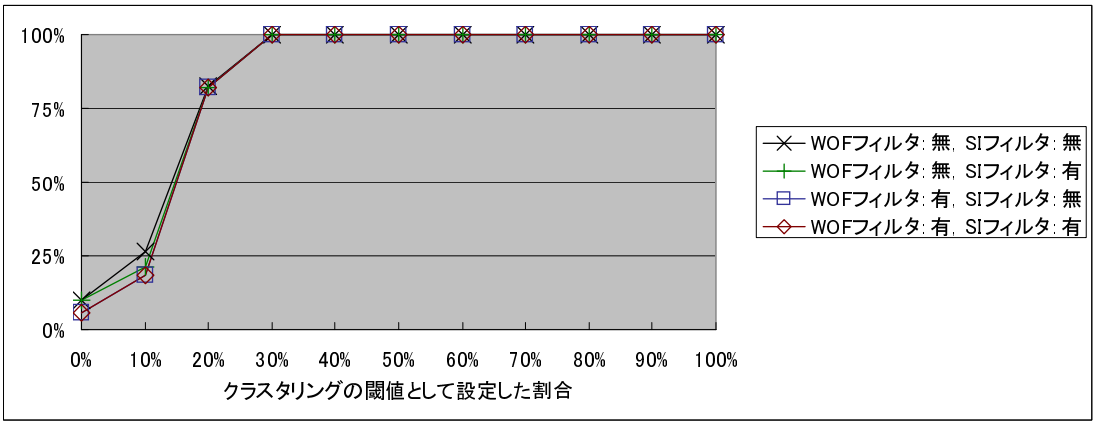


図 19: かんなの server ディレクトリ以下における再現率

また，Libra で行った実験の適合率，再現率は次のように定義した．

$$\text{適合率} = \frac{\text{修正箇所がコードクローンとして検出された正解集合の数}}{\text{検出されたコードクローンの数}} \times 100(\%) \quad (6)$$

$$\text{再現率} = \frac{\text{修正箇所がコードクローンとして検出された正解集合の数}}{\text{正解集合の総数}} \times 100(\%) \quad (7)$$

5.2 提案手法の実験結果

かんな，SPARS-J を対象とした提案手法の実験結果について述べる．なお，実験結果を表すグラフの値は，平均値を表している．

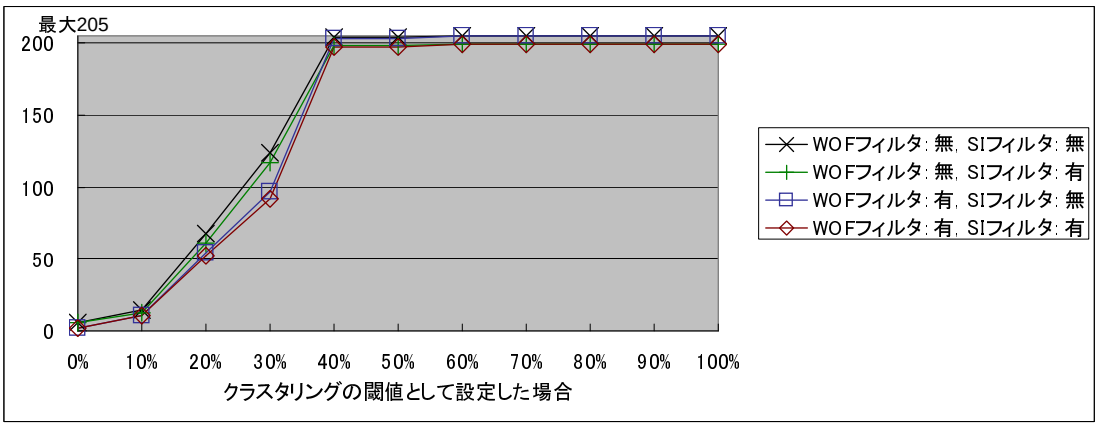


図 20: かんなの server ディレクトリ以下における検出数

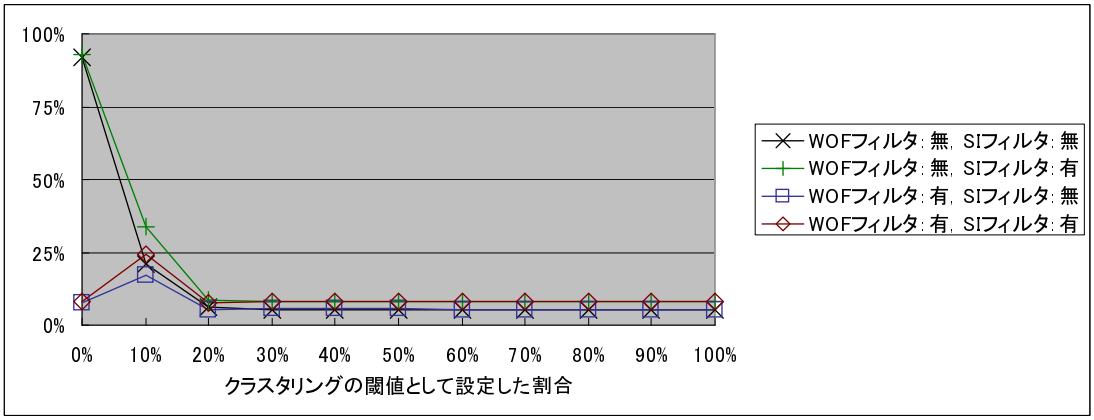


図 21: SPARS-J における適合率

5.2.1 かんなの実験結果

図 18 は，かんなの server ディレクトリ以下を対象とした適合率の結果である．設定した割合が 10% 以降では，WOF フィルタが有効な場合の方が若干高い値となった．特徴語を制限することによって，不正解のモジュールを除外できているためである．また，設定した割合が 10% 未満では，WOF フィルタが有効であると適合率が低い値となっている．関連語の範囲が狭いため，入力コード片の特徴語の集合と対応する特徴語の集合を持つモジュールが発見できないためである．なお，SI フィルタについては，有効であると若干高い値となる場合が存在するが，あまり差異は見られない結果となった．

図 19 は，かんなの server ディレクトリ以下を対象とした再現率の結果である．WOF フィルタが無効な場合の方が，若干高い値となっている．特徴語を制限することによって，正解

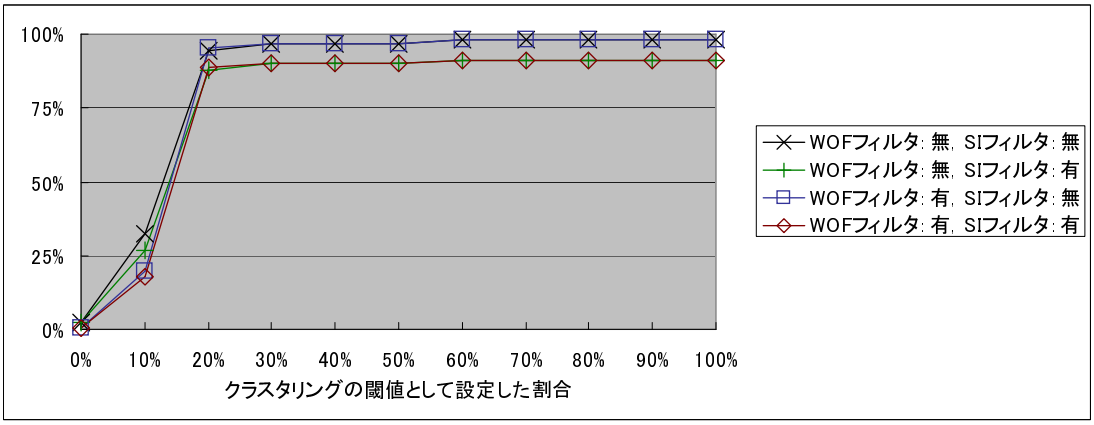


図 22: SPARS-J における再現率

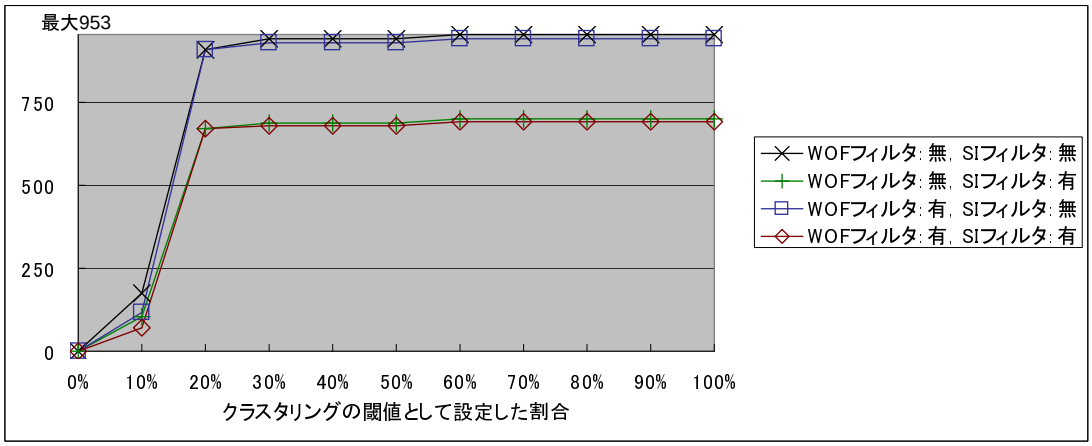


図 23: SPARS-J における検出数

の関数が誤って除外されているためである．しかし，設定した割合が 20% 以降では，差が見られない結果となっている．なお，SI フィルタについては，WOF フィルタが無効な場合に影響が見られるが，WOF フィルタが有効な場合では，有無に関わらず同じ値となっている．

図 20 は，かんなの server ディレクトリ以下を対象とした検出数の結果である．ソースコード集合中に存在する関数の総数は 205 個となっている．WOF フィルタ，SI フィルタ共に，有効な方が検出数が少なくなっている．特に，設定した割合が 20% から 30% の間でフィルタの効果が大きい結果となっている．

5.2.2 SPARS-J の実験結果

図 21 は，SPARS-J を対象とした適合率の結果である．設定した割合が 10% 未満では，かんなの場合と同様に WOF フィルタが有効であると低い値となっている．それ以降については，WOF フィルタの有無による影響は小さくなっている．関連語の範囲が広がることで，特徴語が同じ関連語の集合に属するようになるためである．なお，SI フィルタについては，有効な方が設定した割合 10% で高い値となっている．しかし，その他の割合では，フィルタの有無に関わらず同じ値となっている．

図 22 は，SPARS-J を対象とした再現率の結果である．SI フィルタが有効な場合の方が，設定した割合に関わらず低くなっている．構文情報を用いることによって，正解の関数が誤って除外されているためである．なお，WOF フィルタについては，設定した割合が 30% 以降で，有無に関わらず同じ値となっている．

図 23 は，SPARS-J を対象とした検出数の結果である．ソースコード集合中に存在する関数の総数は，953 個となっている．設定した割合が 20% 以降，SI フィルタが有効になると大きく検出数が減少している．関連語の範囲が広いので，特徴語の集合が対応する関数の数が多いためである．なお，WOF フィルタについては，有無に関わらず同じ値となっている．

5.3 Libra の実験結果

表 2 は，かんな，SPARS-J における Libra の検出結果である．Libra では，構文解析に失敗した場合，検出されたコードクローンの数が 0 として出力される．この場合，検出に失敗したとして適合率は 0% に設定している．比較実験では，SPARS-J において，5 つの入力コード片がコードクローン検出に失敗していた．Libra の結果は，本システムに比べて，適合率の平均はとても高い結果となったが，再現率の平均は低い結果となった．特に，SPARS-J での再現率の平均は，かなり低い値となっている．

表 2: Libra の実験結果

ソフトウェア	適合率の平均	再現率の平均	検出数の平均
かんな (server ディレクトリ以下)	100.00%	53.38%	10.14
SPARS-J	86.01%	3.31%	30.21

6　考察

6.1　実験の結果に関する考察

提案手法を実装したシステムの実験結果についての考察を述べる．まず，かんなの実験結果についての考察を述べる．WOF フィルタについては，再現率が大きく変化するところで，フィルタによって再現率があまり減少することなく，検出数が減少していた．不正解となる関数のみを除外できていることから，フィルタとして機能していると考えられる．また，設定した割合が 10% 未満では，適合率に大きな差ができている．関連語の範囲が狭いため，特徴語が一致しなければ検出が困難であると考えられる．一方，SI フィルタについては，適合率，再現率にあまり変化は見られなかった．入力コード片中に出現する特徴語の集合のみでも問題なく検出が行えることになる．このことから，特徴語とその特徴語が出現する構文の結びつきが強いと考えられる．つまり，特徴語が文の特徴を表す語として機能していることが言える．

次に，SPARS-J の実験結果についての考察を述べる．WOF フィルタについては，かんなの場合と同様に設定した割合が 10% 未満では，適合率に大きな差ができている．入力コード片のサイズが小さいため，元々のソースコードでは，出現回数が小さい識別子が特徴語となっていることが考えられる．一方，SI フィルタについては，フィルタを有効にすることで検出数は大きく減少したが，再現率も減少していた．適合率にあまり変化が見られないことから，不正解のフィルタとしては機能していないと考えられる．このことから，類似コードの特徴語が出現する構文がまちまちであると考えられる．そのため，SI フィルタによって検出漏れが発生していると言える．

次に，本システムの実験結果についての考察を述べる．かんな，SPARS-J 共にクラスタリングの閾値がある閾値以降高い再現率となっていた．手法が利用される状況を考えると，欠陥を漏れなく検出することが要求されるため，高い再現率は重要であり，意義ある結果と言える．また，設定した割合が 10% から 20% の間で再現率が大きく変化していることから，検査を行う際に目安となる閾値を設定できると考えられる．また，クラスタリングの閾値が高い場合，適合率，再現率にあまり変化が見られなかった．検査する閾値の範囲を絞り込むことで，検査の効率を上げることが期待できる．

フィルタリングについては，SI フィルタは，実験対象によって結果が異なっていた．入力コード片によって，SI フィルタの向き不向きがあると考えられる．入力コード片の特徴語とその特徴語が出現する構文の結びつきが強い場合，構文情報によって関係のないコード片を除外することが容易であると考えられる．かんなの実験結果がこの場合に該当する．一方，入力コードの特徴語が出現する構文がまちまちである場合，構文情報の対応を確認することは難しく，類似コードをうまく検出できないと考えられる．SPARS-J の実験結果がこの場合

に該当する．また，フィルタリングによって，適合率にあまり改善が見られなかったことから，フィルタリングの制約を増やすことが必要であると考えられる．例えば，構文情報を比較する際に次のような条件の追加が考えられる．

- 特徴語を含むコードの出現順序を考慮して比較を行う
- 分類する文の種類を増やす

関連語については，実際に生成された関連語の集合が字句の意味的には関連していない場合がある．これは，識別子の共起関係のみに着目して関連語を求めているためである．単語間の関係を分類した辞書であるシソーラス [22, p. 110] を用いることで，検出結果の精度が向上すると考えられる．

最後に Libra の実験結果についての考察を述べる．Libra の実験結果は，本システムに比べて適合率が高く，再現率が低いという結果なっていた．コードクローンとなっているコード片は，トークン列の並びが同じであるため，同じ処理内容を示していると考えられる．そのため，高い適合率になったと考えられる．しかし，処理内容が同じであっても，トークン列の並びは常に同じであると限らないため，低い再現率になったと考えられる．特に，SPARS-J の事例では，型キャストの追加という一般的な内容であったため，ほとんど正解集合を検出できない結果になったと考えられる．

以上のことから，検出のノイズを抑えたい場合には Libra が有効であり，検出漏れを少なくしたい場合には本システムが有効であると考えられる．

6.2 実験の設定に関する考察

実験では，欠陥を含むコード片を入力として与え，同じ欠陥を含むコード片を類似コードとして検出を行った．しかし，同じ欠陥を含んでいなくても構文情報が対応すれば，類似コードとして判定される．そのため，全体的に低い適合率になっていたと考えられる．この問題を解決するには，ツールの出力結果を分類する必要がある．分類の種類は，次の 3 つが考えられる．

- 入力コード片と同じ欠陥を含むモジュール
- 入力コード片と同じ処理内容であるが欠陥を含まないモジュール
- 入力コード片と異なる処理内容のモジュール

これらの分類のうち，入力コード片と同じ処理内容であるが欠陥を含まないモジュールについては，入力コード片の類似コードを検出できたと言える．類似コードを検出するという意味では，正しく検出できていると見なすことができる．

入力コード片を修正された箇所の前後合わせて 3 行と設定した．入力コード片を 1 つのモジュールと見なして特徴語を抽出するため，入力コード片のサイズが小さいと出現回数の小さい識別子が特徴語として抽出されることがある．抽出された特徴語が，実際のソースコード集合では出現回数の小さい識別子であった場合，特徴語として抽出されず，類似コードが検出できない可能性がある．そのため，入力コード片のサイズを変化させることによって，適合率，再現率の変化を調べる必要がある．

また，特徴語の閾値に関しては，閾値の有無のみで実験を行った．クラスタリングの閾値のように値を変化させて，適合率，再現率の変化を調べる必要がある．C 言語のソフトウェアを対象としたが，他言語のソフトウェアでも同じような結果となるか調べる必要がある．

6.3 類似コード検索への適用に関する考察

提案手法や 2.3 節で述べた既存手法等を用いた類似コード検索は，検出可能な類似コードの向き不向きがある．従って，欠陥検出を行う際に，これらの手法を組み合わせることで，より多くの欠陥を検出できることが期待される．例えば，提案手法の結果に grep を用いることで検出漏れを調べるということができる．また，手法の組み合わせにより，欠陥の存在する箇所を絞り込むことが期待される．例えば，提案手法の結果に Libra を用いることで，欠陥が含まれているモジュール中のコードクローンを調べる．調べたコードクローン中に欠陥が含まれている場合，コードクローンとなっている他のコード片にも欠陥が存在していると考えられる．

7 むすび

本稿では，識別子間の共起関係に着目して類似コードを検索する手法を提案した．本手法では，ソースコードから頻出する識別子である特徴語を抽出し，特徴語が出現する構文の情報を比較することで類似したコードを検出する．

また，本手法の有用性を評価するために，欠陥の検出に適用し，実験を行った．C 言語のソフトウェアを対象として実験を行ったところ，再現率はある閾値を境に高い値となった．欠陥を漏れなく検出することが目的であるため，高い再現率が意義のある結果と言える．一方，特徴語の抽出，類似コードの抽出にフィルタリングを行ったが，適合率の改善にあまり影響を与えていなかった．また，フィルタリングの効果が実験対象によって異なっており，入力コード片によって現在用いているフィルタの向き不向きがあると考えられる．今回の実験で用いた入力コード片のサイズは，合計で 3 行という小さい規模であった．入力コード片のサイズによって，入力コード片の特徴語が変化すると考えられるため，入力コード片のサイズを変化させることで，適合率，再現率も変化すると考えられる．関連語については，字句の意味的に関連していない識別子が関連語と見なされることがあり，検出結果に影響を与えていると考えられる．

今後の課題としては，次が挙げられる．

- 他のプログラミング言語を対象とした実験
- フィルタリング条件の追加
- 入力コード片のサイズが出力結果に与える影響の調査
- 関連語の定義方法の検討

謝辞

本研究の全過程を通して，常に適切な御指導，御助言を賜りました 大阪大学大学院情報科学研究科 コンピュータサイエンス専攻 井上 克郎 教授に心より深く感謝致します.

本論文を作成するにあたり，常に適切な御指導，御助言を賜りました 大阪大学大学院情報科学研究科 コンピュータサイエンス専攻 楠本 真二 教授に心から感謝致します.

本論文を作成するにあたり，適切な御指導，御助言を頂きました 大阪大学大学院情報科学研究科 コンピュータサイエンス専攻 松下 誠 准教授に心から感謝致します.

本論文を作成するにあたり，適切な御指導，御助言を頂きました 大阪大学大学院情報科学研究科 コンピュータサイエンス専攻 肥後 芳樹 助教に心から感謝致します.

本論文を作成するにあたり，適切な御指導，御助言を頂きました 大阪大学大学院情報科学研究科 コンピュータサイエンス専攻 早瀬 康裕 特任助教に心から感謝致します.

本研究を通して，様々な御協力を頂きました 大阪大学大学院情報科学研究科 コンピュータサイエンス専攻 吉田 則裕 氏に深く感謝致します.

本研究に対して，産業界の立場から貴重なコメントを頂きました，株式会社富士通研究所 松尾 明彦 氏，小林 健一 氏，前田 芳晴 氏，ならびに株式会社富士通東北システムズ 須藤 茂雄 氏に深く感謝致します.

最後に，その他様々な御指導，御助言等を頂いた 大阪大学大学院情報科学研究科 コンピュータサイエンス専攻 井上研究室の皆様に深く感謝致します.

参考文献

[1] B. S. Baker. Finding Clones with Dup: Analysis of an Experiment. *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 608–621, 2007.

[2] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proc. of ICSM '98*, pp. 368–377, 1998.

[3] I. Dagan, L. Lee, and F. C. N. Pereira. Similarity-Based Models of Word Cooccurrence Probabilities. *Machine Learning*, Vol. 34, No. 1-3, pp. 43–69, 1999.

[4] M. Dorfman and R. H. Thayer. *Software Engineering*. IEEE Computer Society Press, 1997.

[5] N. Ford and M. Woodroffe. *Introducing Software Engineering*. Prentice Hall, 1994.

[6] GNU grep. `http://www.gnu.org/software/grep/`.

[7] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue. Simultaneous Modification Support based on Code Clone Analysis. In *Proc. of APSEC 2007*, pp. 262–269, 2007.

[8] IEEE Std 1219: Standard for Software Maintenance, 1997.

[9] JavaCC. `https://javacc.dev.java.net/`.

[10] L. Jiang, Z. Su, and E. Chiu. Context-Based Detection of Clone-Related Bugs. In *Proc. of ESEC/FSE 2007*, pp. 55–64, 2007.

[11] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multi-Linguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.

[12] M. Kim, L. Bergman, T. Lau and D. Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proc. of ISESE 2004*, pp. 83–92, 2004.

[13] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *Proc. of SAS 2001*, pp. 40–56, 2001.

[14] S. Kullback. *Information Theory and Statistics*. John Wiley and Sons, 1959.

[15] T. M. Pigoski. *Encyclopedia of Software Engineering*, Vol. 1, chapter Maintenance. John Wiley & Sons, 1994.

[16] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*, chapter Fixing the Defect. Morgan Kaufmann, 2005.

[17] 泉田 聡介, 植田 泰士, 神谷 年洋, 楠本 真二, 井上 克郎. ソフトウェア保守のための類似コード検索ツール. 電子情報通信学会論文誌, Vol. J-86-D-I, No. 12, pp. 906–908, 2003.

[18] 井上 克郎, 神谷 年洋, 楠本 真二. コードクローン検索法. コンピュータソフトウェア, Vol. 18, No. 5, pp. 47–54, 2001.

[19] 上田 尚一. クラスター分析. 朝倉書店, 2003.

[20] 北 研二, 津田 和彦, 獅々堀 正幹. 情報検索アルゴリズム. 共立出版, 2002.

[21] 齋藤 堯幸, 宿久 洋. 関連性データの解析法–多次元尺度構成法とクラスター分析法. 共立出版, 2006.

[22] 徳永 健伸. 情報検索と言語処理. 東京大学出版会, 1999.

[23] 松尾 豊, 石塚 満. 語の共起の統計情報に基づく文書からのキーワード抽出アルゴリズム. 人工知能学会論文誌, Vol. 17, No. 3, pp. 217–223, 2002.

[24] 横森 励士, 梅森 文彰, 西 秀雄, 山本 哲男, 松下 誠, 楠本 真二, 井上 克郎. Java ソフトウェア部品検索システム SPARS-J. 電子情報通信学会論文誌, Vol. J-87-D-I, No. 12, pp. 1060–1068, 2004.

[25] 日本語入力システム “かんな”. `http://canna.sourceforge.jp`.

付録

Jensen-Shannon divergence

提案手法で用いている Jensen-Shannon divergence の定義について述べる．

Kullback-Leibler divergence とは，2 つの確率分布間の違いを表す値である．識別子の集合を I，識別子 a，b 間の Kullback-Leibler divergence を $D(a\|b)$ とすると，

$$D(a\|b) = \sum_{I'} P(i \mid a) \log \frac{P(i \mid a)}{P(i \mid b)} \tag{8}$$

$$I' = \{i \in I \mid (i \neq a) \cap (i \neq b)\} \tag{9}$$

と表せる．この値は，0 以上の値を取る．0 となるのは，2 つの確率分布間の差が全くない場合である．また，非対称であり，三角不等式が成り立たないという特徴がある．

Jensen-Shannon divergence とは，2 つの確率分布に対する平均の分布とそれぞれの確率分布間における Kullback-Leibler divergence を平均した値である．識別子 a，b 間の Jensen-Shannon divergence を $J(a, b)$ とすると，

$$J(a, b) = \frac{1}{2}\left[D\left(a \,\middle\|\, \frac{a+b}{2}\right) + D\left(b \,\middle\|\, \frac{a+b}{2}\right)\right] \tag{10}$$

と表せる．ただし，$P(i \mid a)$ については，提案手法では次のように定義している．

$$P(i \mid a) = \frac{\text{識別子 } i, a \text{ が共起するモジュールの数}}{\text{識別子 } a \text{ が出現するモジュールの数}} \tag{11}$$

この値は，Kullback-Leibler divergence が基になっていることから，0 以上の値を取る．また，対称であるという特徴を持っている．