修士学位論文

題目

ソースコード出典調査のための
コードクローン情報収集システムの試作

指導教員
井上 克郎 教授

報告者
佐々木 裕介

平成 23 年 2 月 7 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 22 年度 修士学位論文

ソースコード出典調査のためのコードクローン情報収集システムの試作

佐々木 裕介

内容梗概

近年，開発コストの削減や信頼性の高いソフトウェアの開発を目的として，企業などでオープンソースソフトウェア（OSS）全体やその一部を再利用することが増えてきている．しかし OSS の再利用にはライセンス違反などのリスクが生じる．ライセンス違反などのリスクを軽減するためには，再利用したいソースコードファイル（ファイル）の開発元やライセンスなどの調査を行う必要がある．

しかし，OSS 間ではコピー＆ペーストなどによる再利用が頻繁に行われており，再利用に必要な情報すべてを把握することは難しい．OSS 間でコピー＆ペーストが行われると，コードクローンが発生する．コードクローンとは，同一もしくは類似したコード片のことである．再利用が頻繁に行われている OSS のプロジェクト間には，多くのコードクローンが存在している．

そこで再利用に必要な情報を集めるために，コードクローン情報を用いて出典調査を行う．コードクローン情報とは，再利用したいファイルに対してコードクローンを共有しているファイル（類似ファイル）とその類似ファイルを取得することのできる URL の対を指す．出典調査とは再利用したいファイルの入手元と起源を特定することである．入手元とは，再利用したいファイルとコード部分が全く同じファイルをもつコードクローン情報を指す．起源とは，更新日時が最も古いファイルをもつコードクローン情報である．入手元と起源を特定することで，再利用に必要な情報を効率的に集めることができる．入手元と起源を特定するには多くのコードクローン情報を集める必要があるが，手動でコードクローン情報を集めるには膨大な時間がかかるという問題がある．

そこで Web 上にある既存のソースコード検索システムを利用して，コードクローン情報を自動で収集するシステム（提案システム）を試作した．提案システムはまず，入力として利用者が再利用したいファイルを受け取る．入力されたファイル中に出現する単語を用いて検索クエリ（クエリ）を生成する．このとき，ファイル名を用いたクエリも生成し併用する．次に生成したクエリを検索システムへ入力し，検索結果として返ってきたファイルとその URL を検索システムから取得する．入力されたファイルとソースコード検索システムか

ら取得したファイルを比較してコードクローン検出を行い，入力されたファイルに対する類似ファイルを検出する．最後に類似ファイルとその URL の対をコードクローン情報として出力する．

さらに，ソースコード検索システムから効率的に類似ファイルを集めるために，どのようなクエリを入力すればよいか実験を行い，クエリの生成方法に関する考察を行った．実験により，同じクエリでも利用する検索システムによっては異なる類似ファイルが見つかることが判明した．また全文検索を用いたソースコード検索システムから，少ない取得ファイル数で効率的に類似ファイルを集めるには，ファイル名を検索システムに入力するよりも，入力されたソースコード中に登場する単語を複数組み合わせてクエリを作成した方がよいことがわかった．一方で，利用するソースコード検索システムからできるだけ多くの類似ファイルを取得したい場合は，ソースコード中に登場する単語を組み合わせたクエリよりも，ファイル名のみを用いてクエリを生成する方がよい結果が出ることがわかった．

試作したシステムを用いて，実際に出典調査を試みた．結果として，入力したファイルの入手元と起源を取得することができた．さらに，特定した入手元と起源から，ファイルの開発元及びそのファイルがどのように再利用されてきたかを知ることができた．

主な用語

コードクローン
ソースコード検索システム
検索クエリ

目 次

図 目 次

表 目 次

1　はじめに

近年，企業などでオープンソースソフトウェア（以下，OSS）全体やその一部を再利用することが増えてきている．既存の OSS を再利用することで，開発コストを削減することができる．また既存の質の高いソースコードを用いることができれば，質の高いソフトウェアを短期間で開発することができる．

しかし，OSS の再利用には様々なリスクを伴う．例えばライセンス違反を行うと多額の賠償を請求されたり，製品回収を求められる．また再利用したファイルにバグが混入するといったことも生じる．安全に再利用を行うためには，例えば再利用したいファイルの開発元やライセンスを調査し，把握しておく必要がある．しかし，OSS 間ではコピー＆ペーストなどによる再利用が頻繁に行われており，再利用に必要な情報すべてを把握することが困難であるという問題がある．

互いに再利用を行っている OSS のプロジェクト間には多くのコードクローン [7] が存在する．コードクローンとは，同一もしくは類似したコード片のことである．また以降の説明では，あるファイルに対してコードクローンを有するファイルのことを類似ファイルと呼ぶ．

そこでコードクローン情報を用いて出典調査を行う．コードクローン情報とは，再利用したいファイルに対する類似ファイルと，その類似ファイルを取得可能な URL の対である．出典調査とは，再利用したいファイルの入手元と起源を特定することである．入手元とは，再利用したいファイルとコード部分が全く同じファイルを指すコードクローン情報を指す．起源とは，更新日時が最も古いファイルを指すコードクローン情報である．入手元と起源は 1 つのファイルに対し複数存在することもある．

出典調査を十分に行うためには，多くのコードクローン情報を集める必要がある．しかし再利用したいファイルのコードクローン情報を手動で集めるには多くの時間を要する．この問題を解決するため，Web 上にある既存のソースコード検索システムを用いて，コードクローン情報を収集するシステム（以降，提案システム）を試作した．

提案システムは，入力として利用者が再利用したいファイルを受け取り，そのコードクローン情報を Web 上から収集し，出力する．より具体的には，まず提案システムは既存のソースコード検索システムに対して，入力されたファイルに出現する単語を用いた検索クエリ（以降，クエリ）を生成する．入力されたファイルを含むファイルのファイル名がわかれば，ファイル名を用いたクエリも生成し，併用する．次に生成したクエリを検索システムへ入力し，検索結果としてファイルとその URL の情報を取得する．最後に取得したファイルに対してコードクローン検出を行い，類似ファイルを判別する．そして類似ファイルとその URL の対をコードクローン情報として出力する．提案システムの利用者は出力されたコードクローン情報を利用して，出典調査を行う．

ソースコード検索システムから効率的に類似ファイルを集めるために，どのようなクエリを入力すればよいか実験を行い，クエリの生成方法に関する考察を行った．実験により，同じクエリでも利用する検索システムによっては異なる類似ファイルが見つかることが判明した．また全文検索を用いたソースコード検索システムから，少ない取得ファイル数で効率的に類似ファイルを集めるには，ファイル名を検索システムに入力するよりも，入力されたソースコード中に登場する単語を複数組み合わせてクエリを作成した方がよいことがわかった．一方で利用するソースコード検索システムからできるだけ多くの類似ファイルを取得したい場合は，ソースコード中に登場する単語を組み合わせたクエリよりも，ファイル名のみを用いてクエリを生成する方がよい結果が出ることがわかった．

試作したシステムを用いて，実際に出典調査を試みた．結果として，入力したファイルの入手元と起源を取得することができた．さらに，同じソースコードでも，組織によっては全く異なるライセンスが適用されているケースがあることが判明した．また試作したシステムを用いることで，入力ファイルと類似したソースコードを利用している組織をいくつか見つけることができた．

以降，2 節において本研究の背景について説明する．3 節において提案システムについて説明する．4 節において提案システム内で用いているクエリの生成手法およびシステム全体の処理速度に関する実験と，その考察について述べる．5 節では提案システムの適用事例について述べる．6 節では本研究の関連研究について述べる．7 節でまとめと今後の課題について述べる．

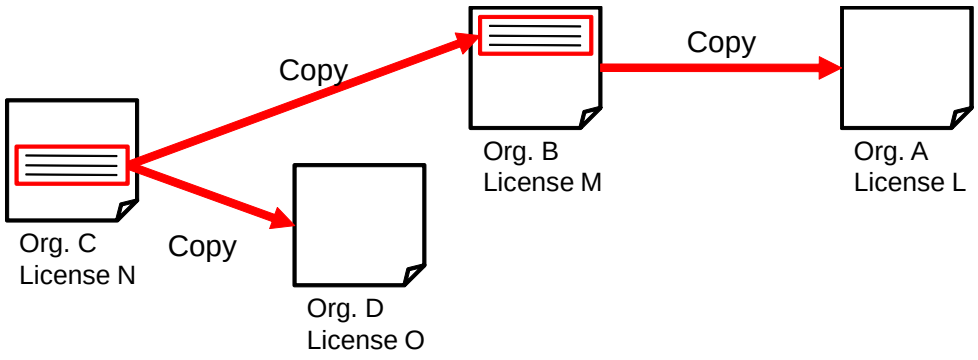

図 1: OSS 間でのソースコード再利用

2 背景

近年，企業などで OSS 全体やその一部を再利用することが増えてきている．既存の OSS を再利用することで，開発コストを削減することができる．また一から新たにソフトウェアを開発するよりも，既存の質の高いソフトウェアを再利用した方が，質の高いソフトウェアを開発することができる．

その一方で再利用にはリスクが生じる．例えばライセンス違反を起こしてしまうと，多額の賠償を請求されたり，製品回収を要求されることがありうる．再利用を行うためには，再利用したいファイルを含むファイルの開発元やライセンスの情報を調査する必要がある．

2.1 OSS 間で起こるソースコード再利用の例

OSS のプロジェクト間では，互いのソースコードを再利用するということが頻繁に行われている．このため 1 つの OSS に多くの団体が関与していることも少なくない．

例えば図 1 において，組織 A はライセンス L を適用されたソースコードを持っているものとする．この組織 A のソースコードはもともと組織 B が持っていたものを移植したものであり，さらに組織 B はそのソースコードを組織 C から移植している．また，組織 A がソースコードを入手した後，ソースコードの開発元である組織 C は組織 D へ開発を引き継いでいる．この場合，図 1 に示す組織 A のもつソースコードには，組織 A, B, C, D の 4 つの組織が関わっていることになる．また組織 B が組織 C のライセンス N に違反してソースコードを利用していた場合，組織 B だけでなく組織 A もソースコードの利用取りやめを要求される可能性がある．

以下に OSS 間で起こるソースコード再利用の具体例を示す．

データの圧縮/展開を行うライブラリ zlib のファイルが Mozilla や wxPython などを始めとする多くのアーカイブで利用されている．Mozilla のアーカイブに含まれている Firefox

などのウェブブラウザでは，プラグインインストールの際に zlib を利用している．また wxPython ではデータ圧縮/展開の機能を提供するために zlib を用いている．

libpng や libjpeg といった画像処理ライブラリのファイルが Mozilla や OpenCV などを始めとする多くのアーカイブで利用されている．Mozilla の Firefox, Thunderbird などでは，PC 画面上に画像表示を行う機能などが実装されている．libpng や libjpeg といったライブラリを利用しているのはこれらの機能を実装するためである．OpenCV は様々な画像処理を行うためのライブラリであり，libpng や libjpeg が提供する機能を用いて新たな機能を実装している．

Mozilla の JavaScript 実行エンジンである OSSP のソースコードが FreeSWITCH でも使われている．FreeSWITCH は VoIP などを使ってインターネット上の PC と会話を行うことのできるソフトウェアだが，機能拡張のために JavaScript を利用している．そのために Mozilla の JavaScript エンジンが使われている．

Boehm GC というガーベッジコレクレターが Mozilla や GCC，Kaffe など多くのアーカイブで利用されている．GCC や Kaffe では対応している Java などの言語でガーベッジコレクターが使われるため，Boehm GC を利用している．Mozilla でもメモリリーク検出を目的として Boehm GC を利用している．

libffi のファイルは gcc や smalltalk などで利用されている．libffi は FFI を提供するライブラリである．FFI とは高レベルレイヤ言語から低レベルレイヤ言語で書かれたライブラリを呼び出す際などに用いるインターフェイスのことで，gcc や smalltalk では高レベルレイヤの言語をコンパイルするために libffi を利用している．

このように，多くの OSS 間で再利用が頻繁に行われており，再利用に必要な情報を集めるには，図 1 の例で挙げたように様々な関係組織に関して調査を行わなければならないという問題がある．

2.2 本研究の目的

互いに再利用を行っている OSS のプロジェクト間には，多くのコードクローン [7] が存在している．本研究では，ファイルを再利用する際に必要な情報を集めるために，そのファイルに対するコードクローン情報が利用できないかと考えた．再利用したいファイルに対するコードクローン情報をたくさん集めれば，そのファイルの入手元と起源を特定することができる．入手元と起源を特定できれば，再利用時に必要な調査を効率的に行うことができる．ただし，出典調査には多くのコードクローン情報を集める必要がある．

そこで本研究では，ファイルを再利用する際の出典調査を効率化するため，Web 上のソースコード検索システムを用いて効率よくコードクローン情報を集めることを目的として，コードクローン情報収集システム（提案システム）を試作した．提案システムについて 3 節で

説明する.

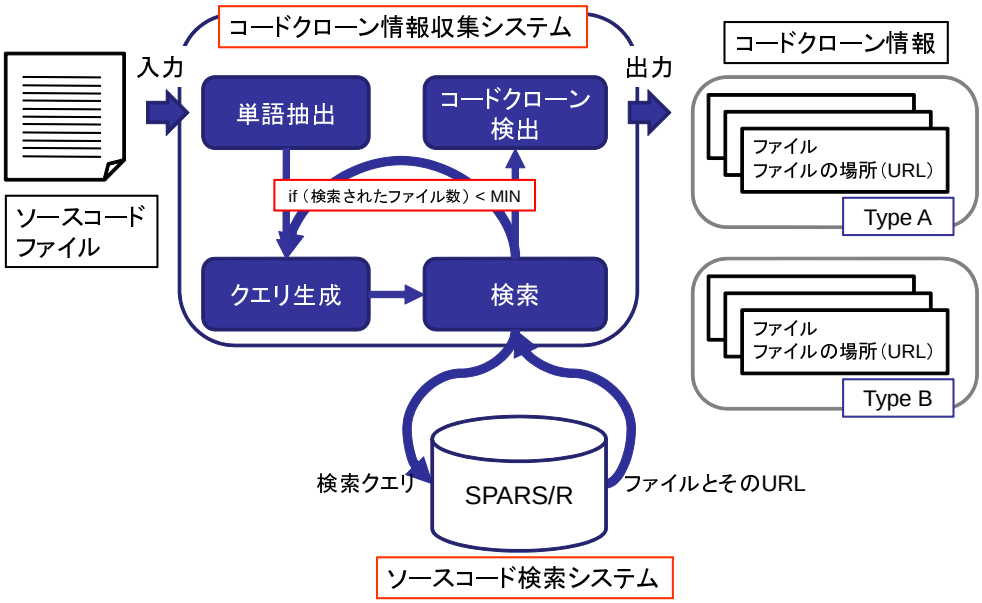

図 2: コードクローン情報収集システム概要

3　コードクローン情報収集システム

出典調査を行うためには，できるだけ多くのコードクローン情報を収集する必要がある．任意のファイルに対して全てのコードクローン情報を集めるには，少なくとも Web 上に存在するすべての OSS のソースコードを把握しておく必要があるが，1 つの組織が短時間で世の中に散らばっているすべての OSS を把握することは難しい．よって，1 つの組織がもつソースコードの情報だけでは，十分なコードクローン情報が集められないことがある．

そこで，手元にあるソースコード以外の情報も把握するため，提案システムでは Web 上のソースコード検索システムを利用する．Web 上のソースコード検索システムからも手元から得られないコードクローン情報を取得することで，より多くのコードクローン情報を取得可能となる．

3.1　システム概要

提案システムの概要を図 2 に示す．提案システムではまずファイルを入力として受け取る．次に入力されたファイルから単語を抽出する．抽出した単語を用いてクエリを生成する．生成したクエリを Web 上のソースコード検索システムに入力として与える．すると生成したクエリに一致したファイルとその URL の集合がソースコード検索システムから返される．もし十分な量のファイルが返ってこなかった場合には再度検索を行う．検索システムから取得したファイル群から，入力されたファイルとコードクローンを共有しているファイル，類

似ファイルを検出する．最後に類似ファイルとその URL の対をコードクローン情報として出力する．

類似ファイルの中でも，特にコメントを除外するなどの正規化を行ったときに入力されたファイルと等しくなるファイルをファイルクローンと呼ぶ [8]．ファイルクローンを含むコードクローン情報を TypeA コードクローン情報として扱う．TypeA として出力されたコードクローン情報は入手元として利用することができる．TypeA に属さない，すなわちコード部分の識別子や定数などが異なるコードクローン情報は TypeB コードクローン情報として扱う．

入力されたファイルに対する入手元と起源をより確実に得るには，検出されるコードクローン情報は多ければ多いほどよい．またコードクローン情報を効率よく収集するには，短時間でシステム全体の処理を完了する必要がある．

3.2 ソースコード検索

ソースコード検索システムからすべてのファイルを取得し，その中から類似ファイルを検出するのでは時間がかかりすぎてしまう．そこで，ソースコード検索システムにクエリを入力し，検索結果として見つかったファイルから類似ファイルを探し出す．

検索システムは各ソースコードに対して索引付けを行っている．多くの場合索引にはソースコード中の単語が用いられるが，どの単語を用いるかは検索システムによって様々である．検索者が検索システムに対して単語を組み合わせて生成したクエリを入力すると，検索システムは入力された単語を含む索引を持つソースコードを出力として検索者に返す．

よって，索引付けの方法が分かっていれば，その検索システムに蓄えられているすべての類似ファイルを効率的に取得可能である．

例えば，SPARS/R[3] は KR 法 [14] に特化した索引付けを行っている．SPARS/R の検索対象は現在のところオブジェクト指向プログラムに限られるため，索引にはクラス名やフィールド名，メソッド名といった識別子名に含まれる単語が優先して使われている．よってこれらを優先してクエリに利用し，検索を行えば効率的に類似ファイルを取得できると考えられる．

一方，索引付けの方法がわからないソースコード検索システムを使う場合，類似ファイルを効率的に取得するのは難しい．うまくクエリを生成しなければ，短時間で多くの類似ファイルを取得することはできない．

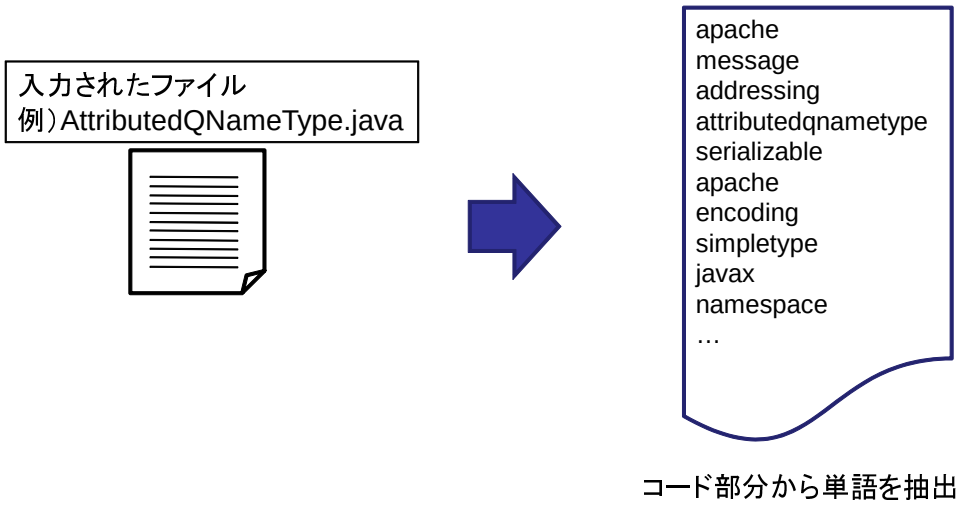

図 3: 単語の抽出

3.2.1 クエリの生成

一般的なソースコード検索システムではキーワード検索が用いられている [1, 2]．キーワード検索では単語を組み合わせることでクエリを作成する．

AND 検索の場合，組み合わせる単語が多ければ多いほど，検索されるファイルは少なくなる．逆に組み合わせる単語が少ないと，大量のファイルが検索結果に表示される．提案システムではこの AND 検索の性質を利用してクエリ生成を行っている．以下に手順を示す．

まず入力されたファイルから単語を抽出する．クエリに利用する単語をコード部分あるいはコメント文のどちらから選出するか決める．以降の説明ではコード部分から抽出した単語を利用することにする．図 3 に示す例は，unicore というソフトウェアに含まれる AttributedQNameType.java ファイルのコード部分から識別子名などの文字列のみを抽出したものである．

さらに抽出した文字列が Snake Case や Camel Case のような複合語であった場合には，これを分割することができる．Snake Case とは，例えば ”SNAKE_CASE” のように，2 つ以上の単語を”_”のような意味のない文字列を間にはさんで記述した複合語のことである．”SNAKE_CASE” の場合は ”SNAKE”, ”CASE” の 2 語に分割可能である．Camel Case とは，例えば ”CamelCase” のように，分割したときに単語の 1 文字目が大文字となるように記述した複合語のことである．”CamelCase” の場合は ”Camel”, ”Case” の 2 語に分割可能である．複合語の分割は，その単語をより一般的な単語に置き換えたいときに行う．

そして不要な単語を削除する．例えば，クエリに cp932 の 932 や年号などの数字列を含めるべきか，含めないべきかということを考慮する必要がある．

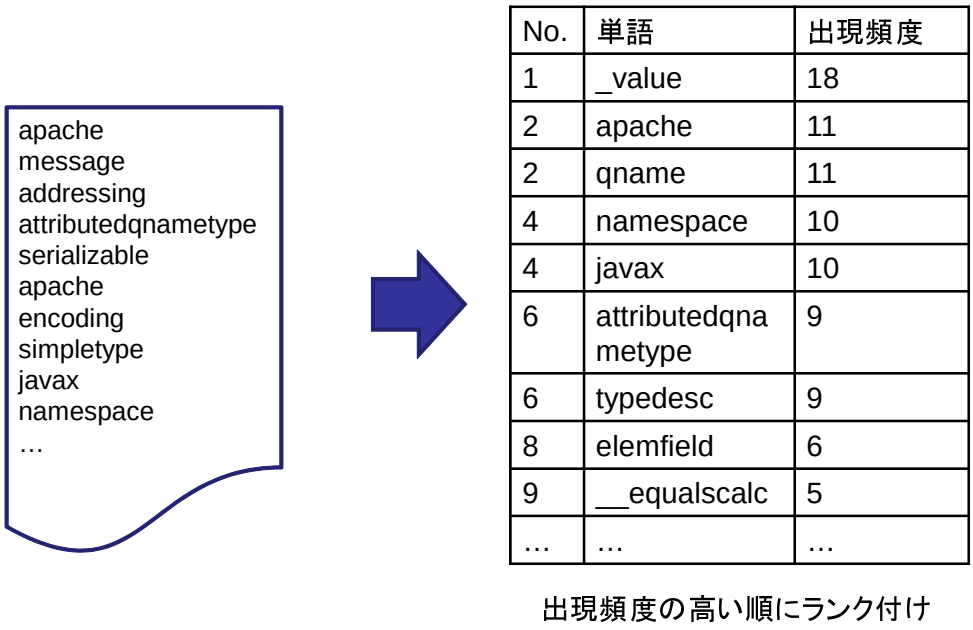

図 4: 単語のランク付け

図 5: 単語の削減

そして得られた単語をランク付けする．図 4 に示す例では，単語を出現回数の多い順にランク付けしている．

始めから単語を一つに絞ってクエリ生成を行うと，検索結果に大量のファイルが表示される．そこで初めはできるだけ多くの単語を用いてクエリを作成する．必要なファイルが見つからなければ，優先度の低い単語を 1 つずつクエリから外していく．

例えば図 5 左表に示す例では，”_value” という単語が最も優先度が高くなっている．こ

の ”_value” 1 つのみを用いてクエリを作成すると，類似ファイルでないファイルが大量に含まれた検索結果が返されてしまい，ファイルの取得や類似ファイルの検出に膨大な時間がかかってしまう．

そこで始めは単語を多めに設定する．この始めに設定する単語数のことを，以降の説明では初期単語数と呼ぶことにする．図 5 左表の例では 18 個の単語を用いてクエリを生成している．クエリは以下のようになる．

- _value & apache & qname & ... & _equals & other

仮に検索結果として十分なファイル数 MIN が返されなかった場合，図 5 右表のように最も優先度の低い ”other” という単語をクエリから削除し，単語数を 17 に減らす．すると単語数 18 でクエリを投げたときよりも多くのファイルが検索結果として返ってくる．クエリは以下のようになる．

- _value & apache & qname & ... & _equals

結果として，システム内部では図 2 に示す流れでクエリ生成/検索が繰り返されることになる．以上の工夫により，類似ファイルでないファイルを Web から取得する時間を節約することができる．

3.3 コードクローン検出

入力されたソースコードとソースコード検索システムから得られたファイル群を比較し，コードクローン検出を行う．入力されたソースコードに対するコードクローンを含むファイルを，類似ファイルとして出力する．

3.4 ファイルクローン検出

コメントを除外するなどの正規化を行ったときにソースコードが等しくなるファイルの関係をファイルクローン関係と呼び，それぞれのファイルをファイルクローンと呼ぶ [8]．ファイルクローンはファイルのコード記述部分を変更せずにファイルのコピー&ペーストを行った場合などに多く発生する．

4 節でも議論するが，検出されるファイルの中には入力されたソースコードに対するファイルクローンも多く含まれている．ファイルクローンはコードクローンよりも短時間で検出できる．提案システムでは，入力されたソースコードに対するファイルクローンをコードクローンとは別に検出し，Type A として扱う．

3.5 システムの試作

前述したコードクローン情報収集システムを実際に試作した.

3.5.1 単語の抽出

試作したシステムでは以下のパラメータを設定可能である.

- Snake Case 分割を行う/行わない
- Camel Case 分割を行う/行わない
- cp932 の 932 や年号などの数字列を除外する/除外しない

なお，単語分割アルゴリズムの都合上，cp932 のような単語は ”cp”,”932” の 2 単語として抽出される.

3.5.2 クエリに用いる単語

試作したシステムでは，クエリ生成時に利用する単語を入力されたファイルのどの要素から抽出するかを選ぶことができる.

- ファイル名
- コード中の単語
  - コメント文などを除いたコード部分に出現する単語
  - コメント文に出現する単語

3.5.3 単語のランク付け

さらに，抽出された単語に対するランク付けの方法を以下の 4 種類用意している.

- 出現回数の多い単語を優先
- 出現回数の少ない単語を優先
- 共起している単語の種類が多い単語を優先
- 共起している単語の種類が少ない単語を優先

3.5.4 利用するソースコード検索システム

試作したシステムでは SPARS/R をソースコード検索システムとして用いる．Google Code Search[2]（以降，GCS）や Koders[1] については利用規約に抵触する可能性があったため提案システムから利用することはできなかった．

3.5.5 コードクローン検出

類似ファイルを検出するために，提案システムでは既存のコードクローン検出ツール CCFinder[4] を利用する．CCFinder は多くの既存研究で引用されている [5, 8, 6]．

3.5.6 ファイルクローン検出

文献 [8] において利用されている FCFinder を用いる．FCFinder は大量のソースコードに対するファイルクローン検出を高速に行うことができる．

4 調査実験

本節では，以下の問題を調査することを目的として行った実験について説明する．

RQ1 試作したシステムにおいて処理時間の内訳はどうなっているか

RQ2 利用するソースコード検索システムによって得られる類似ファイルに差異が生じるか

RQ3 ソースコード検索システムに入力するクエリによって検索精度はどう変化するか

RQ1 については，実際に開発したシステムに対して SPARS/R をソースコード検索システムとして利用した場合，入力が与えられてから各処理にかかる時間を測定する．そしてどの処理がボトルネックとなっているか考察を行う．

RQ2 については，SPARS/R と GCS, Koders をソースコード検索システムとして利用したときに，手動で色々なクエリを入力した結果，得られる類似ファイルに差異はあるかを評価する．もし差異が認められれば，複数のソースコード検索システムを同時に用いた方が，より多くの類似ファイルが得られるということになる．

4.1 検索精度

RQ3 については，以下の項目を調査し，どのようにクエリを生成すれば効率よく十分な量の類似ファイルを収集できるか考察する．

RQ3.1 クエリ生成時に用いる要素（ファイル名/コメントなどを除いたコード部分/コメント文）を変えることで検索精度に変化はあるか

RQ3.2 単語のランク付けは検索精度に何らかの影響を及ぼすか

RQ3.3 クエリに含める単語の数を増減させることによって，検索精度に何らかの影響があるか

検索精度とは図 6 で示す P_1,R_1,P_2,R_2 のことであり，以下の式で表わされる．

$$P_1 = \frac{(\text{検索システムから返ってきた類似ファイル数})}{(\text{検索システムから返ってきたファイル数})}$$

$$R_1 = \frac{(\text{検索システムから返ってきた類似ファイル数})}{(\text{検索システムが保持している類似ファイル数})}$$

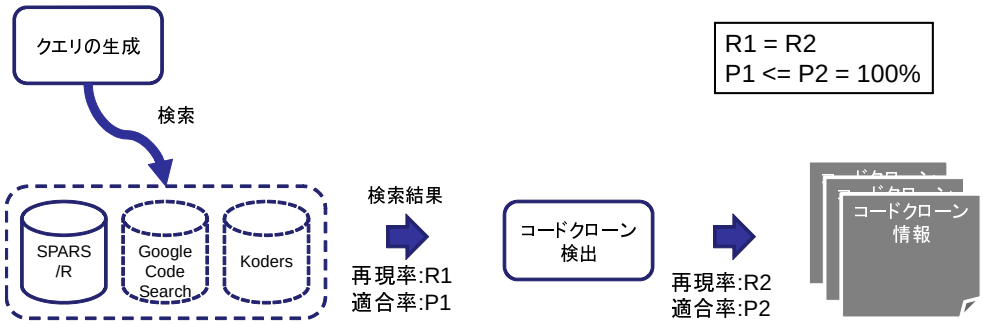

図 6: システム内部の適合率および再現率 (1)

$$P_2 = \frac{(\text{コードクローン検出により検出された類似ファイル数})}{(\text{コードクローン検出により検出されたファイル数})}$$

$$R_2 = \frac{(\text{コードクローン検出により検出された類似ファイル数})}{(\text{検索システムが保持している類似ファイル数})}$$

ここで，コードクローン検出ツール CCFinder の出力は，再現率/適合率ともに 100%であるものとする．よって調査が必要になるのは P_1, R_1 の値である．以降，適合率とは P_1 のこと，再現率とは R_1 のことを指すものとする．

4.1.1 再現率の評価

Web 上からできるだけ多くの類似ファイルを集めるためには，R_1 が 100% にできるだけ近い値である必要があるため，これについて評価を行う．

SPARS/R についてはリポジトリに含まれるファイル情報を予め把握することができるため，R_1 の値を測定することができる．

一方，外部のソースコード検索システム上のリポジトリに含まれているソースコードをすべて把握することは非常に難しい．そのため，外部システムを利用することを考えた場合，再現率の評価は困難である．

よって以下の各項目についてのみ評価を行う．

- 返ってくる類似ファイルの総数
- 返ってくるファイルクローン（FC）数
- 返ってくるコードクローン（CC）数

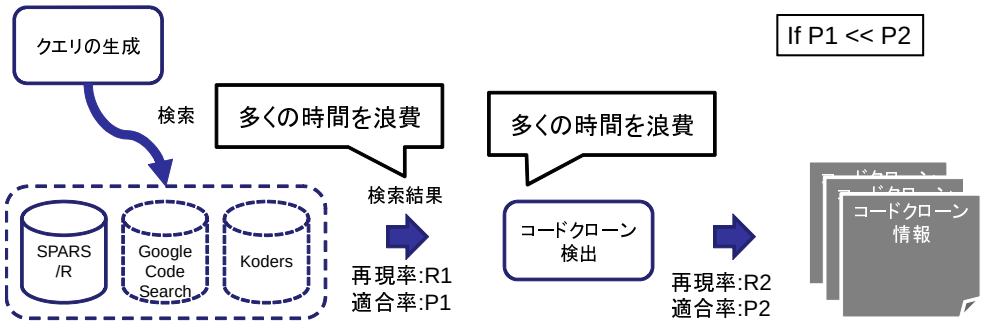

図 7: システム内部の適合率および再現率 (2)

4.1.2 適合率の評価

図 7 に示すように，P_1 が P_2 よりも大幅に低い，もしくは 0 だったと仮定すると，図 7 上 4 において検索結果の取得で大量の余分なファイルを Web 上から取得することになり，さらに図 7 上 5 においてコードクローン検出時に大量の時間を浪費することになるため，コードクローン情報収集の効率は大幅に下がることになる．

よって P_1 について評価を行うことで，提案システムが効率的に類似ファイルを出力できるよう，クエリがうまく作成できているか確認することができる．

4.2 実験手順

まず以下の条件を満たすファイルを入力として与える．

- 対象とする検索システムのリポジトリに含まれていることが予めわかっているファイル

そして実際にアルゴリズムに基づいて単語を選出し，選出された単語を用いて手動でクエリを生成する．生成したクエリを対象とする外部のソースコードシステムに入力する．返ってきた結果に対して以下の各項目について評価を行う．

- 返ってくる類似ファイルの総数
- 返ってくるファイルクローン（FC）数
- 返ってくるコードクローン（CC）数
- 適合率（前述した P_1）

4.2.1 利用するソースコード検索システム

本実験ではクエリを入力する検索システムとして，SPARS/R と GCS, Koders を用いる．ただし，SPARS/R については提案システム内部に組み込まれているため，実験におけるすべての作業を自動で行うが，GCS と Koders については提案システムから利用することができないため，手動でクエリ生成と入力を行う．

4.2.2 入力ファイル

入力とするソースコードのファイルは，2011/01/01 現在 SPARS/R に登録されているファイル群の中から選出した次のファイルである．

- apache shale に含まれている ApplicationListener
- apache shale に含まれている BaseViewController
- apache shale に含まれている ListEntriesRule
- apache shale に含まれている Logon
- apache shale に含まれている Rolodex
- eclipse JDT に含まれている TypeVisitor

いずれも Koders，GCS で検索可能なことが確認されている．なお，SPARS/R には以下のプロジェクト中のソースコードが登録されていた．

- JDK に含まれているプロジェクト群
- apache に含まれているプロジェクト群
- eclipse に含まれているプロジェクト群
- sourceforge に含まれている特定のプロジェクト群

4.2.3 ファイル中に出現する単語を用いたクエリ

本実験では，3.2.1 節で説明した通りの方法で，以下に示す 3 種類のクエリを手動で作成し，検索システムへ入力した．

1. コード中で出現回数の多い単語を用いたクエリ
2. コード中で出現回数の少ない単語を用いたクエリ

3. コメント文中で出現回数の多い単語を用いたクエリ

単語分割の方法は以下に固定する．

- Snake Case 分割は行わない
- Camel Case 分割は行わない

また，数字（cp932 の 932 や年号など）は単語に含める．クエリに含める初期単語数は 19，図 2 における MIN の値は 0 とする．

4.2.4 ファイル名を用いたクエリ

ソースコード検索システムによっては，キーワードを用いたクエリの入力とは別に，ファイル名を入力として与えることが可能である [2, 1].

そこで本実験でも，上記のクエリの他に，ファイル名を用いてクエリを作成し，他の方法と比べて効率よくファイルを取得可能か調べることにする．

試作したシステムで用いている SPARS/R には，ファイル名を入力するインタフェースはないが，本実験で対象としている検索システム GCS と Koders にはファイル名を入力するインタフェースがあるためこれを利用することにする．

4.3 実験結果

以下に 2011/01/01 〜 2011/01/31 にかけて行った実験の結果を示す.

4.3.1 実行時間

表 1 は試作したシステムに対して，ランダムサンプリングにより選出したいくつかのファイルを入力として与えたときの実行時間を示す．ソースコード検索システムには SPARS/R を用いている．# of results は入力ファイルを与えた結果，SPARS/R から検索結果として返ってきたファイルの個数である．total は試作したシステム全体の処理にかかった時間であり，make queries はクエリの生成に要した時間，get files は SPARS/R からの検索結果及びファイル取得に要した時間，FCFinder は FCFinder 実行時間，CCFinder は CCFinder 実行時間である．

表 1: 処理時間

# of results	total (s)	make queries	get files (s)	FCFinder (s)	CCFinder (s)
22	23	2	5	9	1
78	27	2	7	9	1
148	35	2	11	12	3
357	90	1	18	19	43
36	29	4	6	10	2
40	30	3	7	10	2
4	21	1	3	8	1
13	25	2	3	11	1
4	22	3	2	9	1

4.3.2 ファイル名を用いたクエリ

表 2 に示すのは SPARS/R にファイル名を用いたクエリを入力して得られた実験結果である．# of results は検索エンジンから返ってきたファイル数を示す．# of FC はファイルクローン数，# of CC は入力したファイルに対してコードクローンを含むファイルの数（ファイルクローン含む）であり，recall of FC は検索システムから取得できたファイルクローンの再現率，recall of CC は検索システムから取得できた類似ソースコードの再現率，すなわち図 6 における R1 を示す．precision は図 6 における P1 である．

なお，検索結果として表示されたファイル数が 100 を越える場合は，ファイル取得に時間がかかりすぎるため，上位 100 件で検出を打ち切っている．検出が打ち切られた場合，もしくは検索結果が 0 だった場合には，recall of FC と recall of CC，precision は計算不可であるため欄を空白にしている．

表 2: SPARS/R に対してファイル名を用いたクエリを入力した結果

file name	# of results	# of FC	# of CC	recall of FC	recall of CC	precision
ApplicationListener	22	8	14	1	1	0.636363636
BaseViewController	78	6	6	1	1	0.076923077
ListEntriesRule	36	9	9	1	0.5	0.25
Logon	357	13	13	1	1	0.036414566
Relodex	148	5	11	1	1	0.074324324
TypeVisitor	40	2	2	1	1	0.05

表 3 に示すのは GCS にファイル名を用いたクエリを入力して得られた実験結果である．なお，検索結果として表示されたファイル数が 50 を越える場合は，ファイル取得に時間がかかりすぎるため，上位 50 件で検出を打ち切っている．

表 3: GCS に対してファイル名を用いたクエリを入力した結果

file name	# of results	# of FC	# of CC	precision
ApplicationListener	151	1	3	0.01986755
BaseViewController	4	1	1	0.25
ListEntriesRule	1	1	1	1
Logon	113	1	2	0.017699115
Relodex	3	1	2	0.666666667
TypeVisitor	123	2	2	0.016260163

表 4 に示すのは Koders にファイル名を用いたクエリを入力して得られた実験結果である．なお，検索結果として表示されたファイル数が 50 を越える場合は，ファイル取得に時間がかかりすぎるため，上位 50 件で検出を打ち切っている．

表 4: Koders に対してファイル名を用いたクエリを入力した結果

file name	# of results	# of FC	# of FC and CC	precision
ApplicationListener	60			
BaseViewController	2	1	1	0.5
ListEntriesRule	2	1	1	0.5
Logon	43	1	1	0.023255814
Relodex	2	1	1	0.5
TypeVisitor	54			

4.3.3 コード中の出現回数が多い単語を用いたクエリ

表 5 に示すのは SPARS/R にコード中に出現する単語を用いて作成したクエリを投げたときの実験結果である．単語のランク付けには，コード中に出現する回数が多い単語を優先して選ぶ方法をとっている．なお以降の実験結果では，SPARS/R の検索結果として返ってきたファイル数が 50 を越える場合は，処理時間が長すぎるものと予測し，検出を打ち切っている．

表 6 に示すのは GCS にコード中に出現する単語を用いて作成したクエリを投げたときの実験結果である．なお以降では，GCS の検索結果として表示されたファイル数が 30 を越える場合は，ファイル取得に時間がかかりすぎるため，検出を打ち切っている．

表 7 に示すのは Koders にコード中に出現する単語を用いて作成したクエリを投げたときの実験結果である．なお以降では，Koders の検索結果として表示されたファイル数が 30 を越える場合は，ファイル取得に時間がかかりすぎるため，検出を打ち切っている．

表 5: SPARS/R に対してコード中に出現する回数の多い単語を用いたクエリを入力した結果

file name	# of words	# of results	# of FC	# of CC	recall of FC	recall of CC	precision
ApplicationListener	6	0					
	5	2	0	0	0	0	0
	4	2	0	0	0	0	0
	3	1125					
BaseViewController	13	0					
	12	13	6	6	1	1	0.461538462
	4	13	6	6	1	1	0.461538462
	3	39	6	6	1	1	0.153846154
	2	90					
ListEntriesRule	19	0					
	18	9	9	9	1	0.5	1
	12	9	9	9	1	0.5	1
	5	9	9	9	1	0.5	1
	4	18	9	9	1	0.5	0.5
	3	63					
Logon	19	0					
	18	26	13	13	1	1	0.5
	8	26	13	13	1	1	0.5
	7	28	13	13	1	1	0.464285714
	6	42	13	13	1	1	0.30952381
	5	98					
Rolodex	19	0					
	18	11	5	11	1	1	1
	12	11	5	11	1	1	1
	5	11	5	11	1	1	1
	4	11	5	11	1	1	1
	3	33	5	11	1	1	0.333333333
	2	33	5	11	1	1	0.333333333
	1	33	5	11	1	1	0.333333333
TypeVisitor	19	0					
	18	4	2	2	1	1	0.5
	12	4	2	2	1	1	0.5
	5	4	2	2	1	1	0.5
	4	4	2	2	1	1	0.5
	3	5	2	2	1	1	0.4
	2	6	2	2	1	1	0.333333333
	1	549					

表 6: GCS に対してコード中に出現する回数の多い単語を用いたクエリを入力した結果

file name	# of words	# of results	# of FC	# of CC	precision
ApplicationListener	19	8	1	8	1
	18	120			0
BaseViewController	19	1	1	1	1
	18	1	1	1	1
	17	1	1	1	1
	16	1	1	1	1
	12	1	1	1	1
	5	1	1	1	1
	4	1	1	1	1
	3	94			
ListEntriesRule	5	0			
	4	2	1	1	0.5
	3	19348			0
Logon	19	1	1	1	1
	18	1	1	1	1
	17	2	1	1	0.5
	16	2	1	1	0.5
	12	2	1	1	0.5
	11	2	1	1	0.5
	10	2	1	1	0.5
	9	5919			
Rolodex	19	0			
	18	2	1	2	1
	17	2	1	2	1
	16	2	1	2	1
	12	2	1	2	1
	5	2	1	2	1
	4	2	1	2	1
	3	2	1	2	1
	2	66			
TypeVisitor	19	19	4	4	0.210526316
	4	19	4	4	0.210526316
	3	87			

表 7: Koders に対してコード中に出現する回数の多い単語を用いたクエリを入力した結果

file name	# of words	# of results	# of FC	# of CC	precision
ApplicationListener	19	0			
	18	1	1	1	1
	17	1	1	1	1
	16	1	1	1	1
	8	1	1	1	1
	7	221			
BaseViewController	19	0			
	18	1	1	1	1
	17	1	1	1	1
	16	1	1	1	1
	12	1	1	1	1
	5	1	1	1	1
	4	1	1	1	1
	3	7	1	1	0.142857143
	2	598			
ListEntriesRule	19	0			
	18	1	1	1	1
	17	1	1	1	1
	16	1	1	1	1
	12	1	1	1	1
	5	1	1	1	1
	4	2	1	1	0.5
	3	7	1	1	0.142857143
	2	8	1	1	0.125
	1	191198			
Logon	19	0			
	18	1	1	1	1
	17	1	1	1	1
	16	2	1	1	0.5
	12	2	1	1	0.5
	11	2	1	1	0.5
	10	3	1	1	0.333333333
	9	6	1	1	0.166666667
	8	13	1	1	0.076923077
	7	560			
Rolodex	19	0			
	18	1	1	1	1
	12	1	1	1	1
	5	1	1	1	1
	4	1	1	1	1
	3	1	1	1	1
	2	3	1	1	0.333333333
	1	3	1	1	0.333333333
TypeVisitor	19	52			

4.3.4 コード中の出現回数が少ない単語を用いたクエリ

表 8 に示すのは SPARS/R にクエリを入力したときの実験結果である．単語のランク付けには，コード中に出現する回数が少ない単語を優先して選ぶ方法をとっている．

表 8: SPARS/R に対してコード中に出現する回数の少ない単語を用いたクエリを入力した結果

file name	# of words	# of results	# of FC	# of CC	recall of FC	recall of CC	precision
ApplicationListener	2	1	0	0	0	0	0
	1	1096	0	0	0	0	0
BaseViewController	9	13	0	0	0	0	0
	4	52	6	6	1	1	0.115384615
	1	78	0	0	0	0	0
ListEntriesRule	4	18	9	18	1	1	1
	2	19	9	18	1	1	0.947368421
	1	91	0	0	0	0	0
Logon	5	4	0	0	0	0	0
	4	16	0	0	0	0	0
	3	39	0	0	0	0	0
	2	506	0	0	0	0	0
	1	1076	0	0	0	0	0
Relodex	1	5	0	0	0	0	0
TypeVisitor	14	4	2	2	1	1	0.5
	2	5	2	2	1	1	0.4
	1	90	2	2	1	1	0.022222222

表 9 に示すのは GCS にクエリを入力したときの実験結果である．

表 10 に示すのは Koders にクエリを入力したときの実験結果である．

表 9: GCS に対してコード中に出現する回数の少ない単語を用いたクエリを入力した結果

file name	# of words	# of results	# of FC	# of CC	precision
ApplicationListener	19	1	1	1	1
	7	1	1	1	1
	5	2	1	2	1
	4	37			
BaseViewController	19	1	1	1	1
	14	1	1	1	1
	9	2747			
Logon	19	1	1	1	1
	9	1	1	1	1
	7	11	1	1	0.090909091
	5	4407			
Relodex	19	2	1	2	1
	4	2	1	2	1
	3	84			
TypeVisitor	19	51			

表 10: Koders に対してコード中に出現する回数の少ない単語を用いたクエリを入力した結果

file name	# of words	# of results	# of FC	# of FC and CC	precision
ApplicationListener	19	1	1	1	1
	14	1	1	1	1
	9	1	1	1	1
	7	1	1	1	1
	5	3	1	3	1
	4	50			
BaseViewController	19	1	1	1	1
	14	1	1	1	1
	9	4	1	1	0.25
	7	4	1	1	0.25
	5	6	1	1	0.166666667
	4	6	1	1	0.166666667
	3	6	1	1	0.166666667
	2	6	1	1	0.166666667
	1	7	1	1	0.142857143
ListEntriesRule	19	1	1	1	1
	14	1	1	1	1
	9	1	1	1	1
	7	1	1	1	1
	5	7	1	1	0.142857143
	4	5024			
Logon	19	1	1	1	1
	14	1	1	1	1
	9	1	1	1	1
	7	1	1	1	1
	5	427			
Relodex	19	1	1	1	1
	14	1	1	1	1
	9	1	1	1	1
	7	1	1	1	1
	5	1	1	1	1
	4	1	1	1	1
	3	134			
TypeVisitor	19	52			

4.3.5 コメント中の出現回数が多い単語を用いたクエリ

表 11 に示すのは SPARS/R にクエリを入力したときの実験結果である．単語のランク付けには，コメント中に出現する回数が多い単語を優先して選ぶ方法をとっている．

表 11: SPARS/R に対してコメント中の出現回数が多い単語を用いたクエリを入力した結果

file name	# of words	# of results	# of FC	# of CC	recall of FC	recall of CC	precision
ApplicationListener	4	0					
	3	11	0	0	0	0	0
	2	176					
BaseViewController	4	0					
	3	19	0	0	0	0	0
	2	137					
ListEntriesRule	7	0					
	6	1	0	0	0	0	0
	5	1	0	0	0	0	0
	4	30	0	0	0	0	0
	3	367					
Logon	4	0					
	3	19	0	0	0	0	0
	2	65					
Rolodex	7	0					
	6	15	5	11	1	1	0.733333333
	5	15	5	11	1	1	0.733333333
	4	29	5	11	1	1	0.379310345
	3	71					
TypeVisitor	6	0					
	5	1	0	0	0	0	0
	4	1	0	0	0	0	0
	3	67					

表 12 に示すのは GCS にクエリを入力したときの実験結果である．

表 13 に示すのは Koders にクエリを入力したときの実験結果である．

表 12: GCS に対してコメント中の出現回数が多い単語を用いたクエリを入力した結果

file name	# of words	# of results	# of FC	# of CC	precision
ApplicationListener	19	70			
BaseViewController	6	0			
	5	9	1	1	0.111111111
	4	145000			
ListEntriesRule	19	17	0	0	0
	18	70			
Logon	19	60			
Rolodex	19	200			
TypeVisitor	19	156000			

表 13: Koders に対してコメント中の出現回数が多い単語を用いたクエリを入力した結果

file name	# of words	# of results	# of FC	# of CC	precision
ApplicationListener	19	17	1	17	1
	14	17	1	17	1
	13	33			
BaseViewController	19	1	1	1	1
	9	1	1	1	1
	8	1	1	1	1
	7	2	1	1	0.5
	6	2	1	1	0.5
	5	5	1	1	0.2
	4	67073			
ListEntriesRule	19	1	1	1	1
	15	1	1	1	1
	14	5	1	2	0.4
	13	46			
Logon	19	1	1	1	1
	17	1	1	1	1
	16	3	1	1	0.333333333
	11	3	1	1	0.333333333
	10	4	1	1	0.25
	8	4	1	1	0.25
	7	296			
Rolodex	19	1	1	1	1
	18	1	1	1	1
	17	5	1	1	0.2
	16	13	1	1	0.076923077
	15	13	1	1	0.076923077
	14	47			
TypeVisitor	10	0			
	9	628			

4.3.6 各検索システムから見つかった類似ソースコード数

表 14 に示すのは，表 2,5,11 までの実験を行う過程で発見した，ファイルクローン，コードクローンの総数である．file name は入力した各ファイルのファイル名（拡張子除く）であり，# of FC は見つかったファイルクローン数，# of CC は入力したファイルに対するコードクローンを含むファイル数（ファイルクローン除く），total は # of FC と # of CC の合計を示す.

表 14: SPARS/R から取得された類似ソースコード総計

file name	# of FC	# of CC	total
ApplicationListener	8	6	14
BaseViewController	6	0	6
ListEntriesRule	9	9	18
Logon	13	0	13
Rolodex	5	6	11
TypeVisitor	2	0	2

表 15 に示すのは，表 3,6,12 の実験で発見した，ファイルクローン，コードクローンの総数である．

表 15: GCS から取得された類似ソースコード総計

file name	# of FC	# of CC	total
ApplicationListener	1	8	9
BaseViewController	1	0	1
ListEntriesRule	1	0	1
Logon	1	0	1
Rolodex	1	1	2
TypeVisitor	4	0	4

表 16 に示すのは，表 4,7,13 の実験で発見した，ファイルクローン，コードクローンの総数である．

表 16: Koders から取得された類似ソースコード総計

file name	# of FC	# of CC	total
ApplicationListener	1	19	20
BaseViewController	1	0	1
ListEntriesRule	1	1	2
Logon	1	0	1
Rolodex	1	0	1
TypeVisitor	13	0	13

4.4 考察

まず RQ1 について考察する．表 1 によると，ソースコード検索システムから返ってくる検索結果の個数がシステムの処理時間に大きく影響していることがわかる．特にファイルの取得時間とコードクローン検出時間は検索結果の個数に大きく依存している．効率的に類似ファイルを集めるには，ソースコード検索システムに対してうまくクエリを入力する必要がある．

RQ2 について，表 14, 15, 16 に示すとおり，ソースコード検索システムによって異なる類似ファイルが見つかっている．

ApplicationListener

SPARS/R では 8 つのファイルクローンが見つかっているのに対して，Koders では 1 つのファイルクローンしか見つかっていない．逆に，SPARS/R ではコードクローンを含むファイル（ファイルクローン除く）が 1 つしかなかったのに対して，GCS では 8 つ，Koders では 19 ものファイルが見つかっている．

BaseViewController

SPARS/R ではファイルクローンが 6 つ発見されたのに対して，他の検索システムでは 1 つしか発見されていない．

ListEntriesRule

SPARS/R では類似ファイルが合計 18 見つかったのに対して，他の検索システムでは発見された類似ファイルが 2 つ以下にとどまった．

Logon

SPARS/R では 13 のファイルクローンが見つかったが，Koders と GCS ではファイルクローン 1 つのみが見つかった．

Rolodex

SPARS/R では類似ファイルが合計 11 見つかったのに対して，他の検索システムでは発見された類似ファイルが 2 つ以下にとどまった．

TypeVisitor

SPARS/R ではファイルクローン 1 つのみが見つかったが，GCS では 4，Koders では 13 のファイルクローンが見つかった．

以上の結果から，1 つのソースコード検索システムを利用するよりも，複数のソースコード検索システムを組み合わせて利用した方が多くの類似ファイルの情報を得られることがわかった．

以降では RQ3 について考察する．まず RQ3.1 について述べる．表 2, 5, 11 の結果から，ファイル名を用いたクエリの方が，ファイル中の単語を組み合わせて生成したクエリよりも再現率が高い傾向にあることがわかった．

全体として検出されたほとんどの類似ファイルがファイルクローンであることがわかる．また表 3, 6 の ApplicationListener に対する結果を比較すると，ファイルクローンではないが入力ファイルに対してコードクローンを含んでいるようなファイルを探す場合には，ファイル名を用いたクエリでは不十分であることがわかる．

逆に適合率に関しては，ファイル名を用いたクエリよりファイル中の単語を組み合わせて生成したクエリの方が高い傾向にある．

表 11 において，SPARS/R に対するクエリをコメント中に出現する単語で構成しても，ほとんどのケースで全く類似ファイルが検出できていない．SPARS/R がソースコードのランク付けに KR を用いており，コメントに対する索引付けがほとんど行われていないことが理由として挙げられる．

RQ3.2 について述べる．コード中から抽出した単語に対し，出現回数を考慮したランク付けを行う有用性はない．例えば SPARS/R の場合，Logon, Rolodex では表 5 の方が表 8 より良い結果が出ているが，ListEntriesRule に関しては表 8 の方が良い結果が出ている．つまり入力するファイルによって結果の良し悪しが一定しない．Koders に関しては表 7 より表 10 の方が結果が良い．GCS に関しては表 6 の方が表 9 より結果が良い．

RQ3.3 について述べる．クエリに含める単語の数をうまく設定しないと，検索結果として取得できるファイルの数に大きな開きが出ることがわかった．ただしほとんどの場合，1 つでも検索結果が見つかったなら，その後は単語の数を減らしても見つかる類似ファイルが増えなかった．

その他の考察について述べる．表 3, 6, 12 などを見る限り，GCS に対してクエリを入力した結果得られた類似ファイルが，他のソースコード検索システムに比べて少ないように見える．今回の実験では GCS からの検索結果が 30 ファイルを超えた時点で類似ファイルの検出を打ち切っているが，打ち切りの上限をより高く設定すればもっと多くの類似ファイルが得られた可能性がある．特に表 12 に関しては，クエリにコメント文中で汎用的に用いられる単語が多く含められてしまったために，膨大な検索結果が返ってきているものとみられる．

また，今回はファイルから抽出された単語に対する Camel Case の分割，Snake Case の分割を行っていないが，場合によっては入力されたファイルに特有の複合語が分割されてしまい，かえって結果が悪くなる可能性がある．

5　適用事例

5.1　適用実験

本節では，提案システムを用いて行った出典調査の事例を示す．出典調査では以下の作業を行うものとする：

1. システムへファイルを入力
2. システムからコードクローン情報を取得
3. Type A を入手元と確定
4. 各ファイルの更新日時を取得
5. 最も古いものを起源と確定

まず Java ソースコードのファイルとして，以下から得られたプロジェクトから選出したファイルを入力し，提案システムを用いて出典調査を試みた．詳細については後述する．

- apache
- eclipse
- sourceforge

また C ソースコードのファイルとして，以下のプロジェクトから選出したものについて，出典調査を試みた．C 言語のファイル調査には試作したシステムが利用できないため，本来システムが行う処理を手動で実行した．詳細については後述する．

- OpenJDK 7.0
- Mozilla に含まれているプロジェクト群

5.1.1　ChangePermissions.java

手元にあった ChangePermissions.java ファイルを提案システムへ入力したところ，図 8 のような結果が得られた．この時点で TypeA のコードクローン情報は入手元として特定される．

さらに得られた各ファイルの URL を調べ，更新日時を調査したところ，図 9 のような結果となった．

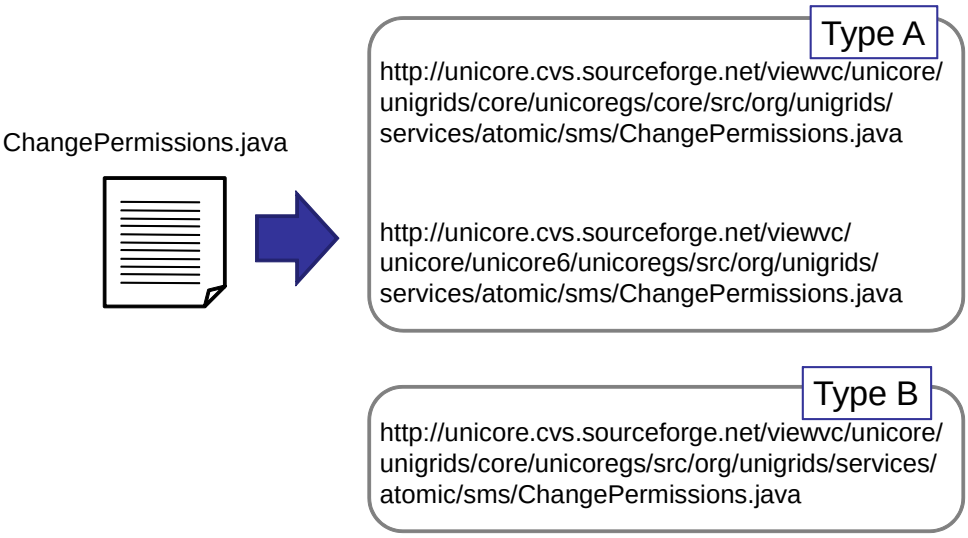

図 8: 適用例 1:システムへの入力と出力

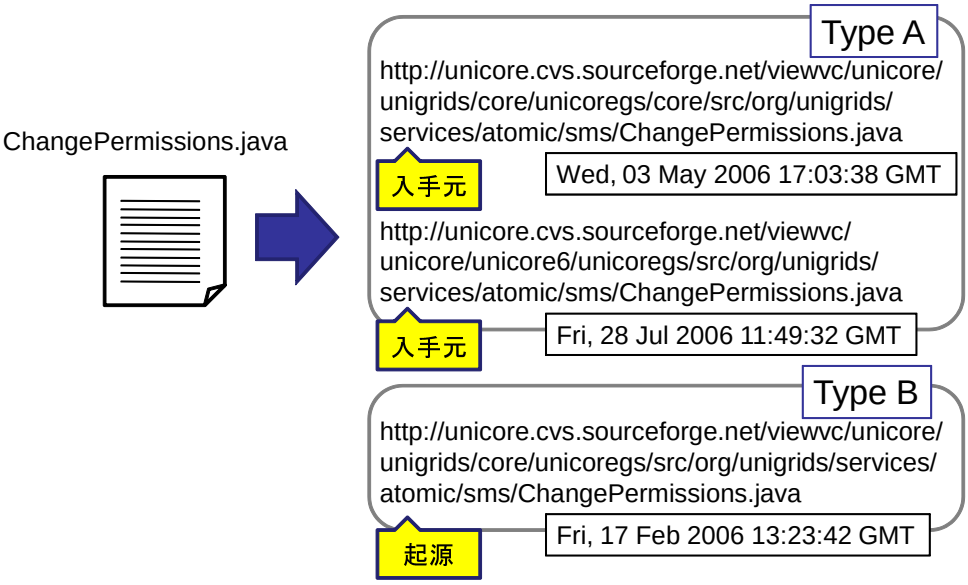

図 9: 適用例 1:入手元と起源の特定

特定した入手元と起源の情報を用いて追加調査を行った．図 10 に示すように，すべて unicore のプロジェクトであることがわかった．まず起源と特定されたファイルは unicoreGS 1.3 がリリースされたときに更新されたものと判明した．さらに，次に古いファイルは unicoreGS 1.4 がリリースされたときに更新されたものと判明した．最も新しいファイルについては，unicore6 プロジェクトが生成されたとき，unicore GS 1.4 からそのままコピーされたものと判明した．

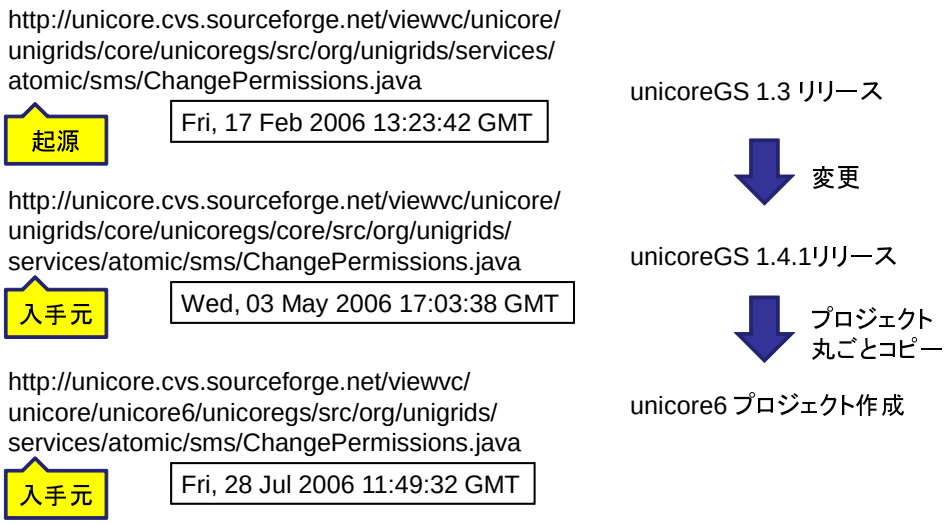

図 10: 適用例 1:追加調査により判明したファイルの生成過程

5.1.2 zutil.c

同様に，OpenJDK 7.0 に含まれていた zutil.c を入力として与えた結果，提案システムにより以下のプロジェクトが zutil.c と類似したファイルを有していると判明した．

- OpenJDK 7.0
- zlib
- dietpython
- cross-stuff
- nsnam
- gnuzilla

他にも多くのプロジェクトが関与していたがここでは省略する．

入手元は以下のプロジェクトから得られたファイルとなった．起源は zlib から得られたファイルとなった．

- OpenJDK 7.0
- zlib

- dietpython
- cross-stuff
- nsnam
- gnuzilla

さらに調査を行ったところ，いずれも zlib をライブラリとして用いているが，適用されているライセンスは以下のように異なるものだった．

- zlib License
- GPL v2.0

結果わかったこととして，zlib の公式プロジェクトから直接ファイルを引用すれば，zlib License もしくはそれに互換性のあるライセンスを適用できることが判明した．またこのソースコードの欠陥や最新版に関する情報も，zlib の公式プロジェクトを調べればわかることが判明した．

5.1.3 bnet.c

FreeSwitch のソースコードアーカイブに含まれていた bnet.c を入力として与えた結果，提案システムにより以下のプロジェクトが bnet.c と類似したファイルを有していると判明した．swiftweasel は有志により Mozilla Firefox を Linux 向けにビルドしたものである．

- Mozilla
- swiftweasel
- FreeSwitch

入手元は以下のプロジェクトから得られたファイルとなった．起源は Mozilla から得られたファイルとなった．

- swiftweasel
- FreeSwitch

さらに調査を行ったところ，適用されていたライセンスは以下の 2 通りであった．

- MPL 1.1/GPL 2.0/LGPL 2.1（トリプルライセンス）

- MPL 1.1

MPL 1.1/GPL 2.0/LGPL 2.1 は基本的に MPL を適用するライセンスだが，利用者によってそのソースコードのライセンスを GPL に変更することが許されている．

以上により，提案システムを用いて出典調査を行い，入力されたファイルに対する入手元と起源を特定できることが確認できた．さらに開発元やライセンスについての調査にも利用することができた．

5.2 その他適用可能性

以下で，提案システムがその他にどのようなケースで有用であるか記述する．

5.2.1 企業での委託開発

ある企業 C_1 が他社 C_2 へ開発を委託したと仮定する．ここで委託先 C_2 は OSS を利用してソースコード開発を行うことを決定したものとする．C_1 は委託先 C_2 からソースコードファイル Q を入手したものの，委託先 C_2 がどのように OSS を利用しているか不明である．よって Q をそのまま利用するには抵抗がある．

ここで Q を安全に利用するために，C_1 は提案システムを利用してその安全性を確認する．Q が安全であるか確認するための手順を以下に示す．

まず提案システムを利用し，ソースコード Q のコードクローン情報を取得する．これにより，Q の引用元である可能性のあるソースコードとそのコードクローン情報をすべて入手することができる．

次に，コードクローン情報を使って Q の安全性を確認する．

類似ソースコードを含むソフトウェア S について以下の項目を調べることで，欠陥に関する情報を得られる．

- メンテナンスは行われているか
- 致命的なセキュリティバグが見つかっていないか

検索システムへの登録日時が古い場合，メンテナンスが正しく行われているか，また最新版のソースコードが存在しないか確認する必要がある．

S の開発元は重要なステークホルダともなりうるため，S の開発元に連絡を取ることが可能か調べることも重要である．

また，Q に適用されているとまずいライセンスがあれば，Q の利用を取りやめることも考慮に入れる必要がある．

5.2.2 ソースコード検索システムを用いた再利用

ソースコード検索を用いて見つけたソースコード Q の再利用を行うことを想定する．

ソースコード検索を用いると，検索者にとって不慣れなソースコードが見つかることが多い．そこで，検索者は以下の情報を検索システムから入手する．前述したようにいずれも再利用時には重要な情報である．

- ソースコード Q を含むソフトウェア S
- 検索システムへの登録日時
- ライセンス

以上の情報があっても，Q がどのような経緯で作成されたかは不明であるため，そのまま利用するには抵抗がある．

そこで，検索されたソースコード Q を含むソフトウェア S について調べる．S の公式ページから以下の出典情報を入手する．

- 開発元はどこか
- そのソフトウェアがどのようなライブラリを用いているか

ソースコード Q は S の開発元がオリジナルで作成したものかわからない．よって安心して再利用するには，以上の情報だけではまだ不十分である．

そこで安全に再利用するために，提案したシステムを利用する．まず検索されたソースコード Q に対して，どういうソースコード（ソフトウェア）が関連をもっているか調べる．入手したコードクローン情報を用いて，Q のオリジナルになっていると問題があるものがないか調査する．

もしくは S が含むすべてのソースコードに対して類似ソースコードを検索する．入手したコードクローン情報を用いて，Q のオリジナルになっていると問題があるものがないか調査する．

6 関連研究

文献 [10] では，再利用を目的としたサンプルコードの収集システム PARSEWeb を提案している．PARSEWeb では手動でクエリ生成を行う必要があるが，本研究の提案システムではクエリ生成を自動で行うことができる．また評価実験では，特定のクエリに対して適切なソースコードを一つに絞り，得られた結果と，適切なソースコードがどの順位で表示されたかを示している．本研究では適切なソースコードが複数存在しており，また検索システム上でのランキングは考慮する必要がないため，ソースコード順位の評価は行っていない．

文献 [9] では，OSS のソースコードを検索するための索引に様々な情報を利用するシステム Sourcerer を提案されている．本研究ではコードクローンを含むファイル，類似ファイルを収集することができるが，Sourcerer で提案されている索引にはコードクローンに関する情報は含まれていない．

A-SCORE[13] では開発時における部品の自動推薦を目的としてソースコード検索システム SPARS/R を利用している．本研究と同じようにソースコードから抽出した単語用いてクエリを作成しているが，tf/idf アルゴリズムを用いて単語のランク付けを行っている点で本研究とは異なる．

文献 [12] では，ソースコード検索システム SPARS-J[14] の適合率評価を行うため，特定のクエリに対する検索結果上位 10 件に対して，対応のある平均値の差の検定を行い，他のシステムと SPARS-J の比較を行っている．本研究では検索結果のランキングを考慮しておらず，また特別な比較対象があったわけでもないため，このような検定は行っていない．

文献 [11] の評価実験では，特定のクエリに対して，利用しているソースコード検索システムが適切なソースコードをいくつ返すか評価実験を行っている．今回 4 節で行った実験でも，生成したクエリに対して検索システムが適切なソースコードをいくつ返すかということを調査している．ただし文献 [11] では対象を検索システム上で見つかった上位 5 件のソースコードに限定しているのに対し，本研究では見つかったソースコードすべてを対象としているので，より厳密な調査となっている．

7　まとめ

本研究では再利用時に必要な情報を調査するために必要な，コードクローン情報収集システムの試作を行った．多くのコードクローン情報を効率的に集めるため，Web 上のソースコード検索システムを用いた．

ソースコード検索システムから効率的に類似ファイルを集めるために，どのようなクエリを入力すればよいか実験を行い，クエリの生成方法に関する考察を行った．実験により，複数の検索システムを用いると，同じクエリでも様々な類似ファイルが見つかることが判明した．また全文検索を用いたソースコード検索システムから，少ない取得ファイル数で効率的に類似ファイルを集めるには，ファイル名を検索システムに入力するよりも，入力されたソースコード中に登場する単語を複数組み合わせてクエリを生成した方がよいことがわかった．また，ファイルクローンではないが入力ファイルに対してコードクローンを含んでいるようなファイルを探す場合にも，入力されたファイルのコード中に登場する単語を組み合わせて生成したクエリを用いた方が良いことがわかった．逆に，入力ファイルに対してできるだけ多くのファイルクローンを取得したい場合は，コード中に登場する単語を多く組み合わせたクエリよりも，ファイル名のみを用いてクエリを生成する方がよい結果が出ることがわかった．

さらに，試作したシステムを用いて，実際に出典調査を試みた．結果，入力したファイルの入手元，起源を特定することができた．さらに特定した入手元と起源を利用した調査により，入力したファイルが開発元でどのように再利用されていたかを知ることができた．

現段階で試作されているシステムでは自動化がコードクローン情報の収集にとどまっているため，今後はコードクローン情報から再利用に必要な情報を取得するところまで自動化されたシステムが必要になる．

謝辞

本研究を行うにあたり，常に適切な御指導，御鞭撻を賜りました 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 教授に深く御礼申し上げます.

本研究において，適切な御指導および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 松下 誠 准教授に深く感謝いたします.

本研究において，逐次適切な御指導および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 石尾 隆 助教に深く感謝いたします.

本研究を通して，随時適切な御指導および御助言を賜りました 東洋大学総合情報学部 早瀬 康裕 助教に心より深く感謝いたします.

FCFinder の開発とその評価に関して随時適切な御指導および御助言を賜りました 立命館大学情報理工学部情報システム学科 山本 哲男 准教授に心より深く感謝いたします.

評価実験の実施にあたり，適切な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 真鍋 雄貴 氏に深く感謝いたします

ツールの作成にあたり，有益な御助言を多数頂きました 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 伊達 浩典 氏に深く感謝いたします.

ソースコード検索システムとして利用させていただいている SPARS/R の開発者である 市井 誠 氏に心より感謝いたします.

CCFinder の開発者であるはこだて未来大学情報アーキテクチャ学科 神谷 年洋 准教授 に心より感謝いたします.

SPARS/R の前身である SPARS-J の開発に携わった方々に心より感謝いたします.

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いたします.

参考文献

[1] Black Duck Koders.com.
http://www.koders.com/.

[2] Google Code Search.
http://www.google.com/codesearch.

[3] SPARS/R.
http://demo.spars.info/r/.

[4] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.

[5] Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *29th IEEE International Conference on Software Engineering*, pp. 106–115, Minneapolis, MN, 2007.

[6] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *12th IEEE International Conference on Software Maintenance*, pp. 244–253, Monterey, CA, 1996.

[7] R. Koschke, et.al. *3rd International Workshop on Software Clones*, Kaiserslautern, Germany, 2009.

[8] Yusuke Sasaki, Tetsuo Yamamoto, Yasuhiro Hayase, and Katsuro Inoue. Finding file clones in freebsd ports collection. In *Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories*, pp. 102–105. ACM, May 2010.

[9] Bajracharya Sushil, Ossher Joel, and Cristina Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pp. 1–4, Washington, DC, USA, 2009. IEEE Computer Society.

[10] Suresh Thummalapenta and Tao Xie. PARSEWeb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pp. 204–213, 2007.

[11] 大須賀俊憲, 金子伸幸, 山本晋一郎, 小林隆志, 阿草清滋. ソフトウェア統合検索を利用した再利用支援システム. *JSSST FOSE 2009*, Vol. 24, No. 3, pp. 63–74, 2007.

[12] 梅森文彰, 西秀雄, 横森励士, 山本哲男, 松下誠, 楠本真二, 井上克郎. Java を対象としたソフトウェア部品検索システム SPARS-J の実験的評価. 信学技法, Vol. 103, No. 708, pp. 19–24, 2004.

[13] 島田隆次, 市井誠, 早瀬康裕, 松下誠, 井上克郎. 開発中のソースコードに基づくソフトウェア部品の自動推薦システム A-SCORE. 情報処理学会論文誌, Vol. 50, No. 12, pp. 3095–3107, 2009-12-15.

[14] 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, 情報・システム, I-情報処理, Vol. 87, No. 12, pp. 1060–1068, 2004-12-01.