プログラム理解を支援するコンセプトキーワードの自動抽出法 ckTF/IDF 法の提案

大場 勝[†] 権藤克彦[†]

本論文では，識別子からコンセプトキーワードを発見するための ckTF/IDF（Concept Keyword Term Frequency/Inverse Document Frequency）法を提案する．ckTF/IDF 法は大規模なソフトウェアにおけるコンセプトキーワードの抽出に適している．その理由は以下の 2 つである．1 つ目は，ckTF/IDF 法は TF/IDF 法に比べ非常に軽量であること．2 つ目は識別子からコンセプトキーワードを抽出するための発見的手法を導入している点である．我々は，教育用 OS udos（約 5,000 行）と gcc（GNU Compiler Collection，約 90 万行）とを事例に予備実験を行った．予備実験の結果，ckTF/IDF 法によるコンセプトキーワードの計算速度は，gcc の場合，TF/IDF 法と比べ新規検索で約 6 倍，ファイル更新にともなう再計算で約 890 倍も高速だった．コンセプトキーワードの抽出の精度と再現率は，udos の場合で，それぞれ 57%と 26%だった．これは，我々のアプローチが識別子におけるコンセプトキーワードの抽出に向いていることを示している．今後の課題は，たとえば，ckTF/IDF 法を使って高速で精度の高いソースコード検索エンジンを開発することといった，コンセプトキーワードの応用である．

ckTF/IDF: a Method for Automatically Extracting Concept Keywords for Program Understanding

Masaru Ohba[†] and Katsuhiko Gondow[†]

We propose the Concept Keyword Term Frequency/Inverse Document Frequency (ckTF/IDF) method as a novel technique to efficiency mine *concept keywords* from identifiers in large software projects. ckTF/IDF is suitable for mining concept keywords, since the ckTF/IDF is more lightweight than the TF/IDF method, and the ckTF/IDF's heuristics is tuned for identifiers in programs. We then experimentally apply the ckTF/IDF to our educational operating system `udos` (consisting of around 5,000 lines in C code) and GNU C Compiler Collection (gcc, consisting of around 900,000 lines in C code), which produced promising results; By ckTF/IDF method, The gcc's source code was processed in 6 times faster than TF/IDF method at first time, and in 891 times faster than TF/IDF method at updating the source code. the `udos`'s source code was processed with an accuracy of around 57%. This preliminary result suggests that our approach is useful for mining concept keywords from identifiers, although we need more research and experience. For example, ckTF/IDF method can apply to fast source code search engine.

1. はじめに

多くのプログラマは，識別子に関数の役割を表す簡潔な名前をつけようと努力している．たとえば，教育用 OS `udos` [11] の FAT ファイルシステムの関数名には，単に `f()` ではなく，「dirent（directory entry）を読み込む」という役割がが分かるように `read_dirent()` と開発者は命名している．この開発者による命名の努力のおかげで，プログラムの読み手は識別子からプログラムの概要や振舞いを容易に理解することができる．

一般的に識別子は人が理解できる単語の集合で構成される．我々の経験では，識別子中の単語のすべてが重要であるわけでなはない．逆にいえば，特定の単語は他の単語に比べプログラム理解にとって非常に重要な意味を持つことが多い．たとえば `read_dirent()` 関数の場合，特に重要な単語は `dirent` である．なぜなら，`dirent` はディレクトリやファイルなどを表現するための基本的なデータ構造に付けられた名前で，FAT file system [22] の理解には欠かせない用語であるからである．我々は識別子に含まれる `dirent` のようなプ

† 東京工業大学大学院情報理工学研究科計算工学専攻
Department of Computer Science, Graduate School of Information Science and Engineering, Tokyo Institute of Technology

ログラム理解に役立つ語に注目した．我々は `dirent` ように重要な概念を表す語を**コンセプトキーワード**と呼ぶ．これまで我々はコンセプトキーワードの発見の方法とその利用法について研究をしてきた[25),26)]．

コンセプトキーワードは以下の 3 点でプログラム理解に貢献できる．

- コンセプトキーワードは，ソースコードの中で重要な部分を強調し，暗黙的な関係をみつけることができる．
  たとえば読み手が注目しているコンセプトキーワードをテキストエディタ上で強調表示☆することによって，読み手はすばやくソースコード中で重要な部分を発見することができる．FAT ファイルシステムのコンセプトキーワード `dirent` に読み手が注目すれば，ソースコード中で “/* `FAT12_read()` is used for sequential access to directory entries, while `read_dirent()` for random access */” といった重要なコメントを発見することができるうえに `read_dirent()` 関数と `FAT12_read()` 関数との暗黙的な関係をみつけることができる．`read_dirent()` 関数と `FAT12_read()` 関数との関係は，明確な呼び出し関係がないためクロスリファレンサやコールグラフでの発見は難しい．しかも，読み手がこのコメントを見逃すとこの暗黙的な関係が分からずプログラム理解に支障をきたす可能性がある．そのため `dirent` はプログラム理解に有用である．
- コンセプトキーワードはソースコードと仕様書との間の抽象度の違いによるギャップを埋めることができる．
  プログラムは細粒度の記述であり，そこから抽象的な概念や仕組みを理解することは難しい．抽象的な概念や目的を理解するには仕様書などの文書を読むと効果的に理解できる．しかし，プログラムと文書との対応関係は不明確であることが多い[7),21)]．もし開発者が識別子中のコンセプトキーワードに注目できれば，仕様書のどの部分に注目すればよいか分かることがある．たとえば，読み手がソースコード中の `read_dirent()` 関数を発見したときに，読み手がコンセプトキーワード `dirent` に注目できれば，FAT 仕様書[22)] から，`dirent`（directory entry）のデータ構造や概念について詳しく知ることができる．ただし，コンセプトキーワードが必ずしも仕様書とソースコードで一致しないこともある．たとえば “directory entry” を読む関数名が `read_folder_entry()` 関数だったときである．このような場合，ほとんどのプログラマは用語の違いを避け，補完しようと努力すると我々は仮定している．しかし，我々の経験では識別子のどれがコンセプトキーワードであるかどうかを判断することは難しい．なぜならば，大規模なソフトウェア開発では実装担当者以外はプログラムの詳細すべてを理解できないことが多いからである．したがって，なんらかのコンセプトキーワードを読み手に示す手法の提案が必要である．
- 識別子はソフトウェアリポジトリとの相性が良い．なぜならば，ソフトウェアリポジトリの一部であるバグトラッキングシステムやメーリングリストシステムは識別子と同様にテキストベース，マシン非依存などの共通の特徴を持つためである．つまり，コンセプトキーワードを発見する技術は大規模なソフトウェアリポジトリへ適用することが容易である．
  バージョン管理システム[6)] やメーリングリストやバグトラッキングシステム（BTS）[1)] で用いられるソフトウェアリポジトリの特徴は言語非依存，テキストベース，機械処理が可能なことである．コンセプトキーワードも同様の特徴を持つ．そのため，ソフトウェアリポジトリとの相性が良いといえる．

これまでにコールグラフ[12)]，クロスリファレンサ[28)]，スライサ[16)]，ソースコードブラウザ[17)]，コード整形ツール[10)]，デバッガ[9)]，ドキュメンテーションツール[20)]，プロファイラ[13)]，コードメトリクスツール[15)] といったプログラム理解ツールが数多く提案されてきた．しかし，既存の提案にはコンセプトキーワードの抽出をサポートするものは我々の知る限りほとんどない．本論文では識別子からコンセプトキーワードを発見するための新しい提案をする．

識別子からコンセプトキーワードを効率良く抽出することは難しい．それはコンセプトキーワードが識別子中のどこに含まれているのかを予測しにくいためと，識別子の数が多いため人手で行うことが難しいからである．したがって，識別子からコンセプトキーワードの発見に適したアルゴリズムが必要である．既存の TF/IDF 法[27)] のようなアルゴリズムではコンセプトキーワード抽出は十分ではない．それは，コンセプトキーワードと識別子との性質が自然言語とは大きく異

☆ たとえば，Emacs エディタの `highlight-regexp.el` で任意の単語を強調表示できる．

なるからである．たとえば，kbd_（keyboard の意）のように識別子の接頭辞はグルーピングの目的で使用されることが多い．自然言語ではこのようなことはほとんどない．TF/IDF 法ではこのようなコンセプトキーワードではない接頭辞も高得点になることが多く，不正確な結果になりやすい．TF/IDF 法では機能語辞書を使って不要語を削除し最適化が可能であるが，識別子用の機能語辞書は我々の知る限りない．

より問題なのは既存のアルゴリズムではコンセプトキーワードを発見するには処理が重すぎることである．現在のソフトウェアは大規模化かつ複雑化が進んでいる．たとえば，gcc（GNU Compiler Collection）は約 90 万行あり，ビルドだけでも 20 分以上☆かかる．さらに，現在のソフトウェア開発では時間の短縮化が重要視されている．そのため，実際の開発では効率の良いコンセプトキーワード抽出ツールが必要である．特に重要なコンセプトキーワード抽出ツールへの要求はソースコードの更新にともなうコンセプトキーワードの再計算の高速化である．新しい識別子を追加したときにコンセプトキーワードの抽出の高速化は開発者へインタラクティブにコンセプトキーワードを提示するために重要である．我々の提案は更新速度を高速化する工夫をしている．

本論文では，効率良くかつ正確にコンセプトキワードを発見するためのアルゴリズム ckTF/IDF 法（Concept Keyword Term Frequency and Inverted Document Frequency）を提案する．基本的なアイディアは以下のとおりである．

- ckTF/IDF 法は，TF/IDF 法の評価値を得るための非常に軽量なアルゴリズムである．特に更新時は TF/IDF 法に比べはるかに高速である．
- ckTF/IDF 法は，識別子からコンセプトキーワードを抽出することに特化している．たとえば，正確のために不必要な識別子の接頭辞を排除する．

我々は ckTF/IDF 法の実用性を実際に確認するために，実験的に我々が開発した教育用 OS udos [11]（C 言語で約 5,000 行）と gcc（GNU Compiler Collection）に適用し ckTF/IDF の予備評価を行った．その結果，ファイルを追加したときの更新時間は gcc，udos ともに 0.1 秒と高速に抽出できた．ckTF/IDF 法によるコンセプトキーワードの抽出は，udos の場合で 57%の精度，26%の再現率で抽出できた．

本論文の構成は以下のとおりである．2 章ではコンセプトキーワードの特徴について述べる．3 章では ckTF/IDF 法の提案と概説を行い，4 章で ckTF/IDF 法を実装するためのフレームワーク Identifier Exploratory Framework（IEF）について述べる．5 章で ckTF/IDF 法を gcc と udos に適用した事例を述べ，6 章で関連研究，7 章で議論，8 章で結論と今後の課題について述べる．

2. コンセプトキーワード

本章ではコンセプトキーワードの特徴と有用性を述べる．コンセプトキーワードの抽出の難しさもあわせて述べる．

2.1 コンセプトキーワードとは

1 章では，コンセプトキーワードはプログラム理解のために重要な概念を表す語であると述べた．たとえば，dirent（directory entry）は FAT ファイルシステムの概念を表すコンセプトキーワードである．しかし，コンセプトキーワードかどうかは主観的な判断がどうしても必要になるため厳密な定義は難しい．そこで，議論を明確にするためにコンセプトキーワードを 3 つに分類して議論する．

- **理想的コンセプトキーワード**：客観的な尺度でプログラム理解に役立つと証明されたコンセプトキーワード．
- **人為選択コンセプトキーワード**：開発者や読み手が理想的であると信じるコンセプトキーワード．
- **機械抽出コンセプトキーワード**：ckTF/IDF 法などを用いて機械的に抽出したコンセプトキーワード．

たとえば，PTE は人為選択コンセプトキーワードである．PTE（Page Table Entry）はページングを実現するためのデータ構造である．PTE は，ckTF/IDF 法の発見的手法と合致しないので，ckTF/IDF 法では発見が難しく人手でしか発見できない．そのため PTE は機械抽出コンセプトキーワードではない．一方，dirent も人為選択コンセプトキーワードであるが，ckTF/IDF 法でも発見が可能である点で機械抽出コンセプトキーワードでもある．また，機械抽出キーワードでも人為抽出コンセプトキーワードではない単語もある．たとえば FAT12 は概念の粒度が大きすぎるため，人為コンセプトキーワードではないと我々は分類しているが，ckTF/IDF 法では機械抽出コンセプトキーワードとして出力される．また，理想的コンセプトキーワードは今のところ客観的な尺度がない．そのため，PTE は理想的コンセプトキーワードであると我々は信じていはいるが，今のところそれを証明する

☆ GCC-4.1.1 Pentium 4-ht 2 GHz 1 GB RAM, Linux-2.6.8-1 でのビルド時間．

表 1 udos での人によって選択されたコンセプトキーワードと他の種類の一覧
Table 1 Human-selected concept keywords and other category words in udos.

	category	#	examples	description
C.	concept keywords	61	dirent, root, PTE, tss, path, signal, yield	helpful key concepts for program understanding
G.	grouping words	18	kbd_, vga_, FAT12_, sys_, FDC_, RTC_, console_, _H, _t	prefixes and suffixes for grouping functions and variables, or for other purposes
A.	attributes, and less important concepts	70	busy, byte, offset, name, memory, end, int8, again	general nouns and adjectives used as attributes, modifiers, etc., being less informative in themselves.
V.	generic verbs	130	read, set, is, move, wait, print, dump, make, init	generic verbs to describe actions or operations; the same names are commonly used for unrelated functions

ことはできない．そこで本論文では，客観的尺度がないため理想的コンセプトキーワードを対象とはせず，人為選択コンセプトキーワードを再現するような機械抽出コンセプトキーワードの低コストな抽出手法を提案する．

表 1 に，udos の実験で得られた開発者による人為選択コンセプトキーワードとその他の語の一覧を示す．本論文ではこれ以降，断わりなくコンセプトキーワードと使った場合は，人為選択コンセプトキーワードを意味する．

コンセプトキーワードは必ずしも識別子だけにあるとは限らない．しかし，多くのコンセプトキーワードは識別子の一部として出現すると我々は考えている．たとえば，文献 18) では，識別子には情報を含み，簡潔で憶えやすい名前にしなければならないと主張がある．また，識別子の命名には規則性を持たせることが重要であるとも主張している．このことから，我々は識別子からであれば，機械的に効率良くコンセプトキーワードを抽出できると考えたのが ckTF/IDF 法提案のモチベーションである．実際に udos から人手で抽出したコンセプトキーワード（表 1 より）は，識別子中のすべての語彙（279 語）のうち約 22%（=61/279 語）であった．本論文では，識別子中にあるコンセプトキーワードの抽出に注目する．

5 章では ckTF/IDF 法による機械抽出コンセプトキーワードと表 1 で示した人為選択コンセプトキーワードとを比較して精度と再現率の予備評価を行う．

2.2 コンセプトキーワードの有用性

本章では，識別子から抽出したコンセプトキーワードの利点について述べる．

プログラム理解はソフトウェア開発で重要な作業の 1 つである．たとえば文献 14) では，“全体のライフサイクルにかかるコストのうち 30〜35%がプログラム理解に費やされている” と報告している．したがってプログラム理解はソフトウェア工学上，重要な問題である．

この問題を解決するために，1 章で述べたとおり，コールグラフ抽出器やクロスリファレンサなど多くのプログラム理解ツールが提案されてきたが，プログラム理解コストの低減についてはまだ十分ではない．

その理由は以下の 2 つである．1 つ目の理由はツールの正確性である．たとえば，文献 23) によればソースが同じでもコールグラフ生成ツールによって異なった結果になってしまうことがある．2 つ目の理由はツールの制限が大きい．たとえば，理想的なコールグラフ抽出器でもプログラム理解に必要なすべての情報を抽出することはできない．つまり，コンセプトキーワードのような新しい情報を提供するツールの開発が必要である．

コンセプトキーワードはプログラム理解のための新しい情報を提供することができる．1 章で言及したように，コンセプトキーワードはプログラム理解を様々な側面からサポートすることができる．さらに，バージョン管理システムやバグトラッキングシステムなどの既存のソフトウェアリポジトリと容易に組み合わせることができる．それは，これらのソフトウェアリポジトリとコンセプトキーワードの特徴が言語非依存，テキストベース，機械処理可能という点で同じだからである．我々が提案するコンセプトキーワードはプログラム理解の向上に非常に有用であると考える．しかし，ソースコードからコンセプトキーワードを発見する研究は我々の知る限りいままでほとんど行われてい

ない．

2.3 コンセプトキーワードの発見の難しさ

TF/IDF 法[27] は文書中から特徴語を発見するアルゴリズムである（詳細は 3.1 節を参照）．1 章で述べたように，TF/IDF 法のような既存の技術では識別子を扱うことが難しい．なぜならば，識別子とコンセプトキーワードは自然言語と異なる特徴を持つうえに，TF/IDF 法はコンセプトキーワードの発見には処理が重すぎるからである．

この問題を解決するために我々は新しいアプローチである ckTF/IDF 法を提案する．ckTF/IDF 法と TF/IDF 法との違いは 3.3 節で説明する．

3. ckTF/IDF 法

本章では，ckTF/IDF（Concdpt Keyword Term Frequency and Inverted Document Frequency）法の有効性を示す．

3.1 TF/IDF 法の概説

ckTF/IDF 法を定義する前に，本節では ckTF/IDF 法の基盤になる TF/IDF 法の概説を行う．TF/IDF 法は，索引語から文書を特徴づける語（=特徴語）を見つけるための計算手法である．TF/IDF 法の基本的なアイディアは，特定の文書のみで多く出現する単語に高得点（重み）を与え，多く文書に出現すれば得点（重み）を下げることで特徴語をみつける．得点の計算は，1 つの文書における単語の出現頻度 *tf*（Term Frequency）と計算対象の文書集合 D の文書数 N と単語 t（ただし $t \in D$ 中にある単語）が出現する文書数の比の対数 *idf*（Inverse Document Frequency）の 2 つの評価値から計算する．直感的にいえば，*tf* は単語 t が文書内 d で高頻度で出現するときに高得点を与える．*idf* は，文書集合中で局所的に出現する単語に高得点を与える．つまり，*idf* は，単語 t がすべての文書に出現する場合に最も低い得点となり，逆に単語が 1 つの文書のみに出現するときに最大となる．文書数 N と *df* の比の対数をとるのは，文書集合の規模に対して *idf* の値の変化を小さくするためである[29]．*idf* は以下のように定義する：

$$idf(t) = \log \frac{N}{df(t)} \tag{1}$$

このとき $df(t)$ は単語 t が含まれる文書の数である．ただし $df(t)$ は 0 にはならない．N は文書の総数である．

文書 d における単語 t の得点 $w(t,d)$ は，以下のように定義する：

$$w(t,d) = tf(t,d) \cdot idf(t) \tag{2}$$

このとき $tf(t,d)$ は d の単語 t の出現頻度である．

3.2 ckTF/IDF 法の定義

ckTF/IDF 法の概要は以下の 2 点である．

- 3.1 節で定義した $tf(t,d)$ と $idf(t)$ を量子化し得点計算を単純化することによって，高速な処理を実現する．
- 識別子特有の接頭辞や接尾辞を取り除く（オプションで適用するかどうかを指定できる）．

ckTF/IDF 法での $idf(t)$ は以下のように定義する：

$$idf(t) = \begin{cases} 1 & \text{if } 1 \leq df(t) \leq n \text{ and } \neg \text{is\_prefix}(t) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

ここで，$df(t)$ は TF/IDF 法の定義と同じである．n（≥ 1）は境界条件（デフォルトは 1）である．$tf(t,d)$ と $idf(t)$ は TF/IDF 法の定義を 0 か 1 に量子化し単純化したものに対応する．$tf(t,d)$ と $idf(t)$ を量子化することによって，ソースコードの更新にともなう再計算のときに追加された単語についてだけ計算すればよくなり高速化できる．一方，TF/IDF 法でも $df(t)$ に閾値を設け得点計算を省力化できるが，閾値を超えないすべての単語について再計算が必要であり ckTF/IDF 法よりも遅くなる．is_prefix(t) は単語 t が接頭辞だったときのみ真になる述語である．接頭辞は識別子中で先頭に 2 回現れる単語と本論文では定義する．

ckTF/IDF 法における $tf(t)$ と単語の得点 $w(d,t)$ は以下のように定義する．

$$tf(t) = \begin{cases} 1 & \text{if } \exists d, tf(t,d) > 1 \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

$$w(t,d) = tf(t,d) \cdot idf(t) \tag{5}$$

ckTF/IDF 法では，$w(d,t) = 1$ のとき単語 t は（機械抽出による）コンセプトキーワードとなる．

以下に ckTF/IDF 法のアルゴリズムを示す．ckTF/IDF 法では globally frequent flag（以下 *global* と省略する）と locally frequent flag（以下 *local* と省略する）の 2 つのフラグを用いて計算すると簡単に計算ができる．$local(t,d)$ は文書 d 中の単語 t ののべ数が 1 つより多いときに 1 になる．$global(t)$ は全体の文書集合中の単語 t が n 個以上の文書中に存在するときに 1 になる．

3.2.1 文書集合から単語の収集するときの処理

1. 文書集合 D から文書 d を取り出す．
   - 1-1. 文書 d から単語 t を取り出す．
   - 1-2. $local(t,d)$ について
     - 1-2-1. $local(t,d) =$ null のとき，*local*

$(t,d) \leftarrow 0$ に初期化する．

1-2-2. $local(t,d) = 0$ のとき，$local(t,d) \leftarrow 1$ にする．

1-3. もしも単語 t が文書 d 以外でも出現していたら $global(t) \leftarrow 1$ にする．そうでなければ $global(t) \leftarrow 0$ とする．

1-4. 文書 d が空でなければ 1-1. を繰り返す．

2. 文書集合 D が空でなければ 1. を繰り返す．

3.2.2 得点の計算処理

1. データベース (t) 中から単語 t を取り出す．

1-1 単語 t について，

(i) $global(t) = \text{null}$ かつ $local(t,d) = 1$ かつ $\text{is\_prefix}(t)$ であれば，$w(t) = 1$ を出力する．

(ii) 上記以外は $w(t) = 0$ と出力する．

2. データベース空でなければ 1. を繰り返す．

ここでいう単語は識別子中の区切り文字で分割したものをいう．本論文でいう「区切り文字」とは，識別子中にあるコーディング規約上で指定された特定の文字をいう．我々の経験では，多くの開発者が識別子を決定するときに単語の境界をはっきりさせるために区切り文字を使っていることが多い．4 章で述べる IEF の実装では区切り文字をアンダースコア（_）と定義する．

3.2.3 接頭辞の除去

ckTF/IDF 法では識別子から接頭辞をコンセプトキーワードの候補から排除するペナルティを与える．これは我々の経験から接頭辞はプログラムの詳細の理解にあまり有用ではなかったからである．たとえば，我々が `udos` の FAT ファイルシステムの詳細を理解しようとしたとき，関数名 FAT_read_write() の FAT は接頭辞であるが，FAT という言葉を知ったところで FAT の仕組みは見えてこない．そのため FAT はコンセプトキーワードではない．一方，read_dirent() 関数の dirent は FAT を理解するうえで重要な概念でプログラム理解に有用であるためコンセプトキーワードである．この dirent は接頭辞ではない．

ただし，接頭辞を機械的に発見することは一般的に難しい．なぜなら，接頭辞は開発者によって決められコンパイラなどによるチェックが乏しい．そのため，開発者が接頭辞として決めた単語が必ず識別子の先頭に現れるというわけではない．したがって，機械的に接頭辞を発見するために本論文では「単語 t が識別子の先頭に 2 回以上出現したら接頭辞とする」．また，単語 t が接頭辞かどうかを判断する述語として $\text{is\_prefix}(t)$ を導入する．$\text{is\_prefix}(t)$ は単語 t が接頭辞であるときに真となり，それ以外は偽となる．

3.3 ckTF/IDF 法 vs. TF/IDF 法

3.3.1 ckTF/IDF の特徴

ほとんどの場合，TF/IDF 法で $\max_d w(t,d)$ が高得点になる場合，ckTF/IDF 法でも同様に $w(t) = 1$ になる．逆に，TF/IDF 法の $\max_d w(t,d)$ が低い得点のときは $w(t) = 0$ になる．ただし以下の 2 つの例外がある．

- ckTF/IDF 法は対象の単語が接頭辞だった場合，ペナルティが課せられる．ペナルティとは ckTF/IDF 法では単語が接頭辞であった場合，コンセプトキーワードの候補にならないようにする．接頭辞は機械的に文書集合中で 2 回以上識別子の先頭に現れた単語と近似して発見する．TF/IDF 法にはない．たとえば，キーボードを意味する `kbd_` のような接頭辞は我々の経験では不必要なものが多い．そのため，ckTF/IDF 法では接頭辞を排除する．
- 2 つのフラグが両方ともセットされたとき，ckTF/IDF 法のスコア（$w(t)$）は 0 になるが，TF/IDF 法では，スコアは様々な値をとる．これは，$tf(t,d)$ が非常に大きい場合，もしくは，$df(t)$ が 1 以上でかつ比較的小さい値をとるとき，ckTF/IDF は不正確な値を示してしまうためである．しかし，この問題は式 (3) と式 (4) の境界値 n を変更することによって軽減することができる．

3.3.2 計算量の比較

すべての $\max_d w(t,d)$ を計算するためには，ckTF/IDF 法と TF/IDF 法はともに $O(|D| + |T|)$ の計算量が必要である．このとき，T はすべての単語の集合を表し，D はすべての文書（ソースファイル）を意味する．

しかし，文書を新たに追加するときに ckTF/IDF 法は TF/IDF 法よりも高速に計算することができる（詳細は 5 章を参照）．ここで，ΔT の単語の集合を含む，加する新しい文書集合を ΔD とすると，ckTF/IDF 法の計算量 $O(|\Delta D| + |\Delta T|)$ で TF/IDF 法は $O(|D + \Delta D| + |T + \Delta T|)$ となる．つまり，この場合では ckTF/IDF 法の方が少ない計算量になる．

4. 設計と実装

本章では，IEF（Identifier Exploratory Framework）の設計と実装を概説する．IEF は，ckTF/IDF 法を実現するために，我々が実験的に開発したフレームワークである（IEF のソースは文献 24) で公開）．IEF の概観を**図 1** に示す．IEF の GUI のスクリーン

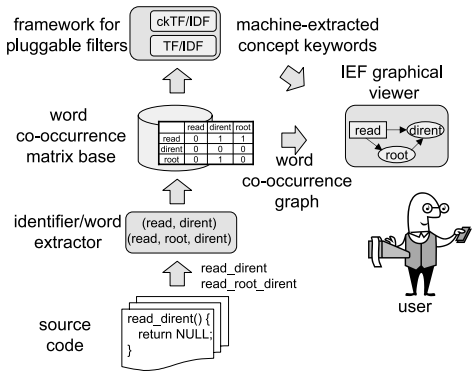


図 1 Identifier Exploratory Framework (IEF)
Fig. 1 Components of Identipier Exploratory Framework (IEF).

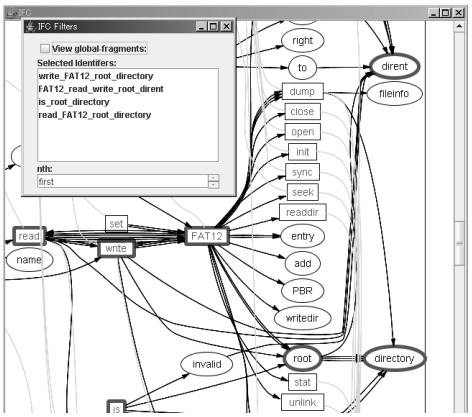


図 2 IEF グラフィカルビューアの実行画面
Fig. 2 Screen snapshot of IEF graphical viewer.

ショットを**図 2** に示す．図 2 は `udos` の `fat12.c` 中の識別子中の単語間の共起関係を示している．我々は図 2 から `udos` の識別子中の単語間の共起関係をグラフ化し調査することで，コンセプトキーワードの出現パターンと共起関係に関連があることを発見した．

IEF の構成は以下のとおりである．

- *Identifier/word extractor*：クロスリファレンサ GNU GLOBAL [28] の `gctags` コマンドを用いてソースコードから関数の定義を抽出し，それと同時に単語に分解する．高速化のために関数呼び出しや変数の使用に関しては抽出しない．我々の経験から，多くの開発者は識別子中の区り文字としてアンダースコアや CamelCase を使用することが多い．IEF では，この性質を利用して単語の抽出を行う．区切り文字はユーザによって選択することが可能である．
- *Word co-occurrence matrix base*（単語コーパス）：識別子から抽出したすべての単語と識別子中の単語の共起関係とをファイル別に保存した単純なデータベースである．
- *Framework for pluggable filters*：ckTF/IDF 法を実装するための IEF で最も中心的役割を担うインタフェースである．IEF のユーザはこれを使って単語コーパスにアクセスし ckTF/IDF 法や TF/IDF 法の実装を行うことができる．IEF では，この性質を利用して単語の抽出を行う．IEF では区切り文字をアンダースコア（_）と定義する．この区切り文字による単語抽出は完全ではないが，開発者がわざと複数の単語を特別な意味で使うことに利用する場合もあるので区切り文字のない複数の単語は 1 つの単語として扱う．たとえば，`udos` にある `kmalloc`（“kernel memory allocation” の意）はこれ以上分割できないが，開発者がユーザレベルの `malloc` とカーネルレベルの `kmalloc` とを明示的に分けたいという意図が含まれている．
  TF/IDF 法を実装するために IEF では本来 ckTF/IDF 法には必要のない単語の数えあげのためのインタフェースなどを用意してある．しかし，ckTF/IDF 法でも TF/IDF 法でも IEF のデータベースから得た情報から得点計算のための表（e.g., $local(t, d)$）を作成するので，得点計算の速度の比較は可能である．
  現在，我々はこのインタフェースを使って ckTF/IDF 法を Ruby 言語でわずか約 30 行で実装している．
- *IEF graphical viewer*：読み手が単語の共起関係とフィルタの出力の結果をあわせてグラフィカルに読み手に示す（図 2 参照）．我々は IEF graphical viewer は Grappa グラフ描画ライブラリ [8] を使って約 500 行の Java 言語で実装した．
  図 2 は `udos` の `fat12.c` の共起関係を IEF graphical viewer で表示したものである．図 2 では，単語 t が $global(t) = 1$ だったときにその単語を四角の枠で表示している．楕円で囲まれている単語 t は $global(t) = 0$ となる単語である．黒い線で結ばれている単語は同一識別子中の共起関係を表したものである．このグラフによって，読み手は人為選択コンセプトキーワードの発見をしやすくなる．たとえば図中では `dirent` や `root` といった単語は多くの線が集中し，ローカルな利用しかない単語であることが分かる．

表 2 精度と再現率との計算に必要な要素（z は利用しない）
Table 2 Elements for precision and recall (z is not used in the calculation).

	ckTF/IDF 法で発見できた	ckTF/IDF 法で発見できなかった
コンセプトキーワード	$w = 16$ (true positive)	$x = 45$ (false negative)
コンセプトキーワードではない	$y = 12$ (false positive)	z (true negative)

5. 予 備 実 験

実際に ckTF/IDF 法が有効であるかことを確かめるために，我々は実験的に ckTF/IDF 法と TF/IDF 法を udos と gcc のソースコードに適用した．本章では，その結果について述べる．

5.1 ckTF/IDF 法の精度と再現率

5.1.1 教育用 OS udos からのコンセプトキーワードの抽出結果

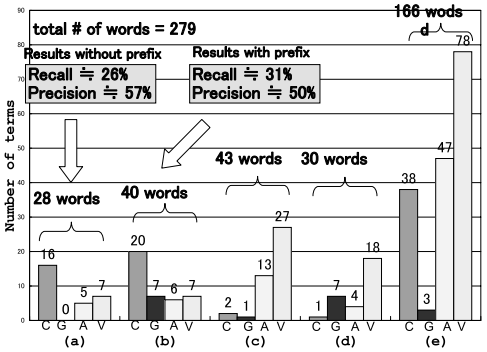
3.2 節では ckTF/IDF 法を計算するときに *global* と *local* の 2 つのフラグを使って計算すると述べた．**図 3** は ckTF/IDF 法で処理した単語で *global* と *local* のそれぞれの組み合わせた内容を分析した結果を示している．図 3 中の分類の概要は以下のとおりある．

(a) 機械抽出コンセプトキーワードとして抽出した場合（接頭辞を除いたとき）
(b) 機械抽出コンセプトキーワードとして抽出した場合（接頭辞を除かなかったとき）
(c) 機械抽出コンセプトキーワードではないと判断された単語．1 つの文書中で複数回出現している単語．
(d) 機械抽出コンセプトキーワードではないと判断された単語．複数の文書で出現し，かつ各文書で複数回出現している単語．
(e) 機械抽出コンセプトキーワードではないと判断された単語．各文書で 1 度のみ出現した単語．globally と locally フラグの両方がついていない単語．

ckTF/IDF 法で (a)〜(e) までの分類をさらに人手で C，G，A，V の 4 つに分類して，ckTF/IDF 法の定義が本当にコンセプトキーワードを抽出できるかを分析した（C，G，A，V の分類は意味は表 1 を参照）．

5.1.2 精度と再現率の導入

機械抽出のコンセプトキーワードを評価するために，指標として精度と再現率を導入する（**表 2** もあわせて参照）．精度は $Precision = \frac{w}{w+y} = \frac{C_r}{C_m}$ で表し，再現率は $Recall = \frac{w}{w+x} = \frac{C_r}{C_h}$ である．C_m は機械抽出コンセプトキーワードの総数（$w+y = 28$）に該当（図 3 の (a) の C，G，A，V の総数）し，C_r は機械抽出コンセプトキーワードのうち正解の個数（$w = 16$）である（図 3 の (a) の C に該当する）．C_h は人為抽出コンセプトキーワードの総数（$w + x = 61$）である（図 3 の (b)〜(e) の C の総数）．表 2 でコンセプトキーワードかどうかの判断には人為選択コンセプトキーワードを正解とした．人為選択コンセプトキーワードの列挙は udos の開発者に依頼した．



Legend: C=concept keywords, G=grouping words, A=attributes, V=verb,
(a)= $\neg global(t) \wedge local(t) \wedge \neg$is_prefix$(t)$
(b)= $\neg global(t) \wedge local(t)$, (c)= $global(t) \wedge \neg local(t)$
(d)= $global(t) \wedge local(t)$, (e)= $\neg global(t) \wedge \neg local(t)$

図 3 udos での ckTF/IDF の精度と再現率結果
Fig. 3 Precision and recall of ckTF/IDF for udos.

5.1.3 udos の精度と再現率の結果

人為選択と機械抽出のコンセプトキーワードを比較した結果，ckTF/IDF 法（図 3 の (a)）の精度と再現率はそれぞれ約 57%（= 16/28）と 26%（=16/61）であった．一方，TF/IDF 法の精度と再現率はそれぞれ約 28%（= 8/28）と約 13%（= 8/61）と低い値であった☆．udos の場合では，ckTF/IDF 法は精度と再現率の両方に関して TF/IDF 法よりも高いことが分かった．ckTF/IDF 法と TF/IDF 法の精度と再現率の有意差については 7.2 節で述べる．

また，識別子の接頭辞を除かなかった場合では（図 3 の (b)）では，再現率が 5 ポイント上ったが精度が 7 ポイント下がった．

5.1.4 gcc からのコンセプトキーワードの抽出結果

ckTF/IDF 法が大規模なソフトウェアにも適用できることを示すために gcc（GNU Compiler Collection）に対して人為コンセプトキーワードと ckTF/IDF 法による機械抽出コンセプトキーワードとの比較を行った．

☆ TF/IDF 法で重み付けされた単語の上位 61 単語との比較

表 3 udos での単語の種類と出現位置別の集計結果
Table 3 Total # of word occrences by position for udos.

	category	word position		
		first	last	middle
1.	concept keywords	*14*	21	75
2.	grouping words	*141*	20	33
3.	attributes	78	17	199
4.	generic verbs	23	38	41
	total	*256*	96	348

その結果，精度と再現率はそれぞれ約 22%（= 38/176）と約 28%（= 38/136）であった．一方，TF/IDF 法で同様に適用した結果，精度と再現率は約 9%（= 15/176）と約 11%（= 15/136）であった☆．以上の結果から gcc の一部（約 3 万行クラスの規模）でも ckTF/IDF 法を適用できることを確認した．

gcc のコンセプトキーワード抽出には，本研究室で gcc を対象に実装実験を行っているソフトウェア工学研究者の協力を得て行った．コンセプトキーワードは，被験者が実際にコードの理解を行ったソースコード（9 ファイルで約 3 万行）を抽出対象とした．被験者が選択したコンセプトキーワードは 136 語あった．一方，ckTF/IDF 法による機械抽出コンセプトキーワードは 176 語あり，そのうち正解は 38 語であった．

5.2 ckTF/IDF 法における，udos での接頭辞を取り除いたときの精度

3.3.1 項では，ckTF/IDF 法では接頭辞となっている単語に対し，得点上ペナルティを課すと説明した．

表 3 は udos の識別子中で単語が出現した位置（先頭，最後，それ以外）の統計である．接頭辞の機械的発見は難しいが，簡単に接頭辞の発見を行うために文書集合 D 中で識別子中の先頭に 2 回以上現れた単語を接頭辞とし，識別子中の最後に 2 回以上現れた単語を接尾辞とした．表 3 から udos のコンセプトキーワードが識別子の先頭部分に出現することがほとんどない（約 5% = 14/256）ことが分かる．逆にグループ化のために使われている単語は識別子の先頭部分に多く出現している（約 55% = 141/256）ことが分かる．そのため，接頭辞となる語を機械抽出コンセプトキーワードの候補から削除することで精度と再現率を向上させることができると我々は期待した．

結果として接頭辞の削除は精度と再現率の向上は必ずしも貢献するわけではないことが分かった．図 3 の項目 (a) と (b) がこの結果を示している．(a) は接頭辞（先頭に出現する単語）をコンセプトキーワードから排除した結果である．(b) は接頭辞も含めた結果である．接頭辞を排除した場合，精度は 57%（+7 ポイント）に向上する，しかし再現率は 26%（−5 ポイント）に減少する．

☆ ただし，udos の実験と同様に ckTF/IDF 法が抽出した 136 語と単語数を合わせるために，TF/IDF 法の上位から 136 位の単語から再現率と精度を計算した．

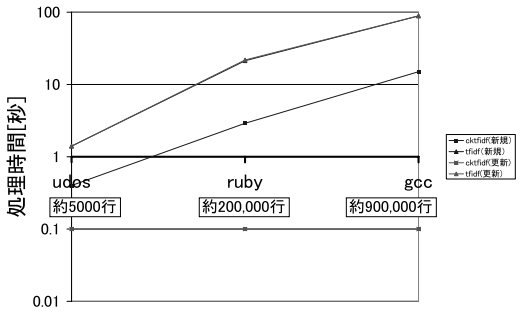


図 4 ckTF/IDF 法と TF/IDF 法を gcc，ruby インタプリタ，udos に適用したときの処理時間の比較
Fig. 4 Execution speeds of ckTF/IDF and TF/IDF for gcc, ruby interpreter, udos (in elapsed time).

5.3 ckTF/IDF 法と TF/IDF 法のパフォーマンス

ckTF/IDF 法が高速に処理できることを示すために，gcc（バージョン 4.1.1），udos, ruby インタプリタ（バージョン 1.8.5）のソースコード全体☆☆についてコンセプトキーワードの機械抽出を行い計算速度を比較した．その結果，開発者が開発をしながら高速に機械抽出コンセプトキーワードを得るのに適しているのは ckTF/IDF 法であることが分かった．

図 4 は TF/IDF 法と ckTF/IDF 法との得点計算の時間の比較をしたグラフである．ただし，単語コーパスの作成時間は含まない（ckTF/IDF 法と TF/IDF 法の両方とも同じ単語コーパスを使うため）．図 4 中の“(更新)”と書かれている凡例は 1 度コンセプトキーワードを機械抽出したあとで，単語コーパスに新しく 1 つのファイル（2 単語を含む）を追加したときの再計算に要した時間である．図 4 から，TF/IDF 法は更新時と新規時とでほとんど同じであることが分かる．一方，ckTF/IDF 法はすべての対象で約 0.1 秒と高速あった．更新時，TF/IDF 法と ckTF/IDF 法では約 890 倍（gcc で比較）も ckTF/IDF 法が高速であることが分かる．ckTF/IDF 法と TF/IDF 法とが共通に使う単語コーパスの作成には udos で約 3 秒，gcc では約 504 秒かかった．

6. 関 連 研 究

現在，我々の知る限り識別子からコンセプトキー

☆☆ gcc についてはソースディレクトリの gcc 以下を対象とした．

ワードを発見する研究はみつかっていない．

TF/IDF 法[29] は，文書を特徴づける索引語をみつけるための古典的な重みづけを機械的に行うためのアルゴリズムである．TF/IDF 法は *tf* と *idf* という 2 つの尺度を使って計算を行う．*tf* は文書 d 中の単語 t の出現頻度を表す．*idf* は単語 t が出現する文書数と全体の文書数の比の対数である．TF/IDF 法では *idf* を計算するために全体の文書数を使っているため，新しい文書を文書集合に追加したときにすべての単語について *idf* の再計算が必要になる．一方 ckTF/IDF 法の *idf* の計算では全体の文書数を利用しないので追加文の文書だけを計算するだけで済む点が異なる．

Caprile ら[4),5)] は識別子再構築ツールを提案している．このツールは識別子を機械処理するために標準的な文法を強制することで識別子の再構築を半自動処理で行う．彼らはプログラム理解に識別子は重要であると考えている点で我々の考えに近い．しかし，彼らの研究はコンセプトキーワードの抽出のための研究ではない．

Anquetil[2)] は識別子とプログラム中のコメントから手動で概念の抽出を試みている．彼はその手法を Mosaic システムや X Window System の `xm` と `xmx` に適用している．コメントや識別子にプログラム理解に役立つ概念があることを指摘している点が我々に近い．しかし，概念の抽出に手動で約 30 時間と非常に長い時間がかかっている点が我々とは異なる．ckTF/IDF 法なら約 90 万行の gcc を約 15 秒でコンセプトキーワードを自動で計算することができる．

Anquetil ら[3)] は，省略したファイル名（たとえば “dbg” は debug の意）から概念の抽出を行う技術を提案している．彼らは英語の辞書を用いて 80%から 85%の高精度で抽出を実現した．我々の注目するところはファイル名ではなくより詳細な情報のある識別子である点が異なる．一般的な英語の辞書でもある程度不要語を削除できる可能性があるが，必ずしもコンセプトキーワードを効率良く抽出できるとは限らないと我々は考える．我々は識別子に特化した辞書が必要であると考える（辞書の作成については我々の今後の課題である）．

Knuth[19)] は，ソースコードと文書を `WEB` 言語を用いて統合し，よりよいプログラム理解を実現しようと試みている．`WEB` 言語とはアプローチは異なるが，コンセプトキーワードがコードと抽象度の異なる文書を暗黙的に関連づけることができる点で近い．コンセプトキーワードは文書を検索するためのキーワードとして利用することが可能である．我々はコンセプトキーワードの応用として文献 26) で文書とソースコード間の追跡性を整理分類するためにコンセプトキーワードを導入している．我々の提案は gcc などの大規模なソフトウェアを対象としているが，`WEB` 言語は教育に重点を置いている点が異なる．我々の知る限り，gcc などの大規模ソフトの多くは `WEB` 言語では記述されていない．ckTF/IDF 法なら大規模ソフトでもコンセプトキーワードの抽出が可能である．

7. 議　　論

7.1 閾値の意味

本章では 3.2 節で述べた ckTF/IDF 法の定義の閾値 n についての意味とデフォルト値が 1 である理由を以下に述べる．

- ckTF/IDF 法の目的はユーザがインタラクティブに利用できる軽量なものを提案したいというモチベーションで我々が開発した（3.2 節参照）．$n = 1$ とすることで，以下の点で軽量化できる．
- 記憶領域を削減できる．*global* は 2 状態（0/1），*local*（null/0/1）は 3 状態を保持すればよい．保持する w の値は $n = 1$ なら 1 次元配列で表現できるため，$n > 1$ よりも記憶領域が少なくて済む（$n > 1$ は 2 次元配列になってしまう）．これは，高速計算のためにオンメモリで処理することが重要だと我々は考える．そのため記憶領域の削減は高速化に重要である．
- 処理を高速化できる．*global* と *local* ともに四則演算をしなくてよく，論理積 ¬globally ∧ locally で計算できるため計算が高速である．

7.2 udos と gcc の精度と再現率における ckTF/IDF 法の有用性

5 章では udos と gcc を対象に ckTF/IDF 法と TF/IDF 法の精度と再現率で比較した．本節では

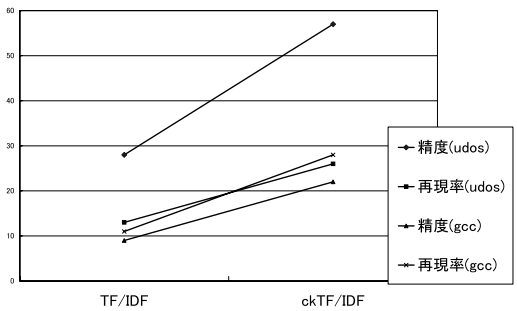


図 5 ckTF/IDF 法と TF/IDF 法を gcc と udos に適用したときの精度と再現率の比較．縦軸の単位はそれぞれパーセント（%）

Fig. 5 Precision and recall of ckTF/IDF and TF/IDF for gcc, ruby interpreter, `udos`.

TF/IDF 法に対する ckTF/IDF 法の優位性の有無について述べる．

udos と gcc における精度と再現率の比較表を**図 5** に示す．図 5 から分かるとおり，ckTF/IDF 法の方が TF/IDF 法よりも優れている．しかし，ckTF/IDF 法は接頭辞を取り去るという処理以外は TF/IDF 法を量子化したものとほとんど変わらないと考える．したがって，ckTF/IDF 法の実験結果が優位であるとはつねにはいえないと我々は考える．ただし，ckTF/IDF 法の方が実行速度の点では優れていることは変わらない．

8. 結　　論

我々は大規模なソフトウェアでの**コンセプトキーワード**を効率良く抽出するための新しい技術 ckTF/IDF（Concept Keyword Term Frequency/Inverse Document Frequency）法を提案した．予備実験の結果，ckTF/IDF 法によるコンセプトキーワードの計算速度は，gcc の場合，TF/IDF 法と比べ新規検索で約 6 倍，ファイル更新にともなう再計算で約 890 倍も高速だった．これは TF/IDF 法にはない特徴である．コンセプトキーワードの抽出の精度と再現率は，`udos` の場合で，それぞれ 57%と 26%だった．この結果は我々のアプローチがコンセプトキーワードの抽出を助けることを示している．

本研究の今後の展開を以下に示す．(1) ckTF/IDF 法を利用した軽量なソースコード検索システムの提案をする．(2) コンセプトキーワードをもとに識別子命名サポートツールの提案をする．(3) 厳密な再現率の計算は一般的に難しい．これをある程度解決するためにコンセプトキーワードの再現率を計測するためにテストコレクションを作成する．(4) より正確なコンセプトキーワード抽出のために，識別子に特化した不要語辞書を作成する．

参 考 文 献

1) Two case studies of open source software development: Apache and Mozilla, *ACM Trans. Softw. Eng. Methodol.*, Vol.11, No.3, pp.309–346 (2002).
2) Anquetil, N.: Characterizing the Informal Knowledge Contained in Systems., *WCRE: Proc. 8th Working Conf. on Reverse Engineering*, pp.166–175 (2001).
3) Anquetil, N. and Lethbridge, T.: Extracting concepts from file names: a new file clustering criterion, *ICSE '98: Proc. 20th Int. Conf. on Software Engineering*, pp.84–93, IEEE Computer Society (1998).
4) Caprile, B. and Tonella, P.: Nomen Est Omen: Analyzing the Language of Function Identifiers, *WCRE '99: Proc. 6th Working Conf. on Reverse Engineering*, p.112, IEEE Computer Society (1999).
5) Caprile, B. and Tonella, P.: Restructuring Program Identifier Names., *ICSM: Int. Conf. on Software Maintenance*, pp.97–107 (2000).
6) Collins-Sussman, B.: The subversion project: buiding a better CVS, *Linux J.*, Vol.2002, No.94, p.3 (2002).
7) Egyed, A.: Resolving uncertainties during trace analysis, *SIGSOFT '04/FSE-12: Proc. 12th ACM SIGSOFT twelfth Int. symposium on Foundations of software engineering*, pp.3–12 (2004).
8) Gansner, E.R. and North, S.C.: An Open Graph Visualization System and its Applications to Software Engineering, *Software — Practice and Experience*, Vol.30, No.11, pp.1203–1233 (2000).
9) GNU Project: GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/
10) GNU Project: `indent` C source beautifier. http://mysite.wanadoo-members.co.uk/indent/beautify.html
11) Gondow, K.: Homepage for an educational operating system `udos`. http://www.sde.cs.titech.ac.jp/~gondow/udos/
12) Gondow, K., Suzuki, T. and Kawashima, H.: Binary-Level Lightweight Data Integration to Develop Program Understanding Tools for Embedded Software in C, *Proc. 11th Asia-Pacific Software Engineering Conference* (*APSEC*), pp.336–345 (2004).
13) Graham, S.L., Kessler, P.B. and Mckusick, M.K.: Gprof: A call graph execution profiler, *SIGPLAN '82: Proc. 1982 SIGPLAN symposium on Compiler construction*, New York, NY, USA, pp.120–126, ACM Press (1982).
14) Hall, P.A.V.: Overview of reverse engineering and reuse research, *Information and Software Technology*, Vol.34, No.4, pp.239–249 (1992).
15) Higo, Y., Kamiya, T., Kusumoto, S. and Inoue, K.: ARIES: refactoring support tool for code clone, *3-WoSQ: Proc. 3rd workshop on Software quality*, New York, NY, USA, pp.1–4, ACM Press (2005).
16) Horwitz, S., Reps, T. and Binkley, D.: Interprocedural slicing using dependence graphs, *ACM Trans. Prog. Lang. Syst.*, Vol.12, No.1, pp.26–60 (1990).

17) Jakobsen, M.R. and Hornbaek, K.: Evaluating a fisheye view of source code, *CHI '06: Proc. SIGCHI conf. on Human Factors in computing systems*, New York, NY, USA, pp.377–386, ACM Press (2006).
18) Kernighan, B.W. and Pike, R.: *The Practice of Programming*, Addison-Wesley Pub (1999).
19) Knuth, D.E.: *Literate Programming* (*Center for the Study of Language and Information*) *Lecture Notes, No*, Van Nostrand Reinhold Computer (1989).
20) Kramer, D.: API documentation from source code comments: A case study of Javadoc, *SIGDOC '99: Proc.17th annual Int. Conf. on Computer documentation*, pp.147–153 (1999).
21) Marcus, A. and Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing, *ICSE '03: Proc. 25th Int. Conf. on Software Engineering*, pp.125–135 (2003).
22) Microsoft Corp.: Homepage for FAT32 File System Specification. http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx
23) Murphy, G., Notkin, D. and Lan, E.-C.: An empirical study of static call graph extractors, *Proc. 18th Int. Conf. on Software Engineering* (*ICSE-18*), pp.90–99 (1996).
24) Ohba, M.: Homepage for the concept keyword mining tool. http://www.sde.cs.titech.ac.jp/˜m-ohba/cktfidf/
25) Ohba, M. and Gondow, K.: Toward mining "concept keywords" from identifiers in large software projects, *MSR '05: Proc. 2005 Int. workshop on Mining Software Repositories*, pp.1–5 (2005).
26) Ohba, M. and Gondow, K.: Maintaining Traceability Links between Implementation-Level Restrictions and Source Code for Program Understanding, *The 10th IASTED Int. Conf. on Software Engineering and Applications* (*SEA2006*), No.514-146 (2006).
27) Salton, G. and Buckley, C.: Termweighting approaches in automatic text retrieval, *Information Processing and Management*, Vol.24, No.5, pp.513–523 (1988).
28) Tama Communications Corp.: GNU GLOBAL Source Code Tag System. http://www.gnu.org/software/global/
29) 徳永健伸：情報検索と言語処理，東京大学出版会 (1999).

(平成 18 年 12 月 5 日受付)
(平成 19 年 5 月 9 日採録)

大場 勝（正会員）

1978 年生．2004 年北陸先端科学技術大学院大学情報科学研究科修士課程修了．同年東京工業大学大学院情報理工学研究科計算工学専攻博士課程に進学，現在に至る．ソフトウェア開発環境やプログラム理解に興味を持つ．日本ソフトウェア科学会，IEEE 各会員．情報科学修士．

権藤 克彦

1966 年生．1994 年東京工業大学大学院理工学研究科情報工学専攻博士課程修了．同大学助手，講師の後，1998 年北陸先端科学技術大学院大学情報科学研究科助教授．ブラウン大学客員研究員（2000〜2001 年）．2003 年より東京工業大学大学院情報理工学研究科助教授．ソフトウェア開発環境に興味を持つ．日本ソフトウェア科学会，ACM 各会員．博士（工学）．