Vol.2010-SE-167 No.29
2010/3/19

ソフトウェア部品利用例抽出のための データフロー解析手法の提案と評価

柳　慶吾[†1]　石尾　隆[†1]　井上克郎[†1]

大規模なソフトウェアの開発では，ソフトウェアコンポーネントを正確に接続することが重要となる．開発者は，コンポーネントの接続方法を既存のプログラムにおける実際の利用例から学習することが多いが，特定のコンポーネントの利用例として適切なソースコードを抽出することは難しい．

そこで，本研究では Java プログラムのデータフロー情報からメソッドの接続関係を抽出し，メソッドの利用例を分析する手法を提案する．

A Data-Flow Analysis Approach to Extract Software Component Usage

Keigo Ryu,[†1] Takashi Ishio[†1] and Katsuro Inoue [†1]

Accurate connections between software components are crucial to develop a large-scale software. Though developers often learn component usage from real instances in existing programs, extracted source code may be inappropriate for usage examples of the components.

This study propose a data-flow analysis approach to extracting data-flow paths among methods as usage examples in Java programs.

1. まえがき

大規模なソフトウェアの開発では，ソフトウェアコンポーネントを正確に接続することが重要となる．ソフトウェアの設計段階で，ソフトウェアの個々の機能を担当するコンポーネントへと分解しておき，各コンポーネントに十分なテストを施したうえで接続することによって，高品質なソフトウェアを効率よく開発することが可能となる．しかし，大規模ソフトウェアの開発では，新規開発するコンポーネントと再利用コンポーネントを多数組み合わせて使用するため，あるコンポーネントをどこで利用すればよいか，すなわちどのコンポーネントと接続すればよいかを理解することは困難になる．開発者にとって未知のコンポーネントを使用する場合，開発者は既存のプログラムからコンポーネントの実際の利用例を調べ，そのコンポーネントをどのコンポーネントと接続すればよいかを理解することが多い．しかし，コンポーネントの利用方法が多岐に渡る場合，そのコンポーネントの利用例として適切なソースコードを特定することは難しく[7]，また，コンポーネントの利用環境や，手順に関する制限，同時に利用するべきコンポーネントといった情報を得られないことが多い．

そこで本研究では，Java プログラムのデータフロー情報からメソッドの接続関係を自動的に抽出し，接続関係からメソッドの適切な利用例に関する情報を分析する手法を提案する．具体的には，プログラムから変数間データフローグラフを構築し，このグラフからメソッド・コンストラクタの戻り値がどのメソッドのオブジェクトとなっているか，あるいはどのメソッド・コンストラクタの引数になっているかといったメソッド接続関係を抽出する．本手法では，プログラムから抽出されたメソッド接続関係の出現回数に着目し，あるコンストラクタによって生成されたオブジェクトに対してどのメソッドが使用されているか，あるいはあるメソッドの戻り値があるメソッドの引数になっていることが多いといった情報から，メソッドの適切な利用例を分析し，その結果を利用して開発者のプログラム理解とコンポーネントの再利用を促進することを目指す．

提案手法を実装したシステムを，39 種類の様々な用途のソフトウェアに対して適用した結果，総数 219,889 個のメソッド接続関係が抽出された．これらの接続関係のうち，出現回数が多いものほど信頼性が高いと考え，出現回数の上位 2,000 個の接続関係に対して分析を行った結果，24 組のメソッドの利用例を抽出することができた．抽出された 24 組のメソッドは，いずれも 1 つのまとまった単位で利用することが多いメソッドの組で，中にはプログラム上で複数のメソッドに渡って利用されているものや，利用順序・条件がそれぞれ異なるようなものも含まれていた．

2. 背　景

ある程度の規模のソフトウェアを開発する場合，ソフトウェアをその機能ごとにコンポーネントとして分割しておき，個別に開発，テストした後に接続することが一般的である．コ

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

ンポーネントは，ソフトウェアの目的に応じて新たに作成される場合もあるし，過去に開発されたソフトウェアから再利用される場合もある．汎用的な機能を持ち，多数のソフトウェアで再利用することが可能なコンポーネント群は，ライブラリとして広く利用されている．コンポーネントの粒度の単位は様々で，Java の場合はクラスあるいはパッケージ単位がしばしば用いられるが，本研究ではクラスを単位として考える．また，コンポーネントを他のコンポーネントから利用する方法については，メソッド呼び出しによるものを対象とする．

コンポーネントを適切に利用するためには，その利用方法を正しく理解することが重要となる．コンポーネントの開発者は，通常，そのコンポーネントの利用方法をドキュメントあるいはソースコードとして提供するが，コンポーネントの利用方法が多岐に渡る場合，その利用方法として適切な情報を提供することは難しい[7]．

あるコンポーネントの利用方法は，Feilkas らによれば，次の 4 種類の制約から構成される[3]．

システムあるいはコンポーネントの状態． ある状態のときにしか使用してはならない機能がある．たとえば，スタックに対する要素の取り出し (pop) 操作は，そのスタックが空でないときにのみ実行可能である．

引数や戻り値の使用方法． 引数や戻り値に対する何らかの処理が必要である．たとえば，メモリの確保のように失敗する可能性のある操作の成否を論理型の戻り値として返されている場合は，その戻り値を使った適切な条件分岐が必要である．

同時に使用すべき操作． ある操作 X を実行するときに必ず別の操作 Y を実行することが要求される．たとえばファイルに対する `open` と `close` 操作の組はこれに該当する．

外部からの利用を想定しない機能． ライブラリ内部からのみ使用される予定で作成されたが，プログラミング言語機構の制限などで外部からも呼び出し可能になっている操作のことを指す．たとえば，Eclipse において `internal` パッケージに属するクラスは，他のソフトウェアから使用されることは想定していない．

このような制約情報を抽出し開発者に提供する手法としては，様々な手法が研究されている．まず，システムあるいはコンポーネントの状態に関する制約に関しては，メソッドの呼び出し順序によって状態が変化すると考え，呼び出し列に対する制約を半順序関係として抽出する手法[1] や，CTL による表現として抽出する手法[9] が提案されている．これらの手法は，呼び出し列についての制約を抽出することから，Specification Mining あるいは Protocol Mining とも呼ばれている．コンポーネントの引数や戻り値の使用方法を調査する手法として，Nguyen らは，呼び出し順序関係とデータ依存関係を用いて複数のオブジェクトに対する呼び出し順序制約を抽出する手法[6] を提案している．また，プログラムスライシング[10] によって引数や戻り値の使用方法を調査することも可能である．本研究におけるデータフロー情報の抽出もこの領域に属しており，プログラムスライシングに用いられるプログラム依存グラフ[4] をデータフロー解析に特化させたグラフ構造を用いて，グラフ探索によってデータフロー情報を抽出している．Nguyen らの手法が手続き単位（Java におけるメソッド単位）でデータ依存関係を抽出するのに対し，本研究では，複数の手続きにまたがるデータ依存関係を抽出している点が異なる．なお，プログラムスライシングにおいてデータフロー情報だけを用いるという発想そのものは，動的解析で過去に適用されている[11] が，静的解析には適用されていない．

同時に使用すべき操作の抽出については，該当するメソッドの呼び出しが同時にソースコードに追加されることが多いという特徴を利用して，開発履歴から抽出する手法が提案されている[5] ．この制約の抽出については，利用例を抽出するというよりも，欠陥の検出を目的としている．本研究でのデータフロー抽出手法は，1 つのオブジェクトに対して同時に使用するメソッドの集合を取り出すことができるが，常に同時に使わなくてはならないという制約は，自動では抽出していない．

利用制約の情報を開発者に提示する方法としては，メソッドのドキュメントに書かれた制約を，そのメソッドを使用しているソースコード上で強調表示する手法[2] が提案されている．

外部からの利用を想定しない機能が外部から呼び出される問題に関しては，言語機構の制限によるところが大きい．Java では，JDK 1.7 から super package という概念を導入し，パッケージとは異なる「モジュール」という単位でのアクセス制限が可能となる[8]．

3. メソッド接続関係の抽出

本研究では，プログラムからコンポーネントの適切な利用例を分析するために必要なメソッドの接続関係を抽出する手法を提案する．提案手法は，メソッドの接続関係を抽出するために，変数間データフローグラフを構築し，グラフ上での経路探索を行う．

3.1 変数間データフローグラフ

変数間データフローグラフ (IVDFG, Inter-Variable Data Flow Graph) は，変数間のデータの流れを表した有向グラフで，プログラム依存グラフを変数に着目して簡略化したものである．**図 1** に IVDFG の例を示す．IVDFG の頂点は変数頂点，演算頂点，条件頂点の 3 種類である．変数頂点は，プログラム中のローカル変数の宣言とメソッド・コンス

Vol.2010-SE-167 No.29
2010/3/19

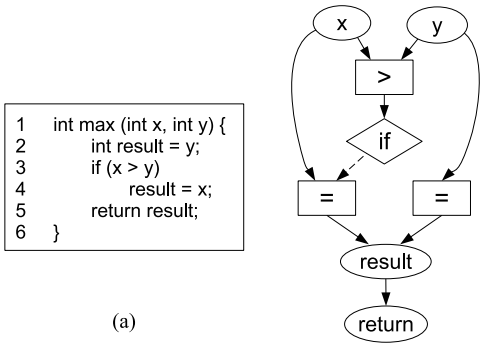

図 1 IVDFG の例，(a) メソッド max のプログラム，(b) (a) の IVDFG
Fig. 1 Example of IVDFG, (a) program of method max, (b) IVDFG of (a)

トラクタの仮引数，戻り値に対して 1 つずつ生成され，IVDFG を図示する際は楕円形の頂点で表す．図 1 (a) のメソッド max に対しては，変数として仮引数 x と y，ローカル変数 result の 3 つが存在するため，図 1 (b) の IVDFG ではそれぞれに対応する変数頂点が合計 3 つ生成される．また，メソッド max は戻り値を持つため，“return” 頂点を生成する．演算頂点は，プログラム中の演算 1 つに対して 1 つ生成される頂点で，図示する際は長方形で表す．図 1 (a) のプログラムでは，演算子は 3 つ存在するため，図 1 (b) の IVDFG では演算頂点は 3 つ生成される．条件頂点は，if 文や for 文などの制御文に対して 1 つ生成される頂点で，図示する際はひし形で表す．図 1 (a) のプログラムには if 文があるため，図 1 (b) の IVDFG では条件頂点が 1 つ生成される．

辺にはデータ辺と制御辺の 2 種類がある．データ辺はデータの流れを表し，図示する際は実線で表す．図 1 (a) のプログラムの 2 行目では，変数 result に変数 y を代入しているため，図 1 (b) のように変数頂点 “y” から演算頂点 “=” へ，“=” から変数頂点 “result” へデータ辺を接続する．また，演算を介する代入文の場合は，演算対象の変数から演算子へ，演算子から結果の値を格納する変数へ辺を接続する．したがって，図 1 (a) のプログラムの 3 行目では，変数頂点 “x” と “y” から演算頂点 “>” へ，“>” から条件頂点 “if” へデータ辺を接続する．複数の演算を介する場合は，演算子の優先順位に従って頂点と演算子をデータ辺で接続していく．

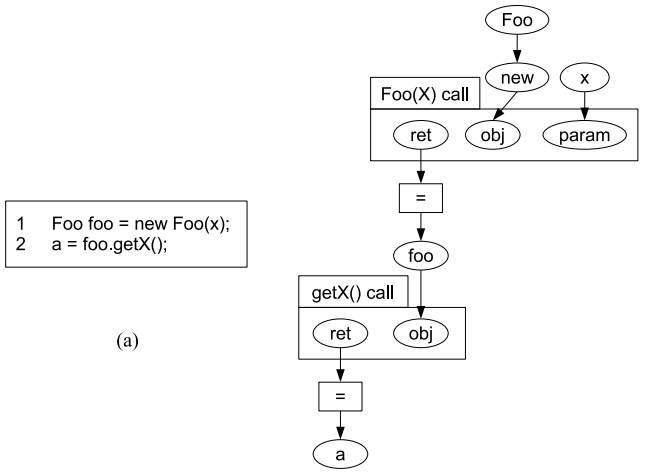

図 2 メソッド・コンストラクタ呼び出しの IVDFG の例，
(a) メソッド getX() とコンストラクタ Foo(X) の呼び出し (b) (a) の IVDFG
Fig. 2 Example of IVDFG of method and constructor call,
(a) method getX() and constructor Foo(X) call (b) IVDFG of (a)

一方，制御辺は制御の流れを表し，図示する際は点線で表す．制御文はそれぞれブロックを持つが，ブロックの中にある演算が実行されるかどうかは制御文の条件式の評価結果に左右される．このような場合，制御文の条件頂点からブロックの中にある演算に対して制御辺を接続する．図 1 (a) のプログラムでは，3 行目の if 文は 4 行目のブロックの中に代入演算子を持つため，図 1 (b) のように，条件頂点 “if” から演算頂点 “=” へ制御辺を接続する．

3.2 メソッド・コンストラクタ呼び出しの IVDFG

メソッド・コンストラクタの呼び出しは，呼び出し式の 1 回の出現ごとにオブジェクト頂点 “obj”，引数頂点 “param”，戻り値頂点 “ret” を生成する．オブジェクト頂点 “obj” は，メソッド呼び出しではインスタンス変数を表し，コンストラクタ呼び出しでは生成されるクラスを表す．static メソッドの呼び出しでは “obj” は生成されない．また，引数頂点 “param” はメソッド・コンストラクタ呼び出しの実引数を表し，引数の数だけ生成される．戻り値頂点 “ret” はメソッド・コンストラクタ呼び出しの戻り値を表す．戻り値が void の

Vol.2010-SE-167 No.29
2010/3/19

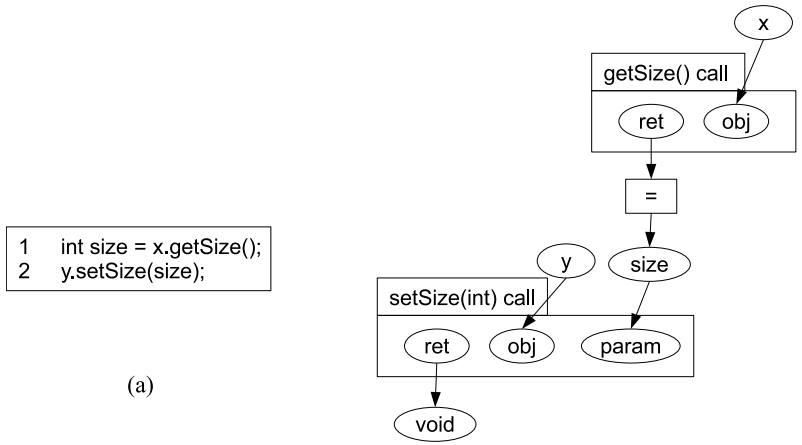

図 3 メソッド接続関係の例,
(a) メソッド getSize() と setSize(int) の接続，(b) (a) の IVDFG
Fig. 3 Example of method association,
(a) association between method getSize() and setSize(int), (b) IVDFG of (a)

メソッドでは，“ret” は “void” という仮想的な変数頂点を生成してこれに接続する．“obj”，“param”，“ret” はいずれも変数頂点の一種として扱う．**図 2** にメソッド・コンストラクタ呼び出しの IVDFG の例を示す．図 2 (a) のプログラムでは，1 行目で引数 x を与えてコンストラクタ Foo を呼び出しているため，図 2 (b) のように “obj” に “Foo” と “new” の順に，“param” に変数頂点 “x” からデータ辺を接続する．そして，コンストラクタ呼び出しによってインスタンス変数 foo を生成しているため，“ret” から変数頂点 “foo” にデータ辺を接続する．また，図 2 (a) の 2 行目では，1 行目で宣言した foo によってメソッド getX() を呼び出しているため，変数頂点 “foo” を “obj” に接続する．引数はないので，“param” は生成されない．最後に，getX() 戻り値が変数 a に格納されているため，“ret” から “a” にデータ辺を接続する．

3.3 メソッド接続関係

IVDFG を走査することで，プログラムのメソッド接続関係を解析することができる．メソッド接続関係は，データ辺によるものと制御辺によるものの 2 種類に分けられる．

IVDFG 上で，メソッド・コンストラクタ呼び出し X の戻り値頂点 “ret” が，メソッド・コンストラクタ Y のオブジェクト頂点 “obj” あるいは引数頂点 “param” にデータ辺をたどって到達するとき，X と Y は接続されていると表現し，それぞれ

$$ret\ \mathrm{X} \rightarrow obj\ \mathrm{Y},\ ret\ \mathrm{X} \rightarrow param\ \mathrm{Y}$$

と表す．前者は X の戻り値が Y のインスタンス変数になっていることを意味し，後者は X の戻り値が Y の引数になっていることを意味する．接続されているメソッドの例を **図 3** に示す．図 3 (a) の 1 行目で呼び出されているメソッド getSize() の戻り値は，変数 size に格納され，最終的に 2 行目で呼び出されているメソッド setSize(int) の引数となっている．このプログラムを IVDFG で表現したものが 図 3 (b) だが，メソッド getSize() の “ret” からデータ辺をたどることで，メソッド setSize(int) の “param” に到達する．したがって，

$$ret\ \text{int getSize()} \rightarrow param\ \text{void setSize(int)}$$

となる．このように，IVDFG を構築することにより，メソッド・コンストラクタの戻り値が変数に格納されたり，途中で何らかの演算が実行されていたとしても，戻り値が最終的にどのように使用されているかを知ることができる．また，データ辺を辿る経路上で 1 度でも条件頂点を経由した場合は，そのデータフローは，制御文を介した間接的な影響であることが分かる．

3.4 メソッド接続関係の列挙方法

メソッド接続関係を列挙するために，本研究では，ある 1 つのソフトウェアのソースコードから変数間データフローグラフを構築し，グラフ上でのデータフローの始点となっている各頂点から，終端となっている頂点への経路をすべて探索，列挙する方式を採用した．経路上で 1 度でも条件頂点を経由した経路については，間接的なデータフロー経路として区別している．

ライブラリなどソースコードを持たない範囲については IVDFG を構築せず，また，フィールドについての代入参照の関係もデータフローから除外しているので，ライブラリに対するメソッド呼び出しの戻り値や定数，フィールド参照がデータフローの始点となり，ライブラリに対するメソッド呼び出しの引数やフィールドへの値の代入がデータフローの終点となる．

4. 適 用 実 験

提案手法をツールとして実装し，適用実験を行った．

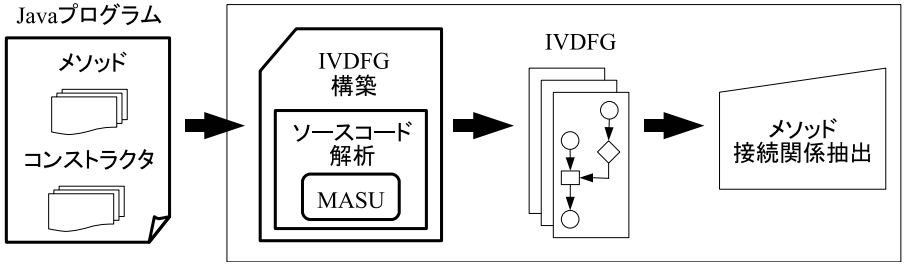

図 4 システムの構成
Fig. 4 Structure of the system

4.1 システムの構成

図 4 に本研究で作成したシステムの構成を示す．システムは，入力として Java プログラムのソースコードが与えられると，まずメソッド・コンストラクタごとに IVDFG を構築する．IVDFG の構築は，ソースコードの解析を行い，その解析結果から文単位で行う．ソースコード解析には，既存のメトリクス計測プラグインプラットフォーム MASU のソースコード解析モジュールを利用している．その後，構築した IVDFG からメソッドの接続関係を抽出する．

なお，システムの実行環境は以下のようになっている．

- OS:Microsoft Windows Vista Business Service Pack 2
- CPU:Intel(R) Core2 Duo CPU 1.80 GHz
- RAM:2.00GB

4.2 実 験 内 容

39 種類のソフトウェアに対してメソッド接続関係を抽出し，集計を行った．実験に使用したソフトウェア名とバージョン，総コード行数 (LOC) の一覧を **表 1** に示す．また，抽出・集計した接続関係を分析し，メソッドの適切な利用例となる情報が得られているかどうかを調査した．接続関係の分析に当たって，以下の視点から調査を行った．

- あるコンストラクタの戻り値がどのようなメソッドのオブジェクトとして使用されているか
- どのメソッドの戻り値がどのメソッドの引数となっているか

4.3 実 験 結 果

39 種類のソフトウェアから，総数 219,889 個のメソッド接続関係が抽出された．システム

表 1 実験に使用した 39 種類のソフトウェア
Table 1 39 softwares used in the experiment

ソフトウェア名	バージョン	規模 [LOC]	実行時間 [sec]
ANTLR	3.0.1	70,845	50(39+11)
Apache Ant	1.7.0	198,394	84(65+19)
Apache Batik	1.6	297,320	188(155+33)
Apache Cocoon	2.1.11	505,715	561(490+71)
Apache Commons Collections	3.2.1	59,490	81(71+10)
Apache Commons Logging	1.1.1	13,335	7(7+0)
Apache Log4J	1.2.15	52,765	36(34+2)
Apache Lucene	2.3.2	161,680	130(105+25)
Apache MyFaces	1.2.4	101,072	69(62+7)
Apache POI	3.1	297,466	255(199+56)
Apache Struts	1.2.7	157,990	101(87+14)
Apache Tomcat	6.0.14	322,971	231(181+50)
ArgoUML	0.24	294,466	199(165+34)
ASM	3.1	56,822	37(31+6)
Azureus	3.0.3.4	552,295	468(353+115)
BerkeleyDB Java Edition	3.2.76	223,921	173(137+36)
BioJava	1.5-beta2	300,578	239(202+37)
Cabos	0.8.1	294,600	413(327+86)
FreeMind	0.8.1	102,414	79(64+15)
GanttProject	2.0.7	69,469	69(62+7)
H2 Database	2008-08-28	150,352	145(121+24)
HSQLDB	1.8.0.10	157,388	95(83+12)
ImageJ	1.43n	86,566	121(101+20)
iText	2.1.3	170,542	145(116+29)
JBoss	4.2.3 GA	696,761	1,051(703+348)
JDK	5.0	885,887	2,055(1,054+1,001)
JEdit	4.3pre11	168,872	125(108+17)
JFreeChart	1.0.9	282,363	181(151+30)
JGraph	5.12.1.1	36,509	26(23+3)
JHotDraw	7.0.9	92,564	76(68+8)
Maven	2.0.9	60,416	44(38+6)
OpenCms	7.5	410,374	426(313+113)
PDFRenderer	2008-09-01	27,896	19(18+1)
SableCC	3.2	35,545	25(22+3)
Soot	2.2.4	381,357	306(235+71)
Spring Framework	2.5.5	487,177	478(358+120)
SVNKit	1.1.8	112,202	118(98+20)
Trove	2.0.4	9,237	8(8+0)
XMind	3.1.1	165,421	217(180+37)

Vol.2010-SE-167 No.29
2010/3/19

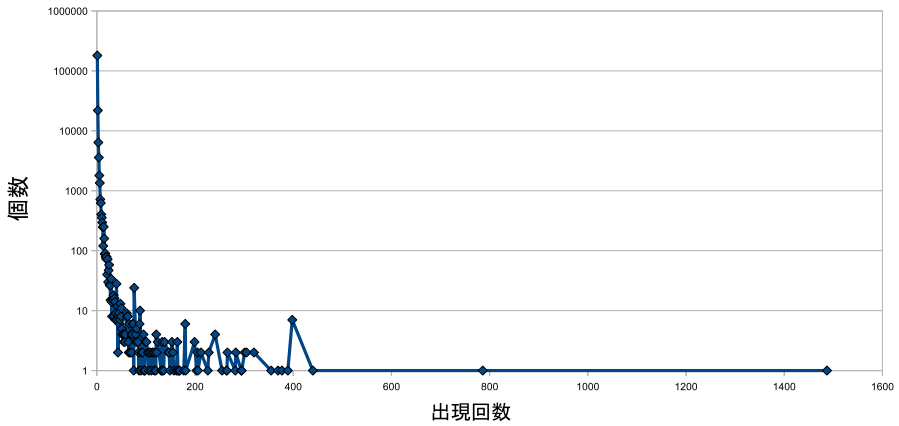

図 5 抽出された接続関係の分布
Fig. 5 Distribution of extracted associations

の実行時間を 表 1 に示す．実行時間は，合計時間（ソースコード解析の所要時間 + IVDFG 構築からメソッド接続関係抽出までの所要時間）となっている．抽出されたメソッド接続関係の分布を **図 5** に示す．図 5 は片対数グラフとなっており，横軸が抽出されたメソッド接続関係の出現回数，縦軸がその出現回数のメソッド接続関係がいくつ抽出されたかを表している．たとえば，1,487 回出現した接続関係は 1 個見つかったことになる．図 5 から明らかなように，メソッド接続関係の分布はごく少数の接続関係に集中し，特に 1 回のみ出現した接続関係は 180,426 個と全体の約 82.05% を占めた．

本システムの出力のうち，接続関係の出現回数により降順でソートした結果の上位 10 個を **図 6** に示す．図 6 の右側の数字はメソッド接続関係の出現回数を表しており，たとえばコンストラクタ RtfCtrlWordHandler(RtfParser, String, int, boolean, int, String, String, UNKNOWN) の戻り値がメソッド put(String, RtfCtrlWordHandler) の第二引数となっている接続関係が 1,487 個見つかったことが分かる．ここで，4 位から 9 位までの接続関係に着目すると，いずれもコンストラクタの戻り値頂点 “ret” とメソッドのオブジェクト頂点 “obj” との間の接続関係だが，戻り値頂点となっているコンストラクタおよび接続関係の出現回数が一致している．これらの接続関係から，コンストラクタ CmsListColumnDefinition(String) の戻り値が各メソッドのオブジェクトとして利用されており，その回数が同じであることからこれらのメソッドは 1 つのまとまった単位で利用する可能性が高いことが分かる．そこ

```
ret_<RtfCtrlWordHandler: RtfCtrlWordHandler(RtfParser, String, int, boolean, int, String, String, UNKNOWN)>  ->
    param_2_<HashMap: UNKNOWN put(String, RtfCtrlWordHandler)>                                                  1487

ret_<PropertyDescriptor: PropertyDescriptor(String, Class, UNKNOWN, String)>  ->
    param_1_<ArrayList: UNKNOWN add(PropertyDescriptor)>                                                        786

ret_<LLkParser: UNKNOWN getFilename()>  ->
    param_2_<NoViableAltException: NoViableAltException(UNKNOWN, UNKNOWN)>                                      440

ret_<CmsListColumnDefinition: CmsListColumnDefinition(String)>  ->
    obj_<CmsListColumnDefinition: boolean isPrintable()>                                                        398

ret_<CmsListColumnDefinition: CmsListColumnDefinition(String)>  ->
    obj_<CmsListColumnDefinition: boolean isSorteable()>                                                        398

ret_<CmsListColumnDefinition: CmsListColumnDefinition(String)>  ->
    obj_<CmsListColumnDefinition: String getId()>                                                               398

ret_<CmsListColumnDefinition: CmsListColumnDefinition(String)>  ->
    obj_<CmsListColumnDefinition: CmsMessageContainer getName()>                                                398

ret_<CmsListColumnDefinition: CmsListColumnDefinition(String)>  ->
    obj_<CmsListColumnDefinition: void setListId(String)>                                                       398

ret_<CmsListColumnDefinition: CmsListColumnDefinition(String)>  ->
    obj_<CmsListColumnDefinition: void setPrintable(boolean)>                                                   398

ret_<CmsListColumnDefinition: CmsListColumnDefinition(String)>  ->
    param_2_<Object: UNKNOWN addIdentifiableObject(String, CmsListColumnDefinition, int)>                       398
```

図 6 抽出された接続関係のうち出現回数の上位 10 個
Fig. 6 Top 10 frequently-appeared associations

で，抽出された接続関係のうち，出現回数が多いものほど信頼性が高いと考え，出現回数の上位 2,000 個を分析した結果，24 組のメソッド利用例の情報を読み取ることができた．

4.4 メソッド接続関係から読み取った利用例

メソッドの接続関係から読み取ることができたメソッドの利用例をいくつか紹介する．図 7 は，BioJava から抽出されたメソッド接続関係とソースコードの一部である．システムによって抽出されたメソッド接続関係は図 7 (a) に示すとおりである．省略して表記しているが，いずれの接続先のメソッド呼び出しのオブジェクト頂点にも同じコンストラクタ呼び出し戻り値頂点が接続されている．抽出されたメソッド接続関係を見ると，コンストラクタ QName(NamespaceConfigurationIF, String) から生成されたオブジェクトによってメソッド getLocalName()，getQName()，getURI() が呼び出されていることが分かる．実際にソースコードを調査すると，20 個のメソッドでこれらのメソッドが 1 つのまとまった単位で利用されていることが判明した．図 7 (b) と図 7 (c) にそのソースコードの例を示す．ソースコードは部分的に省略しており，以降も必要に応じて省略する．

さらに，本手法では利用順序・条件がそれぞれ異なるが 1 つのまとまりで利用されている

```
ret_<QName: QName(NamespaceConfigurationIF, String)> ->
    obj_<QName: String getLocalName()>          138
    obj_<QName: String getQName()>              138
    obj_<QName: String getURI()>                138
```

(a)

```
protected void endElement(QName poQName)
    throws SAXException {

    oHandler.endElement(poQName.getURI(),
            poQName.getLocalName(),
            poQName.getQName());
    }
```

(b)

```
public void endSubHit()
    {
        ...
        attributes.addAttribute(qName.getURI(),
                                qName.getLocalName(),
                                qName.getQName(),
                                "CDATA",
                                (String) hitProperties.get("subject_sq_len"));
        ...
    }
```

(c)

図 7 BioJava から抽出したメソッドの利用例
Fig. 7 Method usage extracted from BioJava

ようなメソッドの組も抽出できた．図 8 は，Apache Ant から抽出されたメソッド接続関係とソースコードの一部である．図 8 (a) のシステムの出力結果から，一連のメソッドはソースコード上で 1 つのまとまった単位で利用されている可能性が高いと判断できる．実際にソースコードを調査した結果，図 8 (b) のメソッド verifyBorlandJarV4(File) ではメソッド setClassname(String)，if 文のブロックでメソッド setClasspath(Path)，setFork(boolean) の順に呼び出されていることが分かる．一方，図 8 (c) のメソッド addGenICGenerated-Files(File, Hashtable) では，メソッド setFork(boolean)，setClasspath(Path)，if 文の else ブロックでメソッド setClassname(String) という順で呼び出されている．このように，これらの 3 つのメソッドは，実際に 14 個のメソッドで利用順序・条件を問わず 1 つのまとまりで利用されていた．

5. 考　　察

実験結果により，コンストラクタの戻り値がどのメソッドのオブジェクトとなっているかという接続関係から，1 つのまとまった単位で利用することが多いメソッドの組を抽出できることが分かった．また，従来のパターンマイニング手法では抽出の対象外となっている，複数のメソッドに渡って利用されているメソッドの組や，順序・条件がそれぞれ異なるよう

```
ret_<Java: Java(Task)> ->
    obj_<Java: void setClassname(String)>    14
    obj_<Java: void setClasspath(Path)>      14
    obj_<Java: void setFork(boolean)>        14
```

(a)

```
private void verifyBorlandJarV4(File sourceJar) {
    ...
    javaTask = new Java(getTask());
    ...
    javaTask.setClassname(VERIFY);
    ...
    if (classpath != null)  {
        javaTask.setClasspath(classpath);
        javaTask.setFork(true);
    }
    ...
}
```

(b)

```
private void addGenICGeneratedFiles(
    File genericJarFile, Hashtable ejbFiles) {
    ...
    genicTask = new Java(getTask());
    ...
    genicTask.setFork(true);
    ...
    genicTask.setClasspath(classpath);
    ...
    if (genicClass == null) {
        ...
    } else {
        ...
        genicTask.setClassname(genicClass);
    }
}
```

(c)

図 8 Apache Ant から抽出した利用順序・条件がそれぞれ異なるメソッドの利用例
Fig. 8 Method usage whose protocols and conditions differ extracted from Apache Ant

なメソッドの組も抽出できた．今回の調査では，39 種類のソフトウェアから抽出された接続関係のうち，出現回数の上位 2,000 個を分析対象としたが，実験結果では 10 数回程度しか出現しない接続関係からも有用な利用例が抽出できたことから，接続関係の出現回数が多いほど信頼できるとは断言できないことも分かった．

しかし，今回分析した 2,000 個の接続関係からは，メソッドの戻り値と引数の接続関係から，どのメソッドの戻り値をどのメソッドの引数として与えればよいかといった利用例に関する情報は抽出できなかった．これは，IVDFG 上でメソッドの戻り値として何が返されているかを始端までたどると，ただのリテラルが返されていてメソッドの接続関係が成立しないことが多かったためと思われる．あるいは，メソッドの引数がどのように使用されるかを終端までたどると，コレクションやマップに挿入されていたため有用な接続関係を得られなかったとも考えられる．これは，コレクションやマップへの挿入もメソッドであり，また一度コレクションやマップの要素となった場合 IVDFG 上ではそれ以上データフローを追跡できなくなるため，メソッドの接続関係がそこで途切れてしまうためである．

Vol.2010-SE-167 No.29
2010/3/19

6. むすび

本研究では，多数のコンポーネントを利用する大規模なソフトウェアの開発において，開発者がコンポーネントの適切な利用例をプログラムから抽出することが困難であるということを問題として，Java プログラムのメソッド接続関係を自動的に抽出する手法を提案した．また，プログラムを解析して構築した変数間データフローグラフからメソッドの接続関係を抽出するシステムを実装し，39 種類のソフトウェアに対してメソッドの接続関係を分析した．その結果，利用例として 1 つのまとまった単位で利用するべきメソッドの組を抽出できることが分かった．また，出現回数が比較的少なかったメソッド接続関係からも有益な利用例を得られることが分かった．

今後の課題としては，今回調査した 39 種類のソフトウェアに対して，個別にさらなる詳細な調査を行うことが挙げられる．特に，出現回数が少ないメソッド接続関係まで詳細に分析する必要がある．しかし，すべての接続関係を手動で調査することは多大な時間コストを要するため，出現回数が同じメソッド・コンストラクタの組を自動で検出するような，接続関係の分析をサポートするためのシステムの作成が望まれる．また，1 つのまとまりで利用することが多いメソッドの組のうち，いずれか 1 つが単独で利用されていると本手法では同じ出現回数の接続関係として抽出できなくなるため，ある程度の出現回数の誤差も許容するかどうかは検討の余地がある．他に，現在の仕様では，メソッド接続関係を抽出する際に始端から終端までどのような経路，すなわちどのようなメソッド・コンストラクタを経由したか，あるいはいくつのメソッド・コンストラクタを経由したかといった情報は保持しなかった．これらの経路情報を考慮すると，抽出される情報が著しく増加すると考えられるが，これによってより詳細なメソッドの接続関係が得られる可能性もある．

謝辞　本研究は，文部科学省科学研究費補助金若手研究（B）（課題番号:21700030）の助成を得た．

参考文献

1) Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API Patterns as Partial Orders from Source Code: from Usage Scenarios to Specifications. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, pp. 25–34, 2007.
2) Uri Dekel and JamesD. Herbsleb. Improving API Documentation Usability with Knowledge Pushing. In *Proceedings of the 31st IEEE International Conference on Software Engineering*, pp. 320–330, 2009.
3) Martin Feilkas and Daniel Ratiu. Ensuring Well-Behaved Usage of APIs through Syntactic Constraints. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, pp. 248–253, 2008.
4) Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, Vol.12, No.1, pp. 26–60, 1990.
5) Benjamin Livshits and Thomas Zimmermann. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In *Proceedings of the 5th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, pp. 296–305, Sep 2005.
6) TungThanh Nguyen, HoanAnh Nguyen, NamH. Pham, JafarM. Al-Kofahi, and TienN. Nguyen. Graph-based Mining of Multiple Object Usage Patterns. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, pp. 383–392, 2009.
7) Christopher Scaffidi. Why are APIs Difficult to Learn and Use? *ACM Crossroads*, Vol.12, No.4, pp. 4–4, 2006.
8) Rok Stmiša, Peter Sewell, and Matthew Parkinson. The Java Module System: Core Design and Semantic Definition. In *Proceedings of the 22nd Annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 499–514, 2007.
9) Andrzej Wasylkowski and Andreas Zeller. Mining Temporal Specifications from Object Usage. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pp. 295–306, Nov 2009.
10) Mark Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pp. 439–449, 1981.
11) Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise Dynamic Slicing Algorithms. In *Proceedings of the 25th International Conference on Software Engineering*, pp. 319–329, 2003.