情報処理学会論文誌 Vol. 50 No. 5 1506–1519 (May 2009)

類義語の特定に基づく類似コード片検索法

吉田則裕[†1] 服部剛之[†1]
早瀬康裕[†1] 井上克郎[†1]

ソースコード中に類似した部分（類似コード片）が散在していると，ソースコードの一部分を修正したときにその類似部分に対しても同様の修正をする必要が生じることがあるため，保守作業が困難になるという問題がある．一般に，ソフトウェア開発者は類似コード片を調査する際には，grep などのキーワード検索ツールを用いる．しかし，類似コード片には様々な差異が存在するため，修正の必要がある類似コード片の多くを列挙できるキーワードを与えることは困難である．本研究では，入力したコード片の類似コード片を，コード片に含まれる識別子の類似性に基づいて検索する手法を提案する．本手法は，まず，共起関係に基づいて語（識別子を分割・正規化した後の文字列）をクラスタリングすることで類義語を求める．その後，入力コード片に含まれるすべての語について，同一もしくは類義語である語を含むコード片を検出し，類似コード片として提示する．適用実験として，提案手法を用いて類似した欠陥を含むコード片の検索を行ったところ，類似した欠陥の多くを提示できることを確認した．また，提案手法と既存ツール（grep およびコードクローン検出ツール CCFinder）との比較実験を行い，それぞれの検索結果が持つ特徴を確認した．

Retrieving Similar Code Fragments Based on Synonymous Word Identification

Norihiro Yoshida,[†1] Takeshi Hattori,[†1]
Yasuhiro Hayase[†1] and Katsuro Inoue[†1]

A similar code fragment is a code fragment that has similar part to it in source code, and is generally considered as one of factors that make software maintenance more difficult. If developers modify a code fragment, they have to determine whether or not to modify similar code fragments in source code. For finding similar code fragments, developers can use keyword-based search (e.g. grep). However, it is difficult to determine appropriate search keywords since there are various code fragments to implement similar functions. In this paper, we propose a method to retrieve code fragments that are similar in their identifier names to an input code fragment. In our method, at first, to make clusters of synonymous words in source code, the clustering is performed based on co-occurrence of words that derived from identifier names. Then, code fragments that involve words in an identical cluster are presented as similar code fragments. We show the usefulness of our method through case studies to retrieve similar code fragments involving similar defects. In addition, we present a case study for comparing our proposed method with a code clone detection tool CCFinder and a keyword-based search tool grep.

1. はじめに

ソフトウェア保守を困難にする要因の 1 つとして**類似コード片**が指摘されている[1),2),8),9),11),13),14),18)]．類似コード片とは，ソースコード中の**コード片**（ソースコードの一部分）のうち，一致もしくは類似した要素（識別子や構文など）を含むコード片を持つコード片を指す[⋆1]．類似コード片は，すでに開発された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が理由で作成される[2),10)]．特に，Linux や JDK（Java Development Kit）などの大規模ソースコードは大量の類似コード片を含むことが報告されている[9),14)]．

ソフトウェアの保守を行う際に，あるコード片を修正するとそのすべての類似コード片を見つけ出し修正を行う必要が生じることがある．特に，ソースコード中に欠陥が見つかった場合には，その欠陥を含むコード片の類似コード片を探し，検査する必要がある[13),14),18)]．しかし，ソフトウェア中の類似コード片を人手で探すためには大きな労力が必要となる．特に，大規模ソフトウェアが対象の場合，すべての類似コード片を人手で探すことはより困難となる．

類似コード片の検索に用いることができる方法として，キーワード検索やコードクローン検出法があげられるが，両者とも効果的な検索を行うことができるクエリ（検索質問）を与えることは難しい．キーワード検索を用いる場合，修正を要するコード片から抽出したキーワードを grep[5)] などのツールに与えることで，キーワードを含むコード片を列挙する．しかし，対象ソフトウェアを十分に理解した開発者でなければ，適切なキーワードを抽出する

†1 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

⋆1 類似コード片と近い概念として**コードクローン**がある．コードクローンの定義は研究者により様々であるが，類似コード片の中でも同値関係（例：トークン列が等しい関係など）を持つコード片が存在するもののみを指すことが多い[1),2),8),9),11)]．本稿では，コードクローンより広い概念である類似コード片を扱う．

© 2009 Information Processing Society of Japan

ことが困難である（**問題点 1**）．その要因の 1 つとして，grep などのツールは文字列が完全に一致するコード片のみを出力するため，わずかに異なる文字列を含むコード片であっても検索結果に含めることはできないことがあげられる．一方，コードクローン検出法を用いる場合，コードクローン検出ツールを用いて修正を要するコード片とトークン列が等価なコード片を列挙する[6]．しかし，トークン列に些細な差異（例外処理の有無など）があるコード片を列挙することはできない（**問題点 2**）．

本研究では，容易にクエリを与えることができ，かつクエリと些細な差異がある部分であっても提示できる手法を提案する．提案する手法は，コード片をクエリとして与えると，識別子の類似性に基づいて対象ソースコードから**類似関数**（クエリとして与えられたコード片の類似コード片を 1 つ以上含む関数）を検索する．具体的には，まず自然言語処理の分野で提案されている Dagan らの手法[3] を用いて，**語**（識別子を分割・正規化した後の文字列）の類義語を特定する（3.2 節参照）．次に，**入力コード片**（クエリとして与えられたコード片）に含まれるすべての語と一致する，もしくは類義語である語を含む関数を検出し，類似関数として提示する．

提案手法は，以下の 3 つの特徴を持っている．

- 修正を要するコード片を入力コード片として与えるのみで検索を行うことができる．**問題点 1** で述べたように，grep を用いる場合，適切なキーワードをコード片から抽出する必要がある．
- 入力コード片と類似した処理を表す関数の一部に異なる識別子が含まれていたとしても，類義語を特定することで類似関数として提示できる可能性がある．**問題点 1** の要因の 1 つとしてあげたように，grep を用いると完全に一致する文字列を含む部分のみが出力される．
- コード片が含む識別子の類似性を判定するため，トークン列に些細な差異（例外処理の有無など）がある関数であっても類似関数として提示できる可能性がある．**問題点 2** で述べたように，コードクローン検出法はこのような関数を検出できない．

提案手法を用いて，開発者が類似関数を検索し，確認する手順を以下に示す．

(1)　対象ソースコード中から入力コード片を抽出する．
(2)　類義語の特定に用いる閾値を提案手法に入力する．提案手法は，入力された閾値より距離が小さい語どうしを類義語とする（詳しくは，3.2 節参照）．
(3)　提案手法に入力コード片を与えることで，類似関数を検索する．
(4)　検索結果に含まれた関数が多すぎる，もしくは少なすぎる場合は (2) に戻り，閾値を再入力する．関数が多すぎる場合には閾値を小さくし，少なすぎる場合には閾値を大きくする．
(5)　検索結果に含まれた関数を 1 つ 1 つ確認する．

適用実験では，類似した欠陥を含む関数の検索に対する提案手法の有効性を確認した．具体的には，類似した欠陥（バッファオーバフローエラーや型キャストの欠如）を複数含むソースコードを対象として，欠陥を含むコード片のうちの 1 つを入力コード片として提案手法に与えたときに，類似した他の欠陥が提示されるかどうか確認した．その結果，提案手法は欠陥を含むコード片を 1 つ入力コードとして与えるだけで，類似した欠陥のうちの多くを提示できることが確認できた．また，grep やコードクローン検出ツール CCFinder[9] を用いて同様の実験を行い提案手法の実験結果と比較することで，それぞれの検索結果が持つ特徴を確認した．

以降，2 章では提案手法を説明するための準備として，類似コード片検索について述べ，3 章では提案手法である識別子の類似性に基づく類似コード片検索法について述べる．4 章では類似した欠陥を含むコード片の検索に提案手法を適用した結果について述べ，5 章では 4 章で述べた結果をふまえ考察を行う．6 章では関連する手法を提案している研究について議論し，最後に 7 章でまとめと今後の課題について述べる．

2. 類似コード片検索

本稿では，**類似コード片検索**を “ソースコード中から，クエリ（検索質問）として与えられたコード片と一致もしくは類似した要素（識別子や構文など）を含むコード片を提示すること” と定義する．コード片は 5 項組（ソースファイルを一意に識別できる番号，開始行，開始桁，終了行，終了桁）で表現する．なお，“コード片 CF と一致もしくは類似した要素を含むコード片” を単に**コード片 CF の類似コード片**と呼ぶ．また，“コード片 CF の類似コード片を 1 つ以上含む関数” を単に**コード片 CF の類似関数**と呼ぶ．

図 1 は，修正を要するコード片 CF_1 と，同様の修正を要する類似コード片 CF_2，CF_3 を表している．コード片 CF_1 に含まれる情報を基にクエリを作成し，類似コード片検索を実現したシステムに与えると，その類似コード片 CF_2，CF_3 を提示する．

図 2 の 2 つのコード片は，日本語入力システム “かんな”[23] バージョン 3.6 のソースコードに含まれていた類似コード片である．これらコード片は，ともにバッファオーバフローエラーを含んでいる．具体的には，各コード片の 3〜5 行目がバッファからの読み込み処理を

© 2009 Information Processing Society of Japan

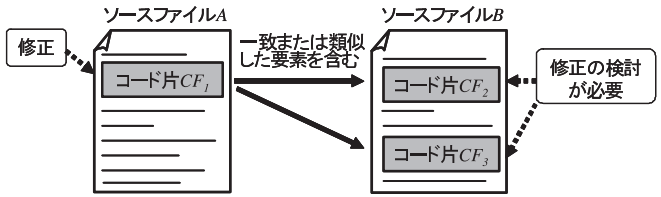


図 1　類似コード片
Fig. 1　Similar code fragments.

```
ir_debug( Dmsg(10, "ProcWideReq7 start!!¥n") );

buf += HEADER_SIZE; Request.type7.context = S2TOS(buf);
buf += SIZEOFSHORT; Request.type7.number = S2TOS(buf);
buf += SIZEOFSHORT; Request.type7.yomilen =(short)S2TOS(buf);
```
バッファからの読み込み処理

(a) 欠陥を含むコード片

```
ir_debug( Dmsg(10, "ProcWideReq14 start!!¥n") );

buf += HEADER_SIZE; Request.type14.mode = L4TOL(buf);
buf += SIZEOFINT;   Request.type14.context = S2TOS(buf);
buf += SIZEOFSHORT; Request.type14.yomi = (Ushort *)buf;
```
バッファからの読み込み処理

(b) 同様の欠陥を含むコード片

図 2　類似コード片間の差異
Fig. 2　Differences between similar code fragments.

表しており，これら処理中にバッファオーバフローエラーを引き起こす可能性がある[⋆1]．そのため，一方のコード片を基にクエリを作成し類似コード片検索を行ったなら，もう一方のコード片が検索結果に含まれることが望ましい．

2.1　grep を用いた類似コード片検索

一般の開発者は，grep[5] を用いて類似コード片検索を行うと考えられる．開発者が grep を用いて修正を要するコード片の類似コード片を検索する手順を以下に示す．

(1)　修正に関連すると思われるキーワード（行，式，識別子など）を抽出する．
(2)　そのキーワードを引数として grep を実行する（**図 4**）．

⋆1 “かんな” バージョン 3.6p1 では，これらの欠陥は修正されていた．**図 3** は，図 2 (a) の欠陥修正した後のコード片である．3，4 行目がバッファオーバフローを検出する部分である．

```
ir_debug( Dmsg(10, "ProcWideReq7 start!!¥n") );

if (Request.type7.datalen != SIZEOFSHORT * 3)
        return( -1 );
buf += HEADER_SIZE; Request.type7.context = S2TOS(buf);
buf += SIZEOFSHORT; Request.type7.number = S2TOS(buf);
buf += SIZEOFSHORT; Request.type7.yomilen = (short)S2TOS(buf);
```
バッファオーバーフローを検知

バッファからの読み込み処理

図 3　図 2 (a) のコード片に対する修正
Fig. 3　Modification to the code fragment in Fig. 2 (a).

```
grep -nr S2TOS *

wconvert.c:2227:    req->datalen = requiredsize = S2TOS(p);
wconvert.c:2319: buf += HEADER_SIZE; Request.type2.context = S2TOS(buf);
wconvert.c:2331: buf += HEADER_SIZE; Request.type3.context = S2TOS(buf);
...
```

図 4　grep を用いた類似コード片検索
Fig. 4　Retrieving similar code using grep.

(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．

grep を用いた方法の問題点は，(1) で適切なキーワードを抽出し検索を行わなければ，効果的な検索を行うことができないことである．たとえば，文全体，もしくは単一の識別子をキーワードとして抽出し検索を行った場合，それぞれ以下の結果になると考えられる．

文全体をキーワードに指定した場合　文字列が完全に一致するコード片のみが出力され，識別子や構文に些細な表現上の差異があったコード片を検索結果に含めることはできない．

単一の識別子をキーワードに指定した場合　大量のコード片が提示されることが多く，検索結果の確認に大きな労力が必要となる可能性が高い．

図 2 を用いて説明すると，文全体（たとえば，図 2 (a) に含まれる Request.type7.yomilen = (short)S2TOS(buf)）をキーワードに指定し検索を行ったとしても，図 2 (b) のいずれの部分も提示されない．また，コード片間に差異があることを考慮し単一の識別子（たとえば buf）を指定すると，欠陥を含まないコード片が大量に提示されやすい．

grep は，キーワード検索に加えて正規表現を用いたパターン検索を行うことができる．しかし，grep の正規表現と対象ソフトウェアの両方を十分に理解した開発者でなければ，正規表現を用いたパターンを適切に指定することは難しい．

© 2009 Information Processing Society of Japan

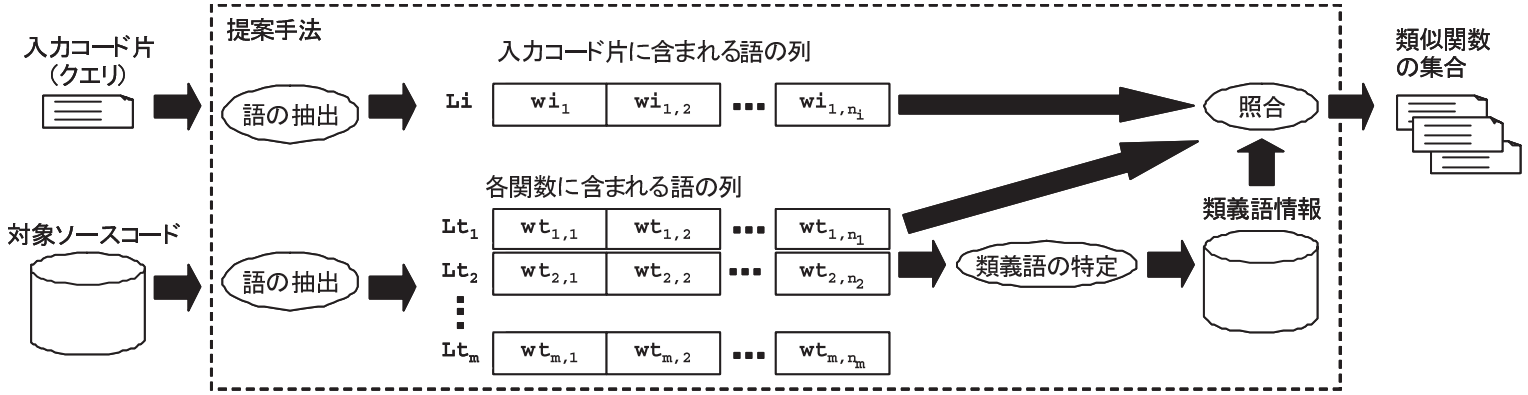


図 5　提案手法の概要
Fig. 5　The overview of the proposed method.

2.2　CCFinder を用いた類似コード片検索

CCFinder[9] などのコードクローン検出ツールを利用して，修正を要するコード片とトークン列が等価なコード片を列挙する方法[6] も考えられる．しかし，この方法では，構文がわずかに異なるなど些細な差異があるコード片を列挙することはできない．

たとえば，図 2 (a) の最終行の右辺は (short)S2TOS(buf) であるが，図 2 (b) の最終行の右辺は (Ushort *)buf であり，字句解析によって得られるトークン列上に差異が存在する．そのため，CCFinder を用いて，一方のコード片とトークン列が等価なコード片を検出したとしても，もう一方のコード片を検出することはできない．

3. 提 案 手 法

本稿では，入力コード片を与えると，識別子の類似性に基づいて対象ソースコードから類似関数を検索する手法を提案する．この手法を用いる開発者は，入力コード片を与えるのみで類似関数を検索できるため，キーワードを指定する必要がない．

提案手法の概要を**図 5** に示す．提案手法は，以下に示す手順からなる．

手順 1（語の抽出）　対象ソースコードの各関数が含む**語**（識別子を分割・正規化した後の文字列）を抽出．

手順 2（類義語の特定）　対象ソースコードが含む語の共起関係に基づいて類義語を特定．

手順 3（入力コード片との照合）　入力コード片が含むすべての語と一致する，もしくは類義語である語を含む関数を類似関数として提示．

以降，手順 1，2，3 について，それぞれ詳述する．

3.1　手順 1（語の抽出）

対象ソースコード中の各関数が含む識別子を抽出し，分割・正規化[⋆1]を行う．本稿では，分割・正規化後の文字列を**語**と呼ぶ．その後，各関数が含む語の出現回数を表す行列を作成する．

図 6 は，語の出現回数を表す行列の例である．この行列は，関数 f_0，f_1，f_2 における語 w_a，w_b，w_c，w_d，w_e の出現回数を表している．以降，この例を用いて説明する．

3.2　手順 2（類義語の特定）

ソースコード中に含まれる語を要素とする集合をクラスタリング（部分集合に分割）し，その結果 1 つのクラスタ（部分集合）に含まれた語を類義語とする[⋆2]．本稿では，**ソースコードから類義語を特定する**とは，上述のようにソースコード中に含まれる語を要素とする

⋆1 行った分割・正規化は，アンダースコアの位置での分割（たとえば，add_host を add と host に分割）や接尾数字の削除（たとえば，type7 の 7 を削除），キャメルケースに従った識別子を大文字から始まる語に分割（たとえば，addHost を add と Host に分割）の全 3 種類である．これらの分割・正規化を行う理由は，分割後の語や接尾数字の削除が行われた語を含む関数を類似関数として提示することで，検索性能を向上させるためである．

⋆2 語集合 S は独立した部分集合 $S_0, S_1, \ldots, S_n$（$S = \bigcup_{i=0}^{n} S_i$ かつ任意の S_i，S_j（$0 \leq i \leq n$，$0 \leq j \leq n$，$i \neq j$）について $S_i \cap S_j = \phi$）に分割される．

© 2009 Information Processing Society of Japan

$$\begin{array}{c} \\ f_0 \\ f_1 \\ f_2 \end{array} \begin{array}{c} \begin{array}{ccccc} w_a & w_b & w_c & w_d & w_e \end{array} \\ \begin{pmatrix} 0 & 2 & 1 & 0 & 1 \\ 1 & 0 & 2 & 3 & 0 \\ 0 & 0 & 2 & 0 & 2 \end{pmatrix} \end{array}$$

図 6　語の出現回数を表す行列の例
Fig. 6　Example of a matrix having the number of occurrences of identifier name.

集合をクラスタリングすることをいう．また，**2 つの語が類義語として特定される**とは，クラスタリングした結果それら語が同一のクラスタに含まれることをいう．

自然文書中に含まれる単語を対象にクラスタリングを行う基準[24]を応用し，ソースコード中に含まれる語を対象にするクラスタリングとして以下の 2 つが考えられる．

(1) 語の共起性　語 w_x と語 w_y が頻繁に**共起**（同一関数など近い位置で出現）しているかどうかを基準とする．もし，頻繁に共起しているなら，同一クラスタに含める[⋆1]．

(2) 共起している語の類似性　語 w_x と共起している語の集合が，語 w_y と共起している語の集合と類似しているかどうかを基準とする．もし，それら集合の類似性が高いなら，語 w_x と語 w_y を同一クラスタに含める[⋆2]．

提案手法では，基準 (2) “共起している語の類似性” の計測モデルの 1 つである Dagan らのモデル[3] に基づくクラスタリングを行う．基準 (1) ではなく基準 (2) の特定法を採用した理由は，自然文書を対象とした実験[3] において，いい換えを行っている単語などの類似した働きをする単語を特定しており，ソースコード中においても類似した概念名（例：入力データサイズと出力データサイズ）の全体または一部を表す語が存在するため，これらを類義語として特定できると考えたからである．また，ソースコードにおいて，ある語と共起している語の集合と，その類義語（その語と類似した概念名を表す語）と共起する語の集合は，同一の関数名や同名の変数名を表す語を多く含み，類似しているのではないかと考えた．前述のとおり，2 つの語の間で，共起する語の集合が類似していれば，基準 (2) の特定法でそれらを類義語と特定することができる．

⋆1 自然文書に含まれる類義語を特定する場合，この基準でクラスタリングを行うとクラスタ全体として 1 つの概念を表すことが多い．たとえば，“doctor”，“nurse”，“hospital” が 1 つのクラスタになる[17]．

⋆2 自然文書に含まれる類義語を特定する場合，この基準でクラスタリングを行うと，類似した働きをする単語（たとえば，いい換えを行っている単語）が同一のクラスタに含まれることが多い．たとえば，“guy” と “kid” は共起している単語の類似性が高いと判定されている[3]．

$$\begin{array}{c} \\ w_a \\ w_b \\ w_c \\ w_d \\ w_e \end{array} \begin{array}{c} \begin{array}{ccccc} w_a & w_b & w_c & w_d & w_e \end{array} \\ \begin{pmatrix} - & 0 & 1 & 1 & 0 \\ 0 & - & 1 & 0 & 1 \\ 1 & 1 & - & 1 & 2 \\ 1 & 0 & 1 & - & 0 \\ 0 & 1 & 2 & 0 & - \end{pmatrix} \end{array}$$

(a) 図 6 を基に作成した共起行列

$$\begin{array}{c} \\ w_a \\ w_b \end{array} \begin{array}{c} \begin{array}{ccccc} w_a & w_b & w_c & w_d & w_e \end{array} \\ \begin{pmatrix} - & 0 & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ 0 & - & \mathbf{1} & \mathbf{0} & \mathbf{1} \end{pmatrix} \end{array}$$

(b) 語 w_a と w_b の行（図 7 (a) から抽出）

図 7　類義語の特定に用いる行列
Fig. 7　Matrices for similar words identification.

なお，本稿では，**2 つの語が共起している**とは，それら語が同一関数内で出現していることをいう．また，**2 つの語が n 回共起している**とは，それら語が n 個の関数内で共起していることをいう．

Dagan らのモデルに基づいてクラスタリングを行う手順は，以下の (A) から (C) である．

(A) 語の共起行列を作成　3.1 節の**手順 1** で作成した語の出現回数を表す行列（図 6）を基に，語の共起回数を表す**共起行列**を作成する．共起行列は，語の数を N とすると，$N \times N$ の対称行列で表される．共起行列の要素（w_x，w_y）は，語 w_x と語 w_y の共起（両者が 1 回以上出現）する関数の数を表す．なお，対角成分は用いないため，記号 “–” をおく．**図 7** (a) は，図 6 を基に作成した 5×5 の共起行列である．この共起行列では，語 w_c と語 w_e は 2 回共起していることを表している．

(B) 語の距離を算出　2 つの語の距離を，それらと共起している語の類似性に基づいて算出する．図 7 (b) は，図 7 (a) から語 w_a と w_b の行を抽出した行列である．太字で表す要素が，語 w_a および語 w_b の他の語（w_c，w_d，w_e）との共起回数を表している．語 w_a と語 w_c，w_d，w_e の共起回数の分布 $[1, 1, 0]$ と，語 w_b と語 w_c，w_d，w_e の共起回数の分布 $[1, 0, 1]$ の距離を算出し，語 w_a と語 w_b の距離とする．2 つの分布の距離は，確率分布間の差異を表す尺度である Kullback-Leibler divergence[12] や Jensen-Shannon divergence[16] を用いて算出することができる．提案手法では，対称性を持つ Jensen-

© 2009 Information Processing Society of Japan

Shannon divergence を用いる．対称性のある Jensen-Shannon divergence を用いる理由は，単語間の距離に対称性を持たせることで，自然言語と同様に類義語であるという関係に対称性を持たせるためである[*1]．また，提案手法を用いる開発者が，検索結果を理解するためには語間の関係に対称性がある方が容易に理解できると考えられる．たとえば，語間の関係に対称性があるなら，単語 A が単語 B の類義語であることを提示するだけで，開発者は単語 B が単語 A の類義語であることが分かるが，対称性がない場合，開発者は単語 B が単語 A の類義語であることを確認する必要がある．

(C) クラスタリング　(B) で求めた距離に基づいて，語のクラスタリングを行う．クラスタリングには様々な方法があるが，提案手法では，すべての語が独立したクラスタという状態から始めて，最も類似度の高いクラスタから順次結合していく方式を採用する．この方式では，クラスタ数が 1 になるか，もしくは任意のクラスタ間の距離が閾値以上になるまで結合を繰り返す．クラスタ間の類似度は**群平均法（平均距離法）**[19),20)] を用いて求める．群平均法は，2 つのクラスタに属する要素間の距離の平均を求め，それらクラスタ間の類似度とする方法である．

3.3　手順 3（入力コード片との照合）

入力されたコード片と対象ソースコード中の各関数を照合することで，類似関数を提示する．本節では，提案手法が検出する語の対応関係および類似関数について述べる．

語の対応関係　2 つの語 w_x，w_y が与えられたとき，語 w_y が語 w_x と一致する，もしくは語 w_y が語 w_x の類義語であるなら，語 w_x は語 w_y と対応関係を持つとする．

類似関数　入力されたコード片 CF_i が含む n 個の語からなる列 $Li = \left[wi_1, wi_2, \ldots, wi_{n-1}, wi_n\right]$ と比較対象の関数 F_t が含む m 個の語からなる列 $Lt = \left[wt_1, wt_2, \ldots, wt_{m-1}, wt_m\right]$ が与えられたとき，列 Li が含む n 個の語 $wi_1, wi_2, \ldots, wi_{n-1}, wi_n$ のすべてに対し，それぞれと対応関係を持つ語が列 Lt 中に存在するなら，関数 F_t をコード片 CF_i の類似関数とする．

図 8 は，入力コード片の語の列 $\left[host, alloc, add, host\right]$ と対象関数の語の列 $\left[\ldots, node, \ldots, alloc, add, \ldots, node, \ldots\right]$ の照合を表している．図中の矢印は，語の対応関係を表している．語 host と語 node が類義語として特定されている場合，入力コード片中のすべての語が対象関数中のいずれかの語と対応関係を持つ．よって，対象関数は類似関数として検出される．

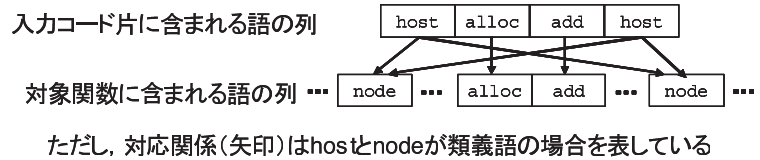


図 8　入力コード片と対象関数の照合
Fig. 8　Matching an input code fragment and a target function.

4. 適 用 実 験

提案手法の有効性を確認するため，提案手法を実装し適用実験を行った．適用実験の目的は，保守対象のソースコード中に欠陥が見つかった際に，同種の欠陥を含む関数を探す作業に対して有効な支援ができているか確認することである．このために，提案手法に**欠陥コード片**（欠陥を含むコード片）を入力コード片として与え，同種の欠陥を含む関数を検索する実験を行った．加えて，grep[5)] やコードクローン検出ツール CCFinder[9)] についても同様の実験を行った．

適用対象には，オープンソースソフトウェア “かんな”[23)] のバージョン 3.6 と我々の研究グループで開発している SPARS-J[25)] を選んだ．適用対象のソフトウェアの構成を**表 1** に示す．なお，これらソフトウェアのソースコードは，C 言語で開発されている．かんなは，クライアント・サーバ型の日本語入力システムである．かんなのバージョン 3.6 において，サーバ機能を実装した server ディレクトリ以下に 19 個のバッファーオーバフローエラー（図 2，図 10）が含まれていた．それら 19 個の欠陥は，18 関数に含まれていた．適用実験では，サーバ機能に含まれるこれら欠陥を探す作業を支援できるか確認するため，server ディレクトリ以下の *.c ファイルを対象とした．SPARS-J は，ソフトウェア部品検索システムである．SPARS-J のあるバージョンにおいて，計 75 個の型キャスト忘れ[*2]がシステム全体にわたって存在した．それら 75 個の欠陥は，50 関数に含まれていた．適用実験では，SPARS-J 全体に含まれるこれら欠陥を探す作業を支援できるか確認するため，すべての *.c ファイルを対象とした．

4.1　提案手法の適用実験

本節で述べる実験の目的は，以下の 2 つである．

⋆1 自然言語において，上位語，下位語のような詳細な単語間の関係には対称性はないが，類義語であるかどうかという関係は一般的に対称性が存在する．

⋆2 ソフトウェア部品を登録するデータベースのデータ表現に合わせるための型キャストが欠如していた．

© 2009 Information Processing Society of Japan

表 1　適用対象のソフトウェア
Table 1　Target software systems.

名前	総行数	関数の数	欠陥関数の数	欠陥数
かんな	7,613	203	18	19
SPARS-J	35,744	859	50	75

目的 1　語のクラスタリング（3.2 節参照）で用いる閾値を変化させることで，類義語として扱う語を増やしたときの検索結果の変化を確認する．

目的 2　入力コード片を変化させたときの検索結果の変化を確認する．

目的 1 に用いる閾値 th_r は，0 以上 1 以下の値をとる r が与えられたときに，以下に手順で求めることができる．

(1)　対象ソフトウェアに含まれる任意の語の距離を求め，その最大値を d_{max} とする．

(2)　最大値と r の積を th_r とする．

なお，実験では，6 つの閾値 $th_{0.0}$，$th_{0.1}, \ldots, th_{0.5}$ を設定した．

目的 2 のために，対象ソースコード中の各欠陥を含むクエリを作成し検索を行った．かんなの場合 19 個，SPARS-J の場合 75 個の入力コード片を作成し，各入力コード片を 1 つずつ与え検索を行った．かんなの検索に与えた各入力コード片は，バッファからの読み込み処理（図 2 参照）を行っている連続する行から構成され，SPARS-J の場合は，型キャスト忘れを含む 1 行もしくは連続する行から構成される．

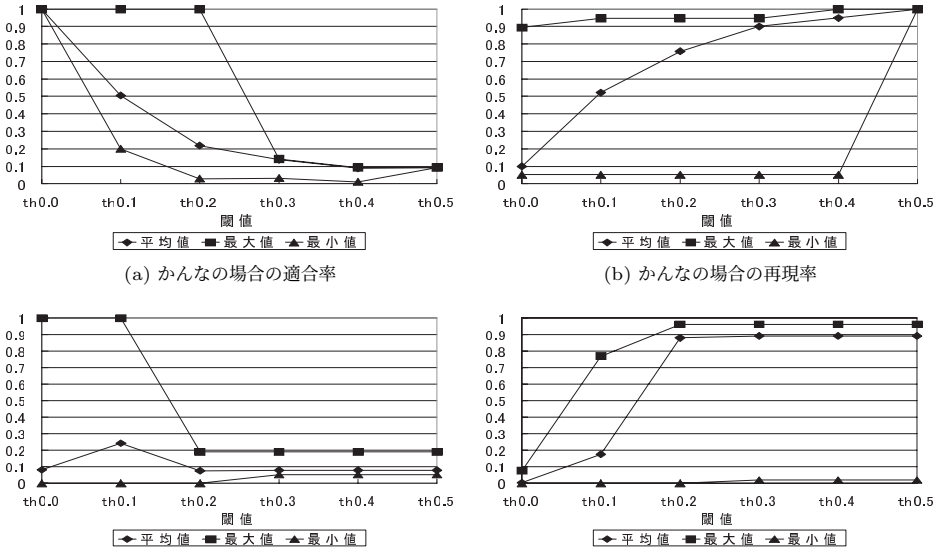
検索結果を評価するために，検索システムの性能評価に用いられる適合率，再現率，F 値を計測した[22]．本稿で用いる適合率 $Precision$，再現率 $Recall$，F 値 F は，D を欠陥関数の集合，R を検索された関数の集合とすると，以下のように表される．

$$Precision = \frac{|D \cap R|}{|R|} \tag{1}$$

$$Recall = \frac{|D \cap R|}{|D|} \tag{2}$$

$$F = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \tag{3}$$

適合率は検索結果の正確性（検索ノイズの少なさ）を表しており，再現率は検索結果の完全性（検索漏れの少なさ）を表す．欠陥関数を探す作業を支援する手法には，検索漏れが少なさを表す再現率が高いことに加え，適合率が高いことが求められる．その理由は，ソフトウェア保守の現場では，全関数を網羅的に検査にするための資源を獲得できるとは限らないため，再現率が高いだけでなく適合率が高いことも求められるからである．F 値は，適合率と再現率の調和平均である．F 値は，検索性能を 1 つのスカラ値として表現でき，また適合率あるいは再現率の一方だけが高い極端な場合も正当に評価することができる[21]．



(a) かんなの場合の適合率　(b) かんなの場合の再現率

(c) SPARS-J の場合の適合率　(d) SPARS-J の場合の再現率

図 9　提案手法の実験結果
Fig. 9　Experimental result of the proposed method.

図 9 は，閾値を $th_{0.0}$ から $th_{0.5}$ まで変化させたときの適合率・再現率の変化を表すグラフである．それぞれのグラフにおいて，閾値ごとに 1 入力コード片あたりの平均値と，全クエリ中の最大値・最小値をプロットしている．図 9 から，閾値 th_r を増加させると，適合率は低下し，再現率は上昇する傾向にあることが分かる．

閾値が $th_{0.0}$ の場合は検出漏れが多かった．また，かんなの場合は閾値を $th_{0.4}$ や $th_{0.5}$ に設定すると，対象ソースコードに含まれるすべてもしくはほとんどの関数を検出し，SPARS-J の場合は $th_{0.3}$ や $th_{0.4}$，$th_{0.5}$ に設定すると同様にすべてもしくはほとんどの関数を検出した．このような検索結果を提示しても欠陥関数の検査作業を支援することはできない．よって，かんなや SPARS-J が対象の場合に有効な支援をできる可能性がある閾値は $th_{0.1}$ および $th_{0.2}$ であるといえる．

© 2009 Information Processing Society of Japan

表 2　各クラスタに属していた語（抜粋）
Table 2　Part of words in each cluster.

(a) $th_r = 0.1$

	語数	コード片 CF_A，CF_B，CF_C のいずれかに含まれた語
クラスタ 1-A	7	buf, HEADER, S2TOS, SIZE, SIZEOFINT, SIZEOFSHORT（6 語）
クラスタ 1-B	43	context, data, debug, dicname, ir, kouho, number, type, Request（9 語）
クラスタ 1-C	5	i, len（2 語）
クラスタ 1-D	2	strlen（1 語）
クラスタ 1-E	12	datalen, ushortstrlen, yomi, yomilen（4 語）
クラスタ 1-F	1	ntohs（1 語）
クラスタ 1-G	4	hinshisize, kouhosize（2 語）

(b) $th_r = 0.2$

	語数	コード片 CF_A，CF_B，CF_C のいずれかに含まれた語
クラスタ 2-A	77	buf, ntohs, HEADER, S2TOS, SIZE, SIZEOFINT, SIZEOFSHORT（7 語）
クラスタ 2-B	126	context, data, datalen, debug, dicname, i, ir, kouho, len, number, strlen, type, ushortstrlen, yomi, yomilen, Dmsg, Request, Ushort（18 語）
クラスタ 2-C	4	hinshisize, kouhosize（2 語）

表 2 は，かんなを対象として，閾値を $th_{0.1}$ もしくは $th_{0.2}$ に設定したときのクラスタリング結果を表している．クラスタ数が多く，また膨大な数の単語を含むクラスタが多く存在したため，一部のクラスタに含まれた一部の単語のみを抜粋している．閾値を $th_{0.1}$ から $th_{0.2}$ に変化させると，クラスタ 1-A と 1-F が結合されクラスタ 2-A になり，クラスタ 1-B, 1-C，1-D，1-E が結合されクラスタ 2-B になった．クラスタ 1-G についてはいずれのクラスタとも結合しなかった（表 2 (b) のクラスタ 2-C）．

語のクラスタリング結果が検索結果にどのように影響したかを考察するため，入力コード片を 3 つ選び，閾値を $th_{0.1}$，$th_{0.2}$ に設定したときの検索結果を示す．なお，これらの閾値を選んだ理由は，提案手法が有効に機能する入力コード片と有効に機能しない入力コード片の両者が存在するため，提案手法の有効性が高い場合と低い場合を比較・議論しやすいからである（他の閾値を選ぶと，提案手法が有効に機能する入力コード片が非常に少ない）．検索結果に関して，以下に示す特徴がある 3 つのコード片 CF_A，CF_B，CF_C（**図 10**）を選んだ．

コード片 CF_A　閾値を $th_{0.1}$ から $th_{0.2}$ に変化させると，適合率が大きく低下し，かつ再現率が上昇したコード片．

```
buf += HEADER_SIZE; Request.type7.context = S2TOS(buf);
buf += SIZEOFSHORT; Request.type7.number = S2TOS(buf);
buf += SIZEOFSHORT; Request.type7.yomilen = (short)S2TOS(buf);
```

(a) コード片 CF_A

```
buf += SIZEOFINT; Request.type10.kouho = (short *)buf; /* short? */
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]));
}
```

(b) コード片 CF_B

```
buf += HEADER_SIZE; Request.type13.context = S2TOS(buf);
len = SIZEOFSHORT ;
buf += len;
Request.type13.dicname = (char *)buf;
len = strlen( (char *)buf ) + 1;
buf += len;
Request.type13.yomi = (Ushort *)buf;
len = ((int)Request.type13.datalen - len - SIZEOFSHORT * 4) / SIZEOFSHORT;
for( data = Request.type13.yomi, i = 0; i < len; i++, data++)
  *data = ntohs( *data );
buf += (ushortstrlen((Ushort *)buf) + 1) * SIZEOFSHORT;
Request.type13.yomilen = S2TOS(buf);
buf += SIZEOFSHORT; Request.type13.kouhosize = S2TOS(buf);
buf += SIZEOFSHORT; Request.type13.hinshisize = S2TOS(buf);
```

(c) コード片 CF_C

図 10　コード片 CF_A，CF_B，CF_C
Fig. 10　Code fragments CF_A, CF_B, and CF_C.

コード片 CF_B　閾値を $th_{0.1}$ から $th_{0.2}$ に変化させると，適合率はほとんど変化しなかったが，再現率が上昇したコード片．

コード片 CF_C　閾値を $th_{0.1}$ から $th_{0.2}$ に変化させると，適合率は低下したが，再現率は変化しなかったコード片．

表 3 (a) に，クラスタリングの閾値を $th_{0.1}$ もしくは $th_{0.2}$ に設定した提案手法に対して，コード片 CF_A，CF_B，CF_C を入力したときの適合率・再現率・F 値を示す．最も F 値が良かった場合は，$th_{0.1}$ に設定したときのコード片 CF_A であった．次に良かった場合は，$th_{0.2}$ に設定したときのコード片 CF_A と CF_B であり，同じ値（0.31）であった（なお，これら

© 2009 Information Processing Society of Japan

表 3　各コード片の検索結果
Table 3　Retrieval results with each code fragment.
(a) 提案手法

	提案手法（$th_r = 0.1$）			提案手法（$th_r = 0.2$）		
	適合率	再現率	F 値	適合率	再現率	F 値
コード片 CF_A	0.50	0.72	0.59	0.18	1.00	0.31
コード片 CF_B	0.20	0.33	0.25	0.18	1.00	0.31
コード片 CF_C	1.00	0.06	0.11	0.33	0.06	0.10

(b) grep，CCFinder

	grep（キーワード：buf）			CCFinder		
	適合率	再現率	F 値	適合率	再現率	F 値
コード片 CF_A	0.19	1.00	0.32	1.00	0.06	0.11
コード片 CF_B	0.19	1.00	0.32	1.00	0.06	0.11
コード片 CF_C	0.19	1.00	0.32	1.00	0.06	0.11

表 4　コード片に含まれる語に属するクラスタ
Table 4　Clusters to which each code fragment belongs.
(a) $th_r = 0.1$

	対応するクラスタ
コード片 CF_A	クラスタ 1-A，クラスタ 1-B，クラスタ 1-E
コード片 CF_B	クラスタ 1-A，クラスタ 1-B，クラスタ 1-C
コード片 CF_C	クラスタ 1-A，クラスタ 1-B，クラスタ 1-C，クラスタ 1-D，クラスタ 1-E，クラスタ 1-F，クラスタ 1-G

(b) $th_r = 0.2$

	対応するクラスタ
コード片 CF_A	クラスタ 2-A，クラスタ 2-B
コード片 CF_B	クラスタ 2-A，クラスタ 2-B
コード片 CF_C	クラスタ 2-A，クラスタ 2-B，クラスタ 2-C

は，まったく同じ検索結果であった）．以降，$th_{0.1}$ に設定したときのコード片 CF_B，$th_{0.1}$ に設定したときのコード片 CF_C，$th_{0.2}$ に設定したときのコード片 CF_C の順であった．

表 4 に，各コード片に含まれる語が属したクラスタを示す．まず，閾値が $th_{0.1}$ の場合について述べる．コード片 CF_A と CF_B の検索結果は表 3 (a) で示したとおり大きく異なっていたが，対応したクラスタは一部分のみが異なっていた．具体的には，コード片 CF_A は語 yomilen を含んでいるため，クラスタ 1-E が対応し，コード片 CF_B は語 i を含んでいるため，クラスタ 1-C が対応したという点のみ異なっていた．語 i を含むコード片 CF_B のように，多くの関数に出現する語を含むコード片を用いて検索を行うとクラスタ 1-C が対応し，検索結果の F 値が低かった．また，コード片 CF_C は語 strlen や ntohs，hinshisize，kouhosize を含んでいるため，語数の少ないクラスタ 1-D，1-F，1-G が対応していた．そのため，検索結果の再現率は低かった．

閾値を $th_{0.2}$ に設定すると，コード片 CF_A と CF_B には同じクラスタ群（クラスタ 2-A および 2-B）が対応したため，まったく同じ検索結果が得られた．閾値が $th_{0.1}$ の場合に異なるクラスタであったクラスタ 1-C と 1-E は，クラスタ 2-B に包含されていた．コード片 CF_C については，クラスタ 2-A および 2-B に加えて，クラスタ 2-C も対応した．クラスタ 2-C は，閾値を $th_{0.1}$ に設定したときに対応した語数の少ないクラスタ 1-G と同一の語から構成されていた．その結果，閾値を $th_{0.1}$ に設定したときと同様に，検索結果の再現率は低かった．

4.2　grep や CCFinder との比較実験

提案手法と grep，CCFinder の有効性を比較するために実験を行った．grep を用いた実験では，コード片 CF_A，CF_B，CF_C から抽出したキーワードを grep に与えることで，キーワードを含む関数を検出し，類似関数の検索を行った．また，CCFinder を用いた実験では，各コード片とコードクローンになっているコード片を含む関数を検出することで，類似関数の検索を行った．

表 3 (b) は，コード片 CF_A，CF_B，CF_C を入力して，grep や CCFinder を用いた実験を行った結果である．grep の結果については，識別子 buf をキーワードとして指定した場合のみを掲載している．識別子 buf を代表例として選んだ理由は，対象とする欠陥である buf が指すバッファのオーバフローに最も関係が深いため，一般の開発者が grep のキーワードに指定する可能性が最も高いと考えられるからである（他のキーワードを指定した場合については，**表 5** に掲載）．識別子 buf をキーワードとして指定した場合の grep は，再現率は 1.00 であり非常に高かったが，適合率は 0.19 であり低かった．CCFinder については，各コード片を含む関数しか検索結果に含まれなかった．F 値を基準に各検索結果の比較を行うと，提案手法の閾値を $th_{0.1}$ に設定したときのコード片 CF_A が最高値（0.59）であった．次いで，grep と閾値を $th_{0.2}$ に設定したときのコード片 CF_A，CF_B がほぼ同じ値（0.31〜0.32）で並び，その次は閾値を $th_{0.1}$ に設定したときのコード片 CF_B であった．その他の場合は，各コード片を含む関数しか検索結果に含まれず，F 値も低い値（0.10〜0.11）であった．

表 5 は，buf およびそれ以外のキーワードをコード片 CF_A，CF_B，CF_C から抽出し，grep に与えたときの検索結果を表している．いずれのコード片から抽出したキーワードに

© 2009 Information Processing Society of Japan

表 5　grep による検索の結果
Table 5　Experimental result of grep.

(a) コード片 CF_A，CF_B，CF_C のすべてに含まれる部分をキーワードとして与えた場合

キーワード	検出行数	適合率	再現率	F 値
buf	557	0.19	1.00	0.31
HEADER_SIZE	46	0.67	1.00	0.80
Request	323	0.17	1.00	0.30
context	211	0.17	0.94	0.29
S2TOS	40	0.94	0.94	0.94
SIZEOFSHORT	87	0.53	0.89	0.67
buf += HEADER_SIZE	18	1.00	1.00	1.00
buf += SIZEOFSHORT	30	1.00	0.89	0.94

(b) コード片 CF_A にのみ含まれる部分をキーワードして与えた場合

キーワード	検出行数	適合率	再現率	F 値
type7	11	0.33	0.06	0.10

(c) コード片 CF_B にのみ含まれる部分をキーワードして与えた場合

キーワード	検出行数	適合率	再現率	F 値
type10	17	0.25	0.06	0.10
kouho	48	0.25	0.06	0.09
ir_debug	260	0.18	1.00	0.31
Dmsg	266	0.18	1.00	0.30

(d) コード片 CF_C にのみ含まれる部分をキーワードして与えた場合

キーワード	検出行数	適合率	再現率	F 値
type13	21	0.50	0.06	0.10
len	398	0.10	0.56	0.17
dicname	148	0.09	0.22	0.13
strlen	70	0.07	0.17	0.10
yomi	129	0.19	0.28	0.23

おいても，キーワードにより結果が大きくばらついた．

5. 考　　察

5.1　語のクラスタリングに用いる閾値について

閾値の増加にともない再現率が増加していた．このことから，類義語の範囲を広げることで，より多くのコード片を類似関数として検索結果に含めることができたと考えられる．

語のクラスタリングに用いる閾値が $th_{0.0}$ の場合は検出漏れが多かった．また，かんなの場合は閾値を $th_{0.4}$ や $th_{0.5}$ に設定すると，対象ソースコードに含まれるすべてもしくはほとんどの関数を検出し，SPARS-J の場合は $th_{0.3}$ や $th_{0.4}$，$th_{0.5}$ に設定すると同様にすべてもしくはほとんどの関数を検出した．このような検索結果を提示しても欠陥関数の検査作業を支援することはできない．よって，かんなや SPARS-J が対象の場合に有効な支援をできる可能性がある閾値は $th_{0.1}$ および $th_{0.2}$ であるといえる．

以上のことから，かんなや SPARS-J を対象とした場合，再現率を重視するならば閾値を $th_{0.2}$ に設定し，逆に適合率を重視するならば閾値を $th_{0.1}$ に設定すると良いと考えられる．4.1 節で述べたように，ソフトウェア保守の現場ではすべての関数を網羅的に検査するための資源を獲得できるとは限らないため，適合率を重視し，閾値を $th_{0.1}$ に設定する状況は十分に考えられる．

対象ソフトウェアによって有効な閾値が変化する可能性があるため，他のソフトウェアを対象とした実験を通して，多くのソフトウェアに有効な閾値の決定方法を考案する必要がある．現状では，対象ソースコードに依存しない一般に有効な閾値の決定法を実現できていない．よって，有効な閾値を求めるために，何度か検索を繰り返さなければならない場合が多いと考えられる．

多くの閾値において，最大値・平均値・最小値の大きな差があった．提案手法の有効性を高めるためには，これら値の差を縮める必要があると考えられる．

5.2　語のクラスタリングについて

クラスタ 1-A には，buf に加算する定数名（例：HEADER_SIZE）や，buf の特定ビット列を取得するためのマクロ名が含まれていた．buf と他の語が同一関数中で共起することが多かったため，これらの語の共起回数の分布が類似し，同一クラスタに含まれていたと考えられる．

クラスタ 1-B，1-E には，構造体の要素を指定する語が含まれていた．構造体の各要素を指定するためには，これらの語を同時に使用する必要があるため，共起回数の分布が類似し，クラスタ 1-B もしくはクラスタ 1-E に含まれたと考えられる．なお，閾値を $th_{0.2}$ に

© 2009 Information Processing Society of Japan

設定すると，クラスタ 1-B とクラスタ 1-E は結合され，1 つのクラスタになった．

クラスタ 1-C は，C 言語を用いた開発においてよく用いられる語を含んでいた．これらの語は，比較的多くの関数に共通して用いられたため，共起回数の分布が類似し，同一クラスタに含まれた．しかし，特定のプログラミング言語を用いた開発において，よく用いられる語（たとえば，i と len）が，類似した役割を担っている場合は少ないと考えられる．よって，このような語をフィルタリングする方法を検討する必要があると考えられる．

クラスタの中には，含まれる語の数が少ないものがあった（クラスタ 1-C，1-D，1-F，1-G，2-C）．よって，すべてのクラスタを含むコード片だけでなく，1 つ以上のクラスタを含むコード片を提示できる方法に改善する必要があると考えられる．しかし，この方法を採用すると，提示する関数の数が膨大になりやすい．そのため，優先して修正の検討をすべきコード片から提示する方法を新たに考案する必要がある．たとえば，提示する関数を対応関係の数に基づいて順位付けし，上位の関数から順に提示する方法が考えられる．

5.3 grep との比較について

F 値を基準に比較すると，閾値を $th_{0.1}$ に設定した提案手法にコード片 CF_A を与えた場合が最も高い値であり，buf をキーワードとして grep に与えた場合よりも高い値であった．次いで，閾値を $th_{0.1}$ に設定した提案手法にコード片 CF_A もしくは CF_B を与えた場合の検索結果が高い F 値であり，buf をキーワードとして grep に与えた場合とほぼ同じ値であった．その他の場合は，grep の方が高い値であった．

提案手法が提示した検索結果の F 値が，buf をキーワードとして grep に与えたときよりも低い値になった場合があった要因として，以下が考えられる．

(1) 多くの関数に出現する語からなるクラスタ　コード片 CF_B は語 i を，コード片 CF_C は語 i，len を含んでおり，これらコード片を入力として検索した場合は，いずれの閾値の場合もコード片 CF_A より F 値が低かった．また，閾値を $th_{0.1}$ に設定したとき，語 i，len は両者ともクラスタ 1-C に属した．5.2 節で述べたように，クラスタ 1-C に含まれるような，C 言語を用いた開発においてよく用いられる語が類似した役割を担っていることは少ないと考えられる．

(2) 語数の少ないクラスタ　再現率が低かったコード片 CF_C に属する語は，他のコード片に属する語に比べて，語数の少ないクラスタ（クラスタ 1-C，1-D，1-F，1-G，2-C）に含まれていることが多かった．入力コード片が語数が少ないクラスタをいくつか含むと，提案手法が提示する関数の数が大きく減り，再現率が下がると考えられる．

これらの要因の影響を受けなかったと考えられる閾値を $th_{0.1}$ に設定したコード片 CF_A，および閾値を $th_{0.2}$ に設定したコード片 CF_A，CF_B については，提案手法の結果は grep より高い，もしくはほぼ同じ F 値であった．これら要因の影響を受けない場合，提案手法は grep よりも高い，もしくは同等の有効性を持つ可能性があると考えられる．これら要因の影響を低減させるためには，5.2 節で述べたように，多くの関数に出現する語をフィルタリングする手法や，すべてのクラスタを含むコード片だけでなく 1 つ以上のクラスタを含むコード片を提示できる手法に改善する必要があると考えられる．

コード片 CF_B については，閾値を $th_{0.1}$ に設定した場合は grep と比べて再現率・F 値が低かったが，閾値を $th_{0.2}$ に変化させると，再現率・F 値が上昇し，grep と同等の検索性能であった．これは，閾値を $th_{0.1}$ に設定したときに存在した少数の語からなるクラスタ 1-C（5 語）が，閾値を $th_{0.2}$ に変化させると他の複数のクラスタと結合し，多数の語からなるクラスタ 2-B（126 語）が構成され，多くの関数を提示するようになったためと考えられる．一方，コード片 CF_A については，閾値を $th_{0.1}$ に設定した場合に，すでに grep と比べて適合率・F 値が高かった．そのため，閾値を $th_{0.2}$ に変化させると，クラスタ 1-E がクラスタ 1-C など他のクラスタと結合して語数の多いクラスタ 2-B を構成し，適合率が下がったと考えられる．これらのことから，以下のことがいえる．

- 閾値を上昇させると，コード片間の検索結果の差異が小さくなる．
- 語数の少ないクラスタが原因で再現率が低い場合に閾値を上昇させると，そのクラスタと他のクラスタが結合し，適合率がは下がるが再現率が大幅に上昇する可能性がある．
- すでに検索結果が良い場合に閾値を上昇させると，入力コード片に対応するクラスタが他のクラスタと結合し，再現率は上昇するものの適合率が大幅に下がる可能性がある．

grep の実験については，主に buf をキーワードとして与えた場合を取り上げたが，表 5 に示すとおり，キーワードにより結果が大きくばらついている．キーワードを適切に指定できる開発者であれば grep の有効性は高いといえるが，逆にキーワードを適切に指定できない開発者あれば，提案手法の方が有効性が高い場合があると考えられる．

5.2 節で述べたように，提案手法は対象ソースコードに依存しない一般に有効な閾値の決定法を実現できていないため，有効な閾値を決めるための作業量が必要となる．grep は，コード片からキーワードを抽出する必要がある代わりに，閾値を設定する必要がないため，全体的な作業量では grep の方が少ない場合もありうると考えられる．これらのことから，提案手法と grep などの各ツールについて全体の作業量の比較実験を行う必要があると考えられる．

行単位で検出を行う grep の方がより詳細に欠陥を含むコード片の位置を提示できると考

© 2009 Information Processing Society of Japan

えられるが，grep を用いて検出した各行を確認することで検査を行うことが難しい場合もある．具体的には，複数行からなるコード片や小規模の関数全体が 1 つの欠陥を表している場合に grep を用いると，これらコード片中や関数中に含まれる数行を提示することが多く，提示された各行を確認しただけでは欠陥の有無を確認することは難しい．

5.4 CCFinder との比較について

本稿の適用実験では，トークン列から連続して一致する部分列を検出する CCFinder を用いて，入力コード片のトークン列と連続して一致するトークン列を含む関数を検出した．しかし，（入力コード片を含む関数を除く）すべての欠陥関数は連続して一致するトークン列を持たなかったため，検出されなかった．よって，コード片 CF_A，CF_B，CF_C を入力コード片として与える場合は，トークン列でなく識別子の類似性に基づいて類似関数を提示する提案手法の方が有効であるといえる．

本来 CCFinder は，コードクローン検出ツールとして開発されているため，類似コード片検索を行う多くの場合において有効性は低いと考えられる．しかし，対象とする類似コード片によっては，CCFinder のように等価なトークン列を検索した方が有効な場合もあると考えられるため，提案手法と CCFinder を使い分ける必要があると考えられる．提案手法と CCFinder の比較実験をさらに行うことで，それぞれが有効な類似コード片の性質を明らかにする必要がある．

6. 関 連 研 究

これまでに様々なコードクローン検出法が提案されており，その中には CCFinder などのトークン列の等価性に基づく検出法だけでなく，抽象構文木やプログラム依存グラフの等価性に基づく検出法も提案されている[2),8),11)]．抽象構文木の等価性に基づく検出法の中には，抽象構文木が完全一致しなくても，主要な構文要素が一致していれば，コードクローンとして検出する手法も提案されている[8)]．また，プログラム依存グラフの等価性に基づく手法[11)] は，構文が等価でなくてもプログラム依存グラフが等価であれば，コードクローンとして検出する．提案手法は，識別子の類似性に基づいて類似コード片を検索するため，構文上やプログラム依存グラフが異なるコード片であっても検索結果に含めることができる．対象とする類似コード片によっては，上述のコードクローン検出法を用いて類似コードを検索した方が有効な場合もあると考えられる．よって，提案手法とこれらコードクローン検出法の比較実験を行うことで，各手法が有効な類似コード片の特徴を明らかにする必要がある．

Splint[4)] などの lint 系のツールや FindBugs[7)] のように，ソースコード中から検査すべき部分を検出するツールが数多く開発されている．これらツールの多くには，初期化されていない変数への参照や 1 度も参照されない変数を検出するアルゴリズムが実装されている．しかし，ドメインやアプリケーションに特化した欠陥を検出する機能を追加するためには，ツールを拡張する必要がある．提案手法は入力コード片を必要とする代わりに，対象ソースコードやそのドメインに特化した欠陥であっても，欠陥を含むコード片を抽出し入力コード片として与えるだけで類似コード片を検索することができるため，ドメインやアプリケーションに特化した欠陥を検出できる可能性がある．

Li らが提案する PR-Miner は，Frequent Itemset Mining アルゴリズムを用いて，変数名や関数名の出現パターンを特定し，パターンに違反する変数名や関数名の出現を欠陥候補として提示する[15)]．PR-Miner によって，欠陥を含むコード片を検出し，類義語の特定を行う提案手法を用いてそのコード片の類似コード片を検索すると，新たな欠陥を検出できる可能性がある．また，Li らは，コピーアンドペーストにより作り込まれたコード片に対して一貫した識別子の修正が行われているか判定し，一貫した修正が行われていないコード片を欠陥候補として検出する手法を提案している[14)]．この手法についても，PR-Miner と同様に，欠陥を含むコード片を検出し提案手法に入力すると新たな欠陥を検出できる可能性がある．

適用実験において，提案手法を用いて欠陥があるコード片の類似関数を検索することで，欠陥関数を検出した．しかし，提案手法が行うことはあくまで類似コード片検索であるため，欠陥を含む検出が目的の場合は上述の欠陥検出ツールの出力結果と照合し，絞り込みや順位付け（たとえば，複数のツールが欠陥を検出したコード片を上位に順位付け）を行う必要があると考えられる．

7. ま と め

本稿では，入力として与えたコード片に類似したコード片を検索する手法を提案した．提案手法は，完全に一致する識別子を含むコード片を検出するだけでなく，類義語を含むコード片を類似コード片として検出することができる．提案手法を実装し，複数の類似した欠陥を持つソースコードに対して適用したところ，有効な検索を行うためには類義語の特定に用いる閾値を適切に決める必要があることが分かった．さらに，提案手法と grep，コードクローン検出ツール CCFinder にそれぞれ同じ入力コード片を与え類似コード片検索を行うことで，結果の比較を行った．提案手法と grep を比較すると，提案手法が有効に働く入力コード片と grep が有効に働く入力コード片がそれぞれ存在した．一方，CCFinder は（入力コード片を含む関数を除く）欠陥関数を検出することはできなかったため，提案手法の方が

© 2009 Information Processing Society of Japan

有効な検索を行えたといえる．

今後，検索性能を向上させるために，入力コード片と対象コード片の照合方法を改善（5.2 節参照）したいと考えている．さらに，現状の提案手法は構文情報を用いていないため，構文情報を用いた検索を実現したいと考えている．具体的には，特定した類義語を含む部分の構文が一致しているコード片のみを提示する手法に改善したいと考えている．有用性の調査として，他のソフトウェアに含まれる欠陥の検出を行い，提案手法の性能を評価する必要がある．CCFinder をはじめとするコードクローン検出ツールや grep との比較実験も行いたいと考えている．比較実験では，適合率や再現率，F 値などの検索性能の比較だけでなく，使用者の作業量も比較する必要がある．また，同時に機能拡張する必要のある関数の検出といった他の用途への適用も考えている．

謝辞　本研究を進めるにあたり貴重なコメントをいただいた株式会社富士通研究所松尾昭彦氏，小林健一氏，前田芳晴氏，株式会社富士通東北システムズ須藤茂雄氏，大阪大学大学院情報科学研究科松下誠氏に深く感謝する．本研究は一部，文部科学省「次世代 IT 基盤構築のための研究開発」の委託に基づいて行われた．また，日本学術振興会科学研究費補助金基盤研究（A)（課題番号：17200001），特別研究員奨励費（課題番号：20・1964）の助成を得た．

参 考 文 献

1) Baker, B.S.: Finding Clones with Dup: Analysis of an Experiment, *IEEE Trans. Softw. Eng.*, Vol.33, No.9, pp.608–621 (2007).
2) Baxter, I.D., Yahin, A., Moura, L., Anna, M.S. and Bier, L.: Clone Detection Using Abstract Syntax Trees, *Proc. ICSM'98*, pp.368–377 (1998).
3) Dagan, I., Lee, L. and Pereira, F.C.N.: Similarity-Based Models of Word Cooccurrence Probabilities, *Machine Learning*, Vol.34, No.1-3, pp.43–69 (1999).
4) Evans, D. and Larochelle, D.: Improving security using extensible lightweight static analysis, *IEEE Software*, Vol.19, No.1, pp.42–51 (2002).
5) GNU grep. http://www.gnu.org/software/grep/
6) Higo, Y., Ueda, Y., Kusumoto, S. and Inoue, K.: Simultaneous Modification Support based on Code Clone Analysis, *Proc. APSEC 2007*, pp.262–269 (2007).
7) Hovemeyer, D. and Pugh, W.: Finding bugs is easy, *ACM SIGPLAN Notice*, Vol.39, No.12, pp.92–106 (2004).
8) Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones, *Proc. ICSE 2007*, pp.96–105 (2007).
9) Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, *IEEE Trans. Softw. Eng.*, Vol.28, No.7, pp.654–670 (2002).
10) Kim, M., Bergman, L., Lau, T. and Notkin, D.: An Ethnographic Study of Copy and Paste Programming Practices in OOPL, *Proc. ISESE 2004*, pp.83–92 (2004).
11) Komondoor, R. and Horwitz, S.: Using Slicing to Identify Duplication in Source Code, *Proc. SAS 2001*, pp.40–56 (2001).
12) Kullback, S.: *Information Theory and Statistics*, John Wiley and Sons (1959).
13) Laguë, B., Proulx, D., Mayrand, J., Merlo, E.M. and Hudepohl, J.: Assessing the Benefits of Incorporating Function Clone Detection in a Development Process, *Proc. ICSM'97*, pp.314–321 (1997).
14) Li, Z., Lu, S., Myagmar, S. and Zhou, Y.: CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code, *IEEE Trans. Softw. Eng.*, Vol.32, No.3, pp.176–192 (2006).
15) Li, Z. and Zhou, Y.: PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code, *Proc. ESEC/FSE 2005*, pp.306–315 (2005).
16) Lin, J.: Divergence Measures based on the Shannon Entropy, *IEEE Trans. Inf. Theory*, Vol.37, No.1, pp.145–151 (1991).
17) Tanaka-Ishii, K. and Iwasaki, H.: Clustering Co-occurrence Graph based on Transitivity, *Proc. WVLC'97*, pp.91–100 (1997).
18) Zeller, A.: *Why Programs Fail*, Morgan Kaufmann Pub. (2005).
19) 上田尚一：クラスター分析，朝倉書店 (2003).
20) 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).
21) 廣田啓一，佐々木裕：用語解説「F 値」，日本ファジィ学会誌，Vol.12, No.3, p.36 (2000).
22) 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).
23) 日本語入力システム “かんな”．http://canna.sourceforge.jp
24) 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).
25) 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 D-I，Vol.J87-D-I, No.12, pp.1060–1068 (2004).

(平成 20 年 6 月 24 日受付)
(平成 21 年 2 月 4 日採録)

© 2009 Information Processing Society of Japan

吉田　則裕（正会員）

平成 16 年九州工業大学情報工学部知能情報工学科卒業．平成 21 年大阪大学大学院博士後期課程修了．現在，日本学術振興会特別研究員．博士（情報科学）．コードクローン分析およびリファクタリング支援の研究に従事．電子情報通信学会，人工知能学会，IEEE 各会員．

服部　剛之

平成 18 年大阪大学基礎工学部情報科学科卒業．平成 20 年同大学大学院博士前期課程修了．現在，日立電子サービス株式会社に勤務．在学中，コードクローン分析の研究に従事．

早瀬　康裕（正会員）

平成 14 年大阪大学基礎工学部情報科学科卒業．平成 19 年同大学大学院博士後期課程修了．現在，同大学特任助教．博士（情報科学）．オープンソースソフトウェア開発，ソフトウェア保守の研究に従事．IEEE-CS 会員．

井上　克郎（フェロー）

昭和 54 年大阪大学基礎工学部情報工学科卒業．昭和 59 年同大学大学院博士課程修了．同年同大学基礎工学部情報工学科助手．昭和 59〜61 年ハワイ大学マノア校情報工学科助教授．平成 1 年大阪大学基礎工学部情報工学科講師．平成 3 年同学科助教授．平成 7 年同学科教授．平成 14 年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教授．平成 20 年国立情報学研究所客員教授．同年情報処理学会フェロー．同年電子情報通信学会フェロー．工学博士．ソフトウェア工学の研究に従事．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．

© 2009 Information Processing Society of Japan