特徴語抽出を用いた テストケース-モジュール間対応抽出方法

遠藤侑介[†1] 酒井政裕[†1] 今井健男[†1] 岩政幹人[†1]
福元康文[†2] 一條泰男[†2]

運用中のソフトウェアでは，運用時の修正や派生開発によって，設計仕様書に書かれた各機能がどのモジュールで実現されているかが不明確になっていることが散見される．我々は，テスト被覆を介して設計仕様書とモジュール間の対応を復元する手法を提案する．本手法は，設計仕様書とテストケースの対応が比較的維持されているという前提に立ち，テストケースを「文書」，そのテストで実行されたモジュールの集合を「単語」の集合とみなして，自然言語処理に用いられる特徴語抽出手法である TF-IDF を適用し，テストケースと，各々のテストケースの「テスト対象」のモジュールとの対応関係を抽出する．さらに我々は，本手法のプロトタイプ *eacov* を実装し，2 件のプロジェクトに対して適用実験を行い，本手法によってテストケースごとの「テスト対象」のモジュールを正しく抽出できることを確認した．

Extracting a relation between test cases and modules by using important word analysis

YUSUKE ENDOH[†1] MASAHIRO SAKAI[†1]
TAKEO IMAI[†1] MIKITO IWAMASA[†1]
FUKUMOTO YASUFUMI[†2] YASUO ICHIJO[†2]

It is important for software maintenance to clarify the correspondence between each feature in specifications and program modules implementing them. However, customizing software after deployment or revising software in derivative development often blurs or even breaks the correspondence. Assuming that the correspondence between design specification and test programs is well maintained, we propose a new method that restores the correspondence between test programs and their “target” modules by using test coverage. This method uses TF-IDF, an important word analysis popularly used in natural language processing, by regarding test programs as "documents" and modules run by each test as "words". In addition, we implemented a prototype called *eacov*. By applying it to two projects, we confirmed that our method could certainly extract the “target” module of each test.

1. はじめに

ソフトウェア開発においては，要求と設計，設計と実装など，様々な段階でトレーサビリティを管理することが重要である．しかし運用・メンテナンス段階に入ったソフトウェアでは，運用時の調整やアドホックな修正，その後の派生開発などを経ていくうちに，設計仕様書と実際のプログラムコードの構造が乖離することがある．このような状況では，設計書に書かれた各機能がどのモジュールで実現されているかわかりにくいため，継続的なメンテナンスや派生開発が困難になりがちである．

設計とテストスイートの対応，つまり，設計仕様書に書かれた個々の機能とテストケースの対応は，経験的には，テスト仕様書によって対応が比較的維持されやすい．よってテストケースと，各々のテストケースがテストすることを意図したモジュールとの対応関係を抽出できれば，設計書とモジュールの対応関係を抽出できると考えられる．

そこで我々は**テスト被覆**に注目して，テストケースの**テスト対象**を自動的に推定する手法を提案する（図 1）．テスト被覆とは，テストケースによって実行されるモジュールの集合である[a]．テスト対象とは，テストケースの作成者がテストしようとしていた，モジュールである．

テスト被覆とテスト対象は同一ではないことに注意されたい．テストケースの実行ではテスト対象が直接または間接的に実行されるが，テストケースは実行しているモジュールすべてをテストする意図があるとは限らない．つまり，テスト対象はテスト被覆の部分集合であると考えられる．我々は，テスト被覆からテスト対象と思われる部分のみを推定・抽出するために，自然言語処理の特徴語抽出の手法である TF-IDF を応用することを提案する．また我々は，本提案手法のプロトタイプ eacov を開発し，当社の製品コードとオープンソースのプロジェクトを題材として適用実験を行った．これにより，提案方法が適切にテスト対象を推定・抽出できることを確認した．

本論文の残りの構成は以下のとおりである．2 節で提案手法とプロトタイプ実装について，3 節で適用実験につい

†1 (株)東芝 研究開発センター
Corporate Research & Development Center, Toshiba Corporation
†2 (株)東芝 社会インフラシステム社
Social Infrastructure Systems Company, Toshiba Corporation

a) テストカバレッジ（テストによって実行されたコードの割合）とは異なることに注意されたい．

て説明する．4 節で関連研究と比較し，5 節でまとめる．

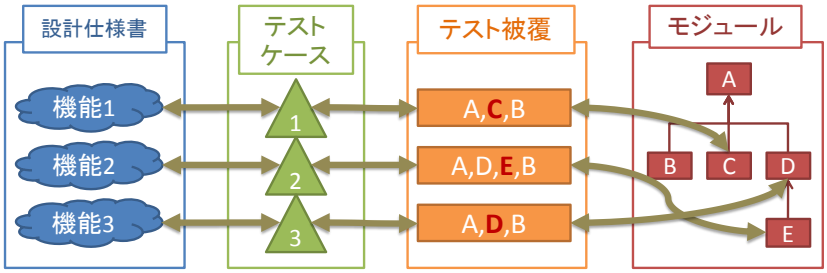

図 1 テストケース・テスト被覆を介した設計⇔モジュール間対応設計

2. 提案手法

2.1 テスト対象の性質

提案手法を説明する前に，テスト対象の性質について考える．

多くのテストケースが実行するモジュールは，補助関数や初期化・後始末の処理などであると考えられる．このようなモジュールは，テスト対象を実行するために間接的に実行されているに過ぎず，それ自身がテスト対象であるとは通常は考えにくい（ただしそのモジュールだけがテスト被覆に含まれるような場合は，そのモジュールがテスト対象であると考えられる）．

我々はこの直観に基づいて，他のテストケースのテスト被覆にあまり現れず，特定のテストケースだけが実行するモジュールは，テスト対象である可能性が高いと仮定した．

この仮定に基づき，図 1 の例についてテスト対象を非形式的に推定してみよう．モジュール A と B は全テスト被覆に現れる．よってこれらは補助関数や初期化・後始末処理など，テスト対象を実行するために間接的に必要になっていると考えられ，それ自身がいずれかのテストケースのテスト対象であるとは考えにくい．モジュール C はテストケース 1 のみが実行するので，テストケース 1 のテスト対象であると推定される．モジュール D はテストケース 2 と 3 が実行するが，テストケース 2 はさらにモジュール E も実行している．このことから，テストケース 2 にとって，モジュール E は D より特徴的であり，テスト対象である可能性が高いと推定される．以上より，テストケース 1 はモジュール C がテスト対象であり，テストケース 2 はモジュール E がテスト対象であり，テストケース 3 はモジュール D がテスト対象であると推定できる．

2.2 特徴語抽出を用いたテスト対象の抽出

自然言語処理において，文書が含む単語の内，その文書を特徴づける単語を特徴語と呼ぶ．特徴語を機械的に推定することは，情報検索や文章要約などの目的で重要である．

特徴語を抽出する著名な推定手法に TF-IDF[1]がある．TF-IDF は，他の文書中にあまり出現しないが特定の文書で頻出する単語はその文書を特徴づけるという仮定に基づいて，文書中の各単語にスコアを付ける．これは，テスト対象の性質に関する我々の仮定によく似ている．tfidf(t, d) は，文書 d 中の単語 t のスコアであり，以下の式で表現される．

$$tfidf(t,d) = \frac{tf(t,d)}{df(t)}$$

ただし tf(t, d) は，文書 d 中における単語 t の出現回数を文書 d 中の単語総数で割った値であり，df(t) は，単語 t が出現する文書数を，全文書数で割った値である．

我々は，テストケースごとのテスト被覆を「文書」，テスト被覆に含まれるモジュールの集合を「単語」の集合とみなして TF-IDF を適用し，モジュールがテストケースを特徴づけるスコアを計算する方法を提案する．その中でスコアの高いものをテスト対象として出力する．（図 2）

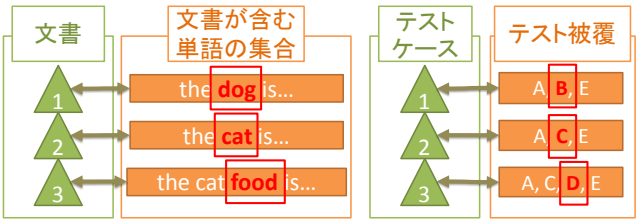

図 2 提案手法の考え方

2.3 プロトタイプ実装

我々は提案手法のプロトタイプ実装 eacov を開発した．eacov は C 言語のヘッダファイル 1 つと，プログラム実行を監視するスクリプト 1 つから構成される．

図 3 に eacov の動作を示す．まずユーザは，テストプログラムを書き換えて eacov のヘッダファイルをインクルードし，テストケースごとに簡単な書き換えを行う（この書き換えは後述する）．そして eacov のスクリプトを実行することで，gcc[2]に付属するカバレッジ測定ツール gcov の上でテストプログラムの実行を開始する（図 3 の①）．eacov のヘッダファイルはテストケースの実行の開始・終了ごとにテストプログラムの実行を中断し，現時点のテスト被覆を出力[b]し，eacov のスクリプトに通知を行う（図 3 の②）．その通知を受けた eacov のスクリプトはテストプログラムからの通知を待ち受け，現時点のテスト被覆を集計して保

b) gcov に現時点のカバレッジを出力させる API である__gcov_flush()を用いる．

存し，テストプログラムの実行を再開する（図 3 の③）．　②と③を繰り返して全てのテスト被覆の実行が完了したら，

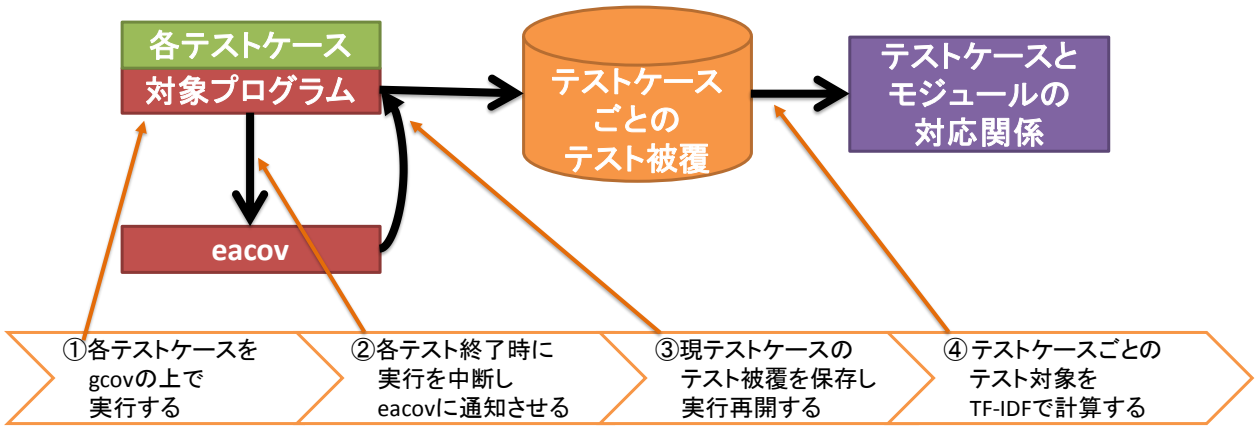

図 3 プロトタイプ実装 eacov の動作

```
1: int add(int x, y) {
2:   return x + y;
3: }
4: int mul(int x, y) {
5:   int r=0;
6:   for(; x>0; x--)
7:     r = add(r, y);
8:  return y;
9: }
```

```
#include "CUnit/Basic.h"
#include "eacov.h"
void test_add(void) {
  EACOV_START("test_add");
  CU_ASSERT(9 == add(5, 4));
  EACOV_END();
}
void test_mul(void) {
  EACOV_START("test_mul");
  CU_ASSERT(20 == mul(5, 4));
  EACOV_END();
}
```

図 4 足し算と掛け算の関数を提供するサンプルプログラム（左）とそのテストケース（右）

集計されたテストケースごとのテスト被覆に対して TF-IDF を適用し，テストケースごとにテスト対象と思われるモジュールの上位 3 件を出力する（図 3 の④）．

2.4 動作例

図 4 に示すサンプルプログラムを題材として，eacov の動作を説明する．このプログラムは足し算を行う関数 add と掛け算を行う関数 mul を提供する．関数 mul は内部的に関数 add を呼び出す．2 関数に対し，それぞれ test_add と test_mul という CUnit で書かれたテストケースがある．

ユーザは，eacov を用いるためにまず，テストケースに対して次の 3 つの変更を行う必要がある．

1. ファイル先頭で eacov のヘッダファイル eacov.h をインクルードする．
2. 各テスト開始時に EACOV_START(テスト名)というマクロ呼び出しを追加する．
3. 各テスト終了時に EACOV_END()というマクロ呼び出しを追加する．

図 4 中の強調している行がこの変更を示している．

次にこのテストプログラムを，eacov のスクリプトの上で実行する．スクリプトは，特定のファイルディスクリプタを開いた状態でテストプログラムを起動する．eacov のヘッダファイルは，これを通じてスクリプトと通信を行う．

以上によってテストケースごとのテスト被覆が表 1 のように得られる．それに対し TF-IDF を適用した結果が表 2 であり，eacov は最終的に表 3 の推定結果を出力する．

1 行目と 2 行目は関数 add であり，4 行目から 8 行目は関数 mul である．すなわち，test_add では関数 add が，test_mul では mul が最も特徴的という結果が得られた．従って，この例では正しく対応関係が抽出されたといえる．

2.5 プロトタイプの制限

eacov では，ソースコードの各行の単位をモジュールとして扱っている．これは gcov のカバレッジ測定単位をそのまま使っているためである．構文解析を行って関数単位やクラス単位など一般的なモジュールの単位に還元するには，

ある単位のどの程度の行数が実行されたらその単位をテスト被覆に含めるかについて検討する必要がある.

また、本プロトタイプは本手法の有効性を確認する目的で作成したため、eacov の測定や解析の動作速度には注力していない。

表 1 テストケースごとのテスト被覆

テストケース	テスト被覆（行番号）
test_add	1,2
test_mul	1,2,4,5,6,7,8

表 2 テストケースごとの各行の TF-IDF スコア

テストケース	1	2	4	5	6	7	8
test_add	0.00	0.00	N/A	N/A	N/A	N/A	N/A
test_mul	0.00	0.00	0.04	0.04	0.04	0.04	0.04

表 3 テストケースとテスト対象

テストケース	テスト対象
test_add	1,2
test_mul	4,5,6,7,8

3. 評価実験

我々は，本手法が実際に有用に動作することを確かめるため，当社製品部が開発した製品コードと，プログラミング言語 Ruby の実装とを用いて，適用実験を行った.

これら 2 つを実験題材として選んだ理由は評価の容易さである．これらのテストケースは，どちらも単体テストの粒度で書かれており，テストケースの名前やコメントの記述から，テスト作成者がどのモジュールをテスト対象として意図していたかが比較的容易にわかるものであった．このため，推定結果の評価（本手法によってテスト対象と推定されたモジュールが，テスト作成者が意図していた実際のテスト対象と一致しているかどうか)を行いやすかった.

ただし，本実験は非常に予備的であり，定量的な評価とは言い難いことをあらかじめ断わっておく．それでも我々は, 本手法の可能性を感じさせる結果であると考えている.

3.1 製品部コードへの適用実験

まず, 当社の製品部が開発した製品コードに対して eacov を適用した．製品の詳細については掲載できず，識別子やファイル名は伏せ字で掲載する.

ソースコード本体の規模は約 14 万行，テストケースは約 3 万行である. テストケースの数は約 300 件である. eacov の下で全テストケースを実行したところ，測定に 100 分程度を要した. eacov なしでテストケースを走らせる場合は 1 秒程度で終わるため、実用には高速化を行う必要がある.

得られた推定結果の一部を表 4 に示す.

XXX_TestCase_1 のテスト対象は，ソースファイル XXX.c の 311 行目が第一候補であり, 続いて XXX.c の 280 行目， XXX.c に 275 行目を挙げている.

表 4 当社製品部コードに対する eacov 測定結果（抜粋）

テスト関数名		ファイル	番号	スコア
XXX_TestCase_1	1	XXX.c	311	346.2
	2	XXX.c	280	235.0
	3	XXX.c	275	203.3

```
228: LONG XXXMain(
273:   if(XXX
274:     == XXX &&
275:      XXX
276:     == XXX)
278:     {
280:         YYY;
281:     }
```

```
197: /*
200: テスト対象関数:    XXX
210: */
211: void XXX_TestCase_1()
212: {
213:    EACOV_START("XXX_TestCase_1");
249:    /* 関数実行*/
250:    ulResultValue = XXX(……);
252: /*
255: ＜期待する結果＞
256: ・正常終了すること
257: ・YYY となること */
276:   EACOV_END();
277: }
```

図 5 プログラム（上）とテストコード（下）の抜粋

図 5 の下図はテストケース XXX_TestCase_1 の抜粋である．このように，このソースコードは，テストケースごとにテスト対象や期待している挙動がコメントとして詳細に記述されているため，これを正しいテスト対象として扱う．この場合，このテストケース XXX_TestCase_1 がテスト対象としている関数は XXX である（200 行目のコメント）．また YYY が実行されることを期待している（255-257 行目のコメント）. 図 5 の上図はソースファイル XXX.c の 311 行目近辺を抜粋したものである. この関数 XXXMain は, テスト対象として指定されている関数 XXX の名前に Main

を付加した名前であり，テスト対象の関数の実装本体が正しく抽出されたと考えられる．さらにテスト対象の第 2 候補であった 280 行目は, まさに YYY を実行する行であった．

同様に 10 件のテストケースをサンプリング検証することで，eacov がテスト対象を適切に抽出していることが確認できた．

3.2 Ruby への適用実験

次に，プログラミング言語 Ruby の実装に対して eacov を適用する実験を行った．テストケースには Array クラスのテストである test/ruby/test_array.rb を用いた。ruby-1.9.2-p180 のソースコード及びテストコードを用いた．

その測定結果を表 5 に示す。

表 5 Ruby の Array のテストに対する eacov 測定結果(抜粋)

テスト名		ファイル	番号	スコア
test_reverse!	1	array.c	1813	1.60
	1	array.c	1815	1.60
	3	array.c	2142	0.58
test_map!	1	array.c	2196	1.52
	2	array.c	2200	1.52
	2	array.c	2197	1.52

テストケース test_reverse!は，配列の順序を逆転させるメソッド Array#reverse!のテストである．このテストケースのテスト対象として推定された array.c の 1813 行目は，C 言語の関数 rb_ary_reverse_bang の定義の先頭であった（図 6）．この関数の直前のコメントから，この関数が Array#reverse!メソッドの実装であることがわかる．よって，テストケース test_reverse!のテスト対象を正しく抽出できたことがわかる．

```
1801: /*
1802:  *  call-seq:
1803:  *     ary.reverse!   -> ary
1804:  *
1805:  *  Reverses +self+ in place.
1806:  *
…
1810:  */
1811:
1812: static VALUE
1813: rb_ary_reverse_bang(VALUE ary)
1814: {
1815:     return rb_ary_reverse(ary);
1816: }
```

図 6 Ruby の関数 rb_ary_reverse_bang の前後(抜粋)

同様の検査を 10 件サンプリングして行ったところ, すべて正しい推定であると確認できた．

3.3 考察

本手法によって，テストケースごとのテスト対象を正確に抽出できることを簡単なサンプリング検査によって確認した．ただし統計的に有意なサンプリング量での検査にはなっているかどうかの検証は行っていないため，今後さらに実験を行う必要がある．

実用のためには測定の速度の改善が課題であることがわかった. テスト被覆集計ごとに gcov プロセスを実行するためで，実装の改良で改善できると考えている．

モジュールの単位としてソースコードの行を採用したのは，gcov の測定単位をそのまま流用することでプロトタイプ実装を容易にすることが目的であった。しかし 3.1 節の実験では, テストケースに書かれた期待が if 文の then 節の中でのみ実行されるものであり，行をモジュール単位と扱ったおかげでこの位置を正確に同定することができた．そのため，リグレッションテストの削減やテスト不足の指標として本手法を活用する場合には，この粒度でも有用である可能性を示唆している．しかし継続的インテグレーションのエラー通知やコードリーディングの補助資料としては明らかに詳細すぎるため，通常のモジュール単位である関数単位，クラス単位，ファイル単位も実用的にはサポートしていく必要があると考えている．

4. 関連研究

テストケースとモジュールの間の対応関係を自動的に抽出する既存研究は多数存在する．それらの多くの研究の動機は，レグレッションテストの工数を減らすため，ソースコードの変更に対して本当に再試が必要なテストケースの集合を同定することである. 中でも Cibulski ら[3]は,「多くのテストにカバーされているモジュールはテスト対象とは考えにくい」という直観に基づいてリグレッションテストの同定を行うという点で, 本手法に非常に近い. 彼らは, モジュールごとに，そのモジュールを実行するテストケースの個数の逆数を重み付けとして定義している（inverse coverage という）．一方本手法は，単純な逆数ではなく，他のテストケースのテスト被覆と比較して特徴的なモジュールを TF-IDF によって抽出する点が異なる．

ソフトウェア工学の分野において, TF-IDF などの自然言語処理を活用すること自体は珍しくない．例えば大場ら[4]は，プログラム理解を支援する目的で，プログラム中の識別子からキーワードを抽出するための TF-IDF の変種を提案している．また Zhang[5]は，テストケース生成を目的として TF-IDF を用いている．メソッドを文章，そのメソッドが読み書きするフィールドを単語と見なして TF-IDF を適用し，メソッドごとに特徴的なフィールドを抽出したテ

ーブルを作成する. 興味深いことに, 前述の Cibulski らも, inverse coverage だけでなく，コメントや識別子に対して TF-IDF を用いて, テストケースとモジュールの間の対応関係を推定する手法もあわせ持つハイブリッドなアプローチとなっている．しかし本手法のように，テスト被覆に対して TF-IDF を適用することはしていない．

本研究は著者らの既存研究[6]に対して，実験とその考察を追加した内容である．

5. おわりに

テストカバレッジ測定ツールで得られるテスト被覆に対して，特徴語抽出の手法である TF-IDF を適用することで，テストケースとモジュールの間の対応関係を抽出する手法を提案した．単体テストが付属しているプロジェクトのソースコードを題材として本手法の適用実験を行い，有効に動作することと，推定精度に期待ができることを確認した．

今後の展望としては，より本格的な実験を行うことが挙げられる．まず，本手法の推定精度を統計的に評価し，他関連研究の手法と比較する必要がある．また，結合テストの粒度で書かれたテストケースに対して適用した場合に有用な推定結果を得られるかどうかも検討すべきであると考えている．

また，実験にあわせて，本手法の実装を改良していくことも挙げられる．まず推定速度の改善が必要である．テストケースごとのテスト被覆を抽出するために，テストケースの実行ごとに gcov が内部的に記録している実行ログを外部ファイルに出力し, そのファイルに対して gcov のツールを外部プロセスとして呼び出し，可読のテキスト形式を得ているのが明らかなボトルネックであるため，内部的に記録している実行ログから直接テスト被覆を取り出すようにすれば大幅に高速化すると考えている．また，現在は gcov の測定単位であるソースコードの行をモジュールの単位として用いているが，関数単位やクラス単位，ファイル単位などより大きな粒度で測定できるように改良することも考えている．

さらに，本手法を用いて得られたテストケースとモジュールの対応関係を用いて，実際に継続的なメンテナンスや派生開発へ活用していくことも挙げられる．具体的には，以下のような応用を想定している．

（1）リグレッションテストの工数を削減する．メンテナンスにおいて，バグ修正などソースコードの変更がリグレッションを起こさないことを確かめるためにリグレッションテストを行うが，このテストに掛かる工数を削減するため，実行するテストを減らす必要がある．本手法によって，ソースコードの変更箇所がテスト対象であるテストケースのみを選択するという用途が考えられる．

（2）テスト不足を検出する新たな指標とする．例えば，いずれのテストケースのテスト対象にもなっていないモジュールを，テストがカバーできていない箇所として報告する．いずれかのテストケースが実行しているだけでは，カバーされているとは限らないに注意せよ．共通のユーティリティモジュールや，何らかの処理のためのプロローグ処理は，他のモジュールのついでに実行するのではなく，その箇所自体をテスト対象とするテストケースを書かなければカバーされたとは見なされない．

（3）継続的インテグレーションのエラーから自動的に担当者へ報告する. 通常の継続的インテグレーションでは, テストでエラーが起きる状態になった際に誰かに通知を行う（例えば開発者全員宛て）が，その通知内容を誰が対応すべきかについては関知しなかった．本手法で得られた推定結果を用いれば，エラーを起こしたテストケースから，そのテストケースに対応するモジュールを同定し，そのモジュールの開発担当者へ個別に通知を行うことができる．

（4）コードリーディングの補助資料とする．派生開発の際，既存のモジュールを再利用したいがドキュメントがない場合, そのモジュールのソースコードを直接読んだり, そのモジュールを使用しているソースコードやテストケースを探して読んだりして，モジュールの使用方法を調べる必要があった．本手法で得られた判定結果を持ち入れば，ソースコードを探す手間なしで，そのモジュールを使用しているテストケースをすぐに引くことができる．

謝辞　本研究を進めるにあたり，数多くの有益なアドバイスを頂いた東芝 研究開発センターシステム技術ラボラトリーの皆様と，実験用のソースコードを提供頂いた東芝社会インフラシステム社に感謝する．

参考文献

1 Karen Sparck Jones. *A statistical interpretation of term specificity and its application in retrieval*. Journal of Documentation, Vol. 28, No. 1. (1972), pp. 11-21.
2 *GCC, the GNU Compiler Collection*. http://gcc.gnu.org/
3 Hagai Cibulski, Amiram Yehudai: *Regression Test Selection Techniques for Test-Driven Development*, Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW '11), pp. 115-124
4 大場勝，権藤克彦：*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*, 情報処理学会論文誌 Vol 48, No.8, pp.2596-2607
5 Sai Zhang, David Saff, Yingyi Bu, Michael D. Ernst: *Combined Static and Dynamic Automated Test Generation*, Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11), pp.353-363
6 遠藤侑介，酒井政裕，今井健男，岩政幹人：*Eacov:特徴語抽出を用いた仕様-コード間対応復元*, 情報処理学会ソフトウェアエンジニアリングシンポジウム 2011