修士学位論文

題目

データフロー情報を用いたソフトウェア部品利用例の抽出

指導教員
井上 克郎 教授

報告者
柳 慶吾

平成 22 年 2 月 8 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 21 年度 修士学位論文

データフロー情報を用いたソフトウェア部品利用例の抽出

柳 慶吾

内容梗概

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．あるコンポーネントをどのコンポーネントと接続するかを理解したいとき，開発者は既存のプログラムからそのコンポーネントの実際の利用例を学習することが多い．しかし，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く，コンポーネントの利用環境や手順といった制限事項，同時に使用するべきコンポーネントといった情報を得ることは困難である．

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，プログラムのデータフローを表現する変数間データフローグラフを構築し，これを解析することでメソッドの接続関係を抽出する．提案手法を実装したシステムを，39 種類の様々な用途の小中規模なソフトウェアに対して適用し，抽出されたメソッド接続関係のうち出現回数の多い上位 2,000 個の接続関係を分析した結果，24 組のメソッドの利用例に関する情報を読み取ることができた．

主な用語

ソフトウェアコンポーネント
データフロー
コンポーネント利用例
静的解析

目 次

1 まえがき **3**

2 背景 **5**

2.1 コンポーネントの利用方法 5

2.2 プログラム依存グラフ 6

3 メソッド接続関係の抽出 **8**

3.1 変数間データフローグラフ 8

3.1.1 代入文の IVDFG 8

3.1.2 制御文の IVDFG 11

3.1.3 メソッド・コンストラクタ呼び出しの IVDFG 13

3.1.4 フィールドの使用 13

3.1.5 配列の使用 14

3.1.6 メソッド・コンストラクタの IVDFG 15

3.2 メソッド接続関係 16

3.2.1 データ辺によるメソッド接続関係 17

3.2.2 制御辺によるメソッド接続関係 18

3.2.3 メソッド接続関係の算出法 20

4 システムの実装 **23**

4.1 システムの構成 23

4.2 メソッド接続関係抽出アルゴリズム 23

5 適用実験 **25**

5.1 実験内容 25

5.2 実験結果 25

5.3 メソッド接続関係から読み取った利用例 28

6 考察 **32**

7 むすび **34**

謝辞 **35**

参考文献 **36**

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．この問題は，プロジェクトに新たに参入したばかりの開発者や，使ったことがないコンポーネントを初めて利用しようとしている開発者にとって特に深刻である．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 [7] することは難しく，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．これは，開発者が目的のコンポーネントあるいはその利用例となっているソースコード片を発見した時点で調査を終えるため，ドキュメントに記載された制限事項や，他の関係あるソースコード断片の情報を見落としやすいことが原因として考えられる．

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，このグラフを解析することでメソッドの接続関係を抽出する．プログラムのデータフローを解析することによって，メソッド・コンストラクタの戻り値がどのメソッドのインスタンスとなっているか，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が明らかになる．本手法では，プログラムから抽出し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がどのメソッドのインスタンスとなっているか，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小中規模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，いずれも一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．

以降，2 章ではコンポーネントの利用法と，本研究で提案した変数間データフローグラフの基となるプログラム依存グラフについて説明する．その後，3 章で変数間データフローグラフの定義とメソッド接続関係の抽出法について説明する．4 章ではシステムの実装とメソッド接続関係を抽出するアルゴリズムについて説明し，5 章では実装したシステムを適用して行った実験の内容と結果について説明し，メソッド接続関係から読み取れた利用例をいくつか紹介する．6 章で実験結果に対する考察を行い，最後に 7 章でまとめと今後の課題について説明する．

2 背景

2.1 コンポーネントの利用方法

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だが，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．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．

同時に実行すべき操作．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` 操作など，資源の確保と解放に関する操作はこれに該当する．

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，内部的に使用される前提のクラスであり，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．

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方法が利用可能である．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．

同時に実行すべき走査の抽出については，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，欠陥の検出としての比重が高い．本研究でのデータフロー抽出手法は，一つのオブジェクトに対して呼び出すメソッドの集合を取り出すことができるため，常に同時に使われていることを示すことはできるが,「同時に使わなくてはならない」という制約を表現することはない．利用制約の情報を明示的に提示する方法としては，メソッドのドキュメントに制約が記述されている場合，そのメソッドを使用しているソースコードに対して強調表示を行う手法 [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」という境界を開発者に設定させ，アクセスを制限する方法を用いることで対応している [8].

2.2 プログラム依存グラフ

プログラム依存グラフ (PDG, Program Dependence Graph) とは，プログラムのデータ依存関係 (Data Dependence) と制御依存関係 (Control Dependence) を表現する有向グラフである．PDG は，頂点としてソースプログラム p の各文に対応する頂点およびメソッドの

引数などを扱うための特殊頂点を持ち，それらの頂点間をデータの流れやメソッド呼び出し，パラメータ渡しなどを表す辺とデータ依存関係，制御依存関係を表す辺によって接続する．

プログラムのデータ依存関係とは，変数 v を定義している文 s_1 から v を参照している文 s_2 へ，v を再定義しない実行経路が PDG 上に少なくとも 1 つ存在するとき，文 s_1 から s_2 に対し変数 v に関して成り立つ関係である．また，制御依存関係とは，文 s_1 が制御文であり， s_1 の結果によって文 s_2 が実行されるかどうかが決定されるとき，文 s_1 から s_2 に関して成り立つ関係である．

図 1 に PDG の例を示す．図 1 (a) のプログラムを PDG で表現すると図 1 (b) のようになる．プログラムの文を表す各頂点は，データの流れを表す辺によって接続される．ここで，図 1 (a) のプログラムでは，1 行目の文で変数 a を定義，3 行目の文で a を参照しており，図 1 (b) の PDG において 1 行目の文から 3 行目の文へ a を再定義しない実行経路が存在するため，1 行目の文から 3 行目の文に対し変数 a に関してデータ依存関係が成立する．また，2 行目の文は制御文であり，2 行目の文の結果によって 3 行目の文が実行されるかどうかが決定するため，2 行目の文から 3 行目の文に関して制御依存関係が成立する．

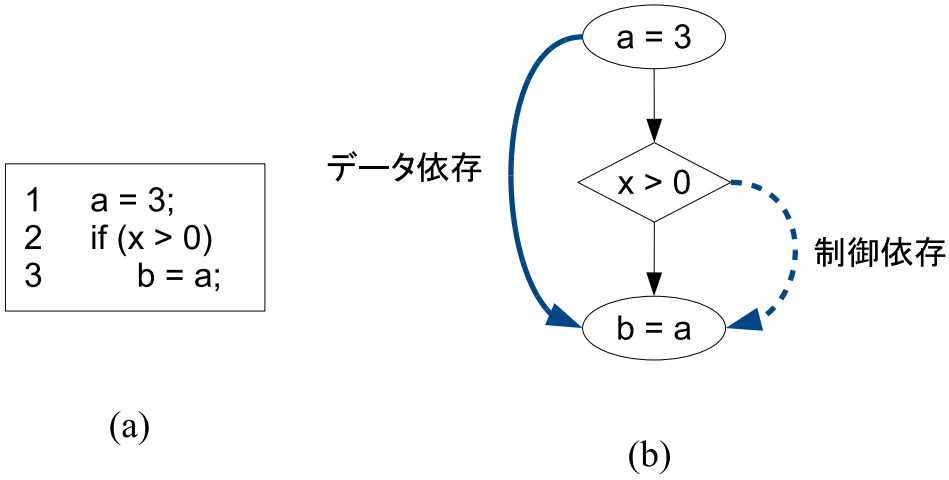

図 1: PDG の例，(a) Java プログラム，(b) (a) のプログラムの PDG

3　メソッド接続関係の抽出

本章では，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について説明する．まず，メソッドの接続関係を抽出するために必要な変数間データフローグラフの定義と，その構築ルールについても説明する．

3.1　変数間データフローグラフ

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 (IVDFG, Inter-Variable Data Flow Graph) を定義する．IVDFG は，変数間のデータの流れを表した有向グラフで，PDG を変数に着目して簡略化したものである．PDG との違いは，頂点が文単位ではなく変数・演算単位で，データフロー解析に特化したものとなっている．IVDFG は頂点と単方向辺から構成される．頂点には変数頂点，演算頂点，条件頂点の 3 種類がある．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・コンストラクタの仮引数に対してそれぞれ一つ生成される．その他に，変数頂点は後述するメソッド・コンストラクタに関連する要素においても汎用的に用いられる．変数頂点は，IVDFG を図示する際は楕円形の頂点で表す．演算頂点は，プログラム中の演算一つに対して一つ生成される頂点で，図示する際は長方形で表す．条件頂点は，if 文や for 文などの制御文に対して一つ生成される頂点で，図示する際はひし形で表す．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，制御辺は制御の流れを表す．グラフ上ではデータ辺は実線で，制御辺は点線で表わされる．以下では，単に辺を接続すると記述する場合はデータ辺を接続するものとする．

IVDFG は文単位で構築される．以降では，各文に対する IVDFG の構築ルールについて説明する．

3.1.1　代入文の IVDFG

代入文では，基本的に右辺に登場する要素から左辺に登場する要素へ辺を接続する．右辺に登場する要素としては，変数，メソッド・コンストラクタ呼び出しの戻り値，配列参照の値，フィールド参照の値，リテラルがある．以降，これらの要素をまとめて rhs と表現する．一方，左辺に登場する要素としては，変数，配列定義の値，フィールド定義の値がある．以降，これらの要素をまとめて lhs と表現する．

代入演算子 $asgOp$ ”＝”のみが登場する直接代入文

$$lhs \ = \ rhs$$

の場合は，rhs から $asgOp$ へ，$asgOp$ から lhs へ辺を接続する．したがって，接続ルールは

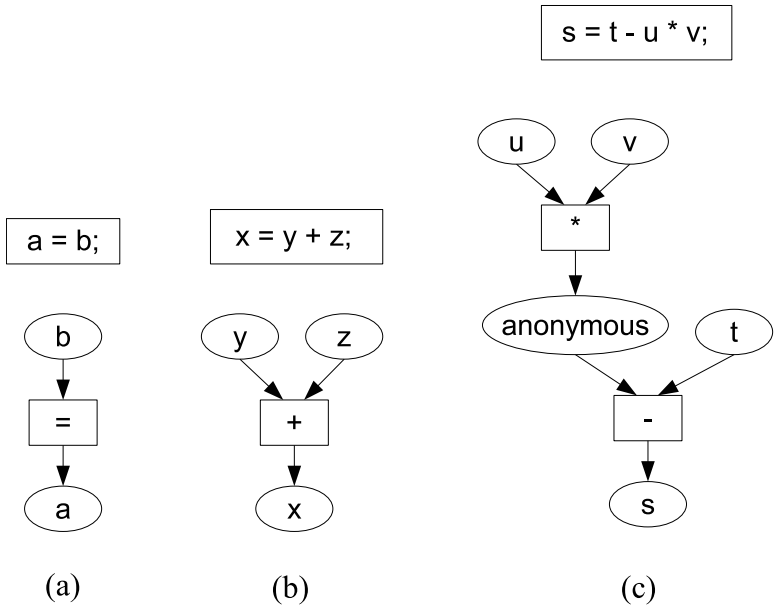

図 2: 代入文の IVDFG，(a) 直接代入文の IVDFG，(b) 二項演算を介する代入文の IVDFG，(c) 複数の二項演算を介する代入文の IVDFG

$$rhs \rightarrow asgOp \rightarrow lhs$$

となる．図 2 (a) に直接代入文の IVDFG の例を示す．図 2 (a) のプログラムでは，右辺の変数 b から演算子 ”= ”へ，”= ”から左辺の変数 a へ辺を接続する．なお，rhs に単項演算子 $monoOp$ が使用されている場合 ($monoOp\ rhs$，例：$++b$) は， rhs の値が単項演算子によって更新されると考え，以下のルールに従い rhs から $monoOp$ へ，$monoOp$ から $anonymous$ へ辺を接続しておく．ここで，$anonymous$ は演算の結果を一時的に保管するために生成される仮想的な変数頂点である．

$$rhs \rightarrow monoOp \rightarrow anonymous$$

ただし，単項演算子が後置されている場合 ($rhs\ monoOp$，例：$b++$) は，代入文が実行された後に rhs の値が更新されることになるため，代入文の結果に影響を及ぼさないものと考え，以下のルールに従い rhs から $monoOp$ へ，$monoOp$ から rhs へ辺を接続する．

$$rhs \rightarrow monoOp$$
$$monoOp \rightarrow rhs$$

一方，右辺に二項演算子 $biOp$ を介している代入文

$$lhs \ = \ rhs_1 \ \ biOp \ \ rhs_2$$

の場合は，まず演算の対象となる二つの要素 rhs_1，rhs_2 から $biOp$ へ，$biOp$ から演算結果を格納する lhs へ辺を接続する．このとき，$biOp$ から $asgOp$ ”= ”へ辺を接続すると，演算頂点から演算頂点へ辺を接続することになるため，演算を介する場合は $asgOp$ は生成されない．したがって，接続ルールは

$$rhs_1 \rightarrow biOp$$
$$rhs_2 \rightarrow biOp$$
$$biOp \rightarrow lhs$$

となる．図 2 (b) に二項演算を介する代入文の IVDFG の例を示す．図 2 (b) のプログラムでは，演算対象となる変数 y と z から演算子 ”+ ”へ辺を接続し，”+ ”から ”= ”ではなく結果の値を格納する変数 x へ辺を接続する．なお，複合代入演算子 $compAsgOp$ を使用した代入文

$$lhs \;\; compAsgOp \;\; rhs$$

の場合は，lhs と rhs との演算を行った結果を lhs に格納することになるため，直接代入文のルールではなく二項演算のルールを適用する．この場合， 複合代入演算子で使用されている二項演算子 $biOp$ を抽出し，lhs を rhs に変換した rhs' と rhs，$biOp$ の三つに二項演算のルールを適用する．したがって，接続ルールは

$$rhs \rightarrow biOp$$
$$rhs' \rightarrow biOp$$
$$biOp \rightarrow lhs$$

となる．

また，右辺に i 個の二項演算子 $biOp_i$ を介する代入文

$$lhs \;=\; rhs_1 \;\; biOp_1 \;\; rhs_2 \;\; ... \;\; biOp_i \;\; rhs_{i+1} \qquad (i \geqq 2)$$

の場合は，中置記法の演算式における計算の優先順位と Java プログラムにおける演算子の優先順位に従って二項演算のルールを適用していく．まず最も優先順位の高い二項演算子 $biOp_{first}$ を特定し，$biOp_{first}$ の演算対象となる rhs_{f1} と rhs_{f2} に対して二項演算の接続ルールを適用する．この演算結果は，仮想的な変数頂点 $anonymous_{i-1}$ を生成し，これに接続する．以降は，$biOp_i$ の優先順位に従って 同様に辺を接続していき，最後に評価された二項演算子 $biOp_{last}$ から lhs へ辺を接続する．したがって，複数の二項演算を介する代入文の接続ルールは

$$rhs_{f1} \rightarrow biOp_{first}$$

$$rhs_{f2} \rightarrow biOp_{first}$$
$$biOp_{first} \rightarrow anonymous_{i-1}$$
$$...$$
$$biOp_{last} \rightarrow lhs$$

となる．図 2 (c) にその例を示す．図 2 (c) では，演算子の優先順位に従ってまず変数 u と v から演算子 ”* ”へ辺を接続し，”* ”から $anonymous$ へ接続する．その後，$anonymous$ と変数 t から演算子 ”- ”へ辺を接続し，”- ”から変数 s へ辺を接続する．

3.1.2 制御文の IVDFG

if 文や for 文などの制御文があると，まず制御文に対して if，for などの条件頂点 $cond$ が一つ生成される．その後，制御文の条件式 $condExp$ から $cond$ へ辺を接続する．$condExp$ は，rhs のみで二項演算を介していなければそのまま $cond$ へ辺を接続し，”rhs_1 $biOp$ rhs_2” のように二項演算を介していれば二項演算のルールに従って辺を接続し，二項演算子 $biOp$ から $cond$ へ辺を接続する．以降,「式 exp の評価結果の値，すなわち式 exp に二項演算が含まれていなければ exp に対応する頂点から頂点 $node$ に直接辺を接続し，二項演算が含まれていれば二項演算のルールに従って辺を接続し，最後に評価される演算頂点 $biOp$ から $node$ へ辺を接続する」ことを

$$Result(exp) \rightarrow node$$

と表現する．さて，制御文はそれぞれ条件式の評価結果が真のとき，偽のときに実行されるブロックを個別に持つが，そのブロックの中に登場する演算の集合 $Operations$，具体的には演算子とメソッド・コンストラクタ呼び出し，配列・フィールド使用は，$condExp$ の評価結果によって実行されるかどうかが決定するため，$cond$ から $Operations$ に対して制御辺を接続する．したがって，制御文の接続ルールをまとめると

$$Result(condExp) \rightarrow cond \dashrightarrow Operations$$

となる．図 3 に制御文の IVDFG の例をいくつか示す．たとえば，図 3 (a) は if 文の IVDF である．図 3 (a) のプログラムでは，if 文の条件式 ”x $>$ y ”の評価結果から条件頂点 if へ接続されており，if からブロック内の演算子 ”$=$ ”に対して制御辺を接続する．なお，図 3 (b) のプログラムのように，制御文が入れ子構造になっている場合は，入れ子の内部の演算には制御辺を接続しない．したがって，図 3 (b) の IVDFG では，1 行目の if 文によって生成される条件頂点 if から，自身のブロックの中にある 2 行目の演算に対してのみ制御辺を接続する．

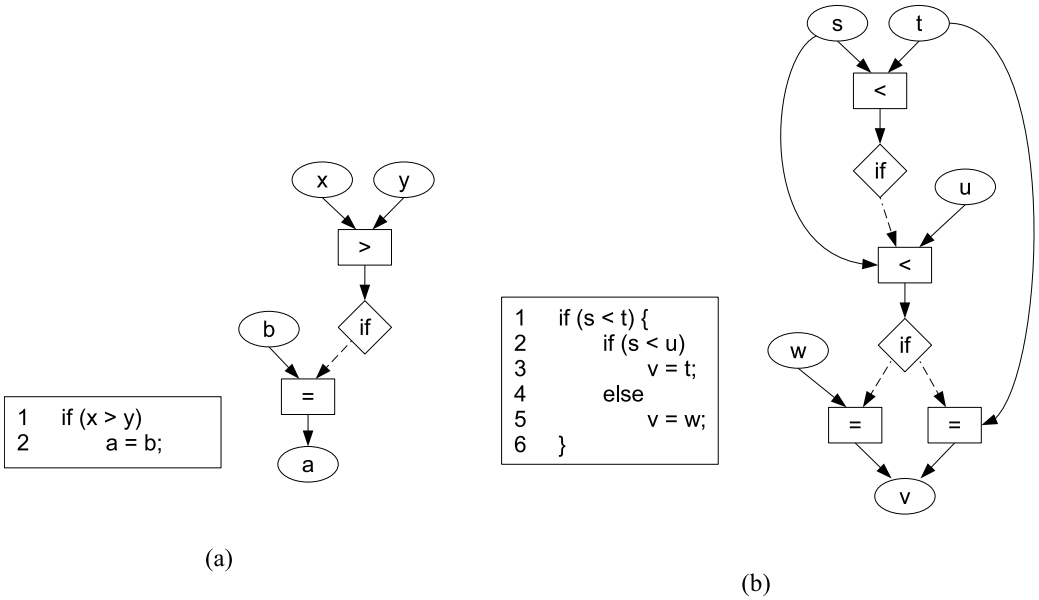

図 3: 制御文の IVDFG，(a) if 文の IVDFG，(b) 入れ子構造になっている if 文の IVDFG

なお，三項演算子 $triOp$ を使用する文

$$condExp \ ? \ trueExp \ : \ falseExp$$

にも同様の接続ルールが適用されるとする．ただし，演算子 ”? ”は条件頂点として扱い，演算子 ”: ”は演算頂点として扱う．すなわち，$triOp$ の条件式 $condExp$ から条件頂点 ”? ”へ辺を接続し，”? ”から条件式が真のときに評価される式 $trueExp$ に登場する演算 $trueOperations$ と偽のときに評価される式 $falseExp$ に登場する演算 $falseOperations$，演算頂点 ”: ”へそれぞれ制御辺を接続する．また，$trueExp$ と $falseExp$ の演算結果は，”: ”によって $anonymous$ へと接続されるが，三項演算子のみの文の場合この処理は省略される．したがって，三項演算子の基本的な接続ルールをまとめると

$$Result(condExp) \rightarrow \text{”?”}$$
$$\text{”?”} \dashrightarrow \text{” : ”}$$
$$\text{”?”} \dashrightarrow trueOperations$$
$$\text{”?”} \dashrightarrow falseOperations$$
$$Result(trueExp) \rightarrow \text{” : ”}$$
$$Result(falseExp) \rightarrow \text{” : ”}$$
$$\text{” : ”} \rightarrow anonymous$$

となる．

3.1.3 メソッド・コンストラクタ呼び出しの IVDFG

メソッド・コンストラクタ呼び出しは，呼び出し 1 回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 *method call* ではインスタンス変数 *inst* を表す．そのため，static メソッドの場合は *obj* は生成されない．また，コンストラクタ呼び出し *ctor call* では生成されるクラス *class* を表す．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 *actParam* を表す．引数がないメソッド・コンストラクタ呼び出しの場合は *param* は生成されない．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．この *ret* が，代入文における *rhs* に登場するときの頂点となる．戻り値が void のメソッド *return void* の場合は，*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．同様に，コンストラクタ呼び出しにおける *class* と予約語 *new* も変数頂点の一種とする．したがって，メソッド・コンストラクタ呼び出しの基本的な接続ルールは

$$
\begin{aligned}
&inst \rightarrow obj && (method\ call)\\
&class \rightarrow new \rightarrow obj && (ctor\ call)\\
&Result(actParam) \rightarrow param &&\\
&ret \rightarrow void && (return\ void)
\end{aligned}
$$

となる．図 4 にメソッド・コンストラクタ呼び出しの例を示す．プログラムの 1 行目では，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，*obj* には *class* Foo と *new* の順に辺を接続する．そして，*ret* は直接代入文における *rhs* として扱うため，*asgOp* を介して *inst* foo に接続される．プログラムの 2 行目では，1 行目で宣言した foo をインスタンス変数としてメソッド getX() を呼び出しているため，*inst* foo を *obj* に接続する．引数は与えられていないため，*param* は生成されない．最後に，メソッド呼び出し getX() の *ret* を変数頂点 a に接続する．

3.1.4 フィールドの使用

フィールドの使用は，メソッド・コンストラクタ呼び出しと同様に考え，使用されるごとにオブジェクト頂点 *obj* と値頂点 *val* を生成する．*obj* はフィールドのインスタンス変数 *inst* を，*val* はフィールドの値をそれぞれ表す．メソッド呼び出しと同様に，static フィールドの場合は *obj* は生成されない．また，*val* はフィールド定義 $field\ definition$ の場合は

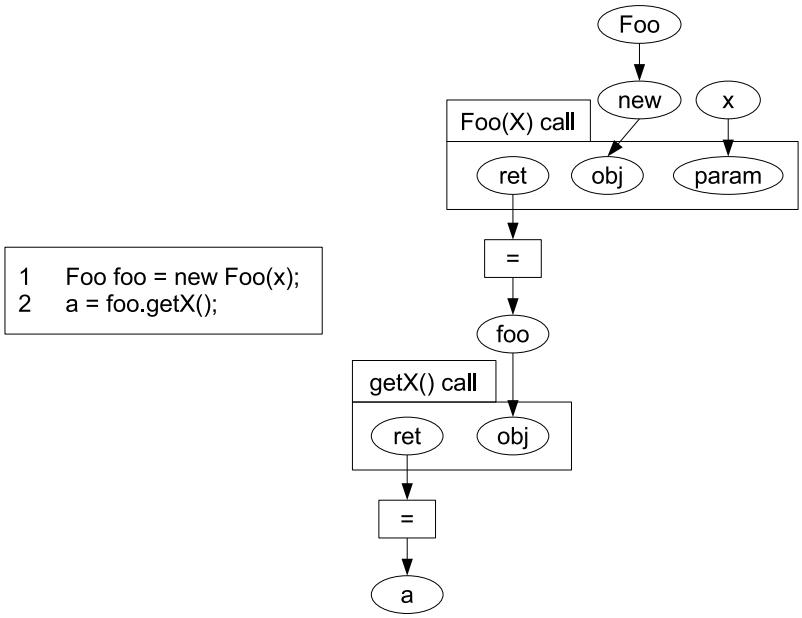

図 4: メソッド・コンストラクタ呼び出しの IVDFG

フィールドの値 $fieldValue$ から辺を接続し，フィールド参照の場合は代入文における rhs として振る舞う．obj と val は変数頂点の一種として扱う．まとめると，フィールド使用の基本的な接続ルールは

$$inst \rightarrow obj$$
$$Result(fieldValue) \rightarrow val \qquad (field\ definition)$$

となる．

3.1.5 配列の使用

配列操作も，メソッド呼び出しと同様に考え，使用されるごとにオブジェクト頂点 obj，引数頂点 ind，値頂点 val を生成する．obj は呼び出される配列 $array$ を，ind は引数 $arrayIndex$ を，val はその引数の要素の値 $arrayValue$ をそれぞれ表す．また，配列の操作には値の定義 $array\ definition$ と値の参照 $array\ reference$ の 2 種類があり，$array\ definition$ の場合は，戻り値頂点 ret も生成する． ret は戻り値が void のメソッド呼び出しと考えて $void$ に接続する．一方，$array\ reference$ の場合は，val が代入文における rhs として振る舞う．したがって，配列の使用に関する接続ルールは

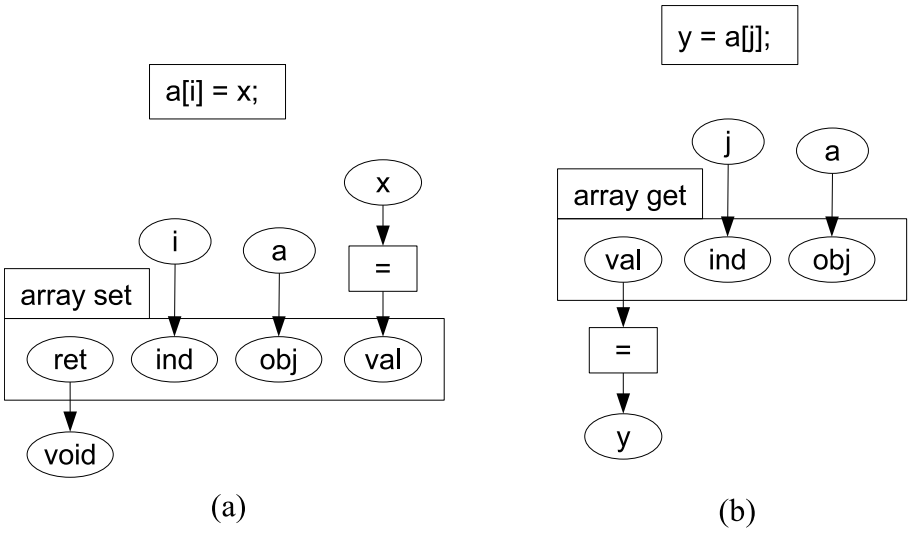

図 5: 配列操作の IVDFG，(a) 配列の値の定義，(b) 配列の値の参照

$$
\begin{aligned}
&array \rightarrow obj \\
&Result(arrayIndex) \rightarrow ind \\
&Result(arrayValue) \rightarrow val && (array\ definition) \\
&ret \rightarrow void && (array\ definition)
\end{aligned}
$$

となる．図 5 に配列使用の IVDFG の例を示す．図 5 (a) のプログラムでは，配列 a の i 番目の要素に変数 x を代入しているため，a を obj に，i を ind に，x を val に， ret を $void$ に接続する．また，図 5 (b) のプログラムでは，配列 a の j 番目の要素を参照して値を変数 y に代入しているため，a を obj に，j を ind に，val を y に接続する．

3.1.6 メソッド・コンストラクタの IVDFG

メソッド・コンストラクタでは，仮引数に対して変数頂点を生成する．仮引数がない場合は変数頂点は生成されない．メソッドの場合は，return 文がある場合，メソッド戻り頂点 $return$ を生成し，return 文によって返される式 $returnExp$ から辺を接続する．$return$ は変数頂点の一種として扱う．したがって，return 文の接続ルールは

$$Result(returnExp) \rightarrow return$$

となる．図 6 のメソッドでは，仮引数 x と y に対して変数頂点が生成され，これらの演算結果から，2 行目の return 文によって生成される $return$ へ辺を接続する．

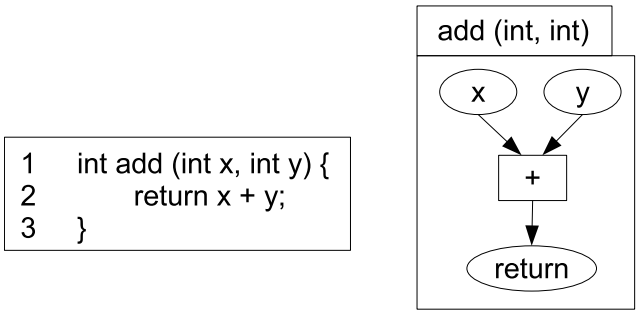

図 6: return 文があるメソッドの IVDFG

ここで，メソッド・コンストラクタ呼び出しとメソッド・コンストラクタを対応付けるための辺を接続する必要がある．まず，メソッド・コンストラクタ呼び出しに引数が i 個あれば，メソッド・コンストラクタ呼び出しの引数頂点 $param_i$ からメソッド・コンストラクタの仮引数 $formalParam_i$ へ辺を接続する．次に，メソッド・コンストラクタ呼び出しのオブジェクト頂点 obj から，メソッド・コンストラクタ内で this キーワードによって参照されている要素のオブジェクト頂点 $thisObj$ へ辺を接続する．最後に，return 文があるメソッドの場合，メソッド戻り頂点 $return$ からメソッド呼び出しの戻り値頂点 ret へ辺を接続する．まとめると，メソッド・コンストラクタ呼び出しとメソッド・コンストラクタの接続ルールは

$$param_i \rightarrow formalParam_i$$
$$obj \rightarrow thisObj$$
$$return \rightarrow ret$$

となる．図 7 にメソッド呼び出しとメソッドの接続例を示す．まず，図 7 (a) のプログラムでは，変数 b を引数として与えてクラス X のメソッド foo(int) を呼び出している．図 7 (b) の 5 行目を見ると，メソッドの仮引数は z であることが分かるため，$param$ から，メソッドの仮引数変数頂点 z へ辺を接続する．次に，(b) の 6 行目でクラス X のフィールド y が使用されている．6 行目の this キーワードによって参照されているのは，(a) の呼び出し時に使用された変数 x であるため，obj から，メソッド内で this キーワードによって参照されているフィールド y の $thisObj$ へ辺を接続する．最後に，(b) の 6 行目の return 文によってメソッドの演算結果を呼び出し側に返すために，$return$ から ret へ辺を接続する．

3.2 メソッド接続関係

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．

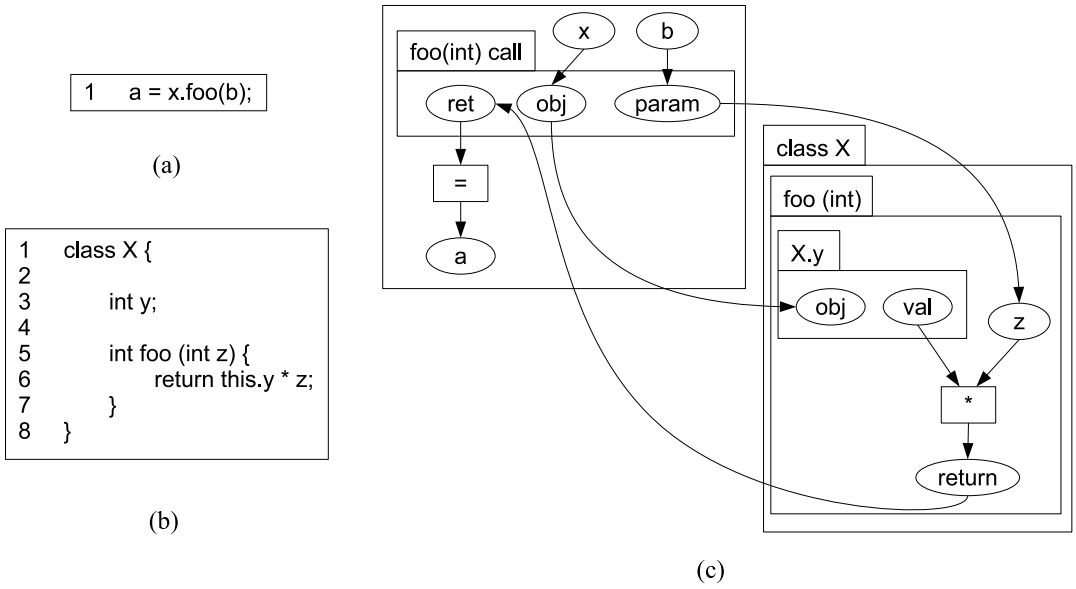

図 7: メソッド呼び出しとメソッドの IVDFG，(a) メソッド呼び出し，(b) 呼び出されているメソッド，(c) メソッド呼び出しとメソッドの接続

3.2.1 データ辺によるメソッド接続関係

抽出した IVDFG をたどることで，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．ここで，IVDFG 上において，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

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

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 8 に示す．図 8 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．この一連の操作を IVDFG で表現すると，メソッド getSize() の ret からグラフを下向きにたどることで，メソッド setSize(int) の $param$ に接続されていることが判明する．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ていたり，途中で何らかの演算を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容易に知ることができる．

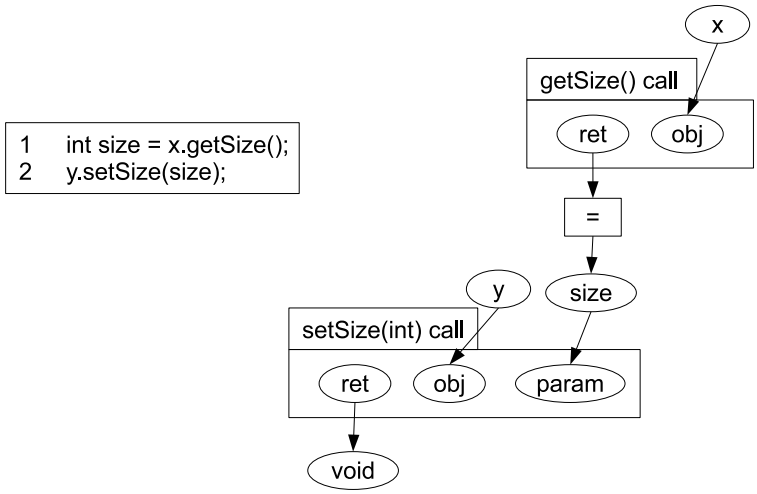

図 8: 接続されているメソッドの例

さらに，IVDFG を構築することで，図 9 のような複数のメソッド間に渡るメソッド接続関係も抽出できる．図 9 (a) のプログラムでは，1 行目でメソッド getName() の戻り値が，インスタンス変数 x で呼び出されているクラス X のメソッド setData(String) の引数として使用されている．その後，2 行目でインスタンス変数 x でクラス X のメソッド getData() を呼び出し，その戻り値をメソッド foo(String) の引数として使用している．クラス X のソースコードは図 9 (b) だが，ここで 9 行目に着目すると，8 行目のメソッド getData() によって返されるのは 3 行目で宣言されているクラス X のフィールド name であることが分かる．さらに，フィールド name は 6 行目の代入文で値をセットされているため，5 行目のメソッド setData(String) の引数の値が格納されていることが判明する．メソッド setData(String) の引数は，図 9 (a) の 1 行目によるとメソッド getName() の戻り値であるため，結果としてメソッド foo(String) の引数にはメソッド getName() の戻り値が使用されていることになる．この一連の操作を IVDFG で表現すると図 9 (c) のようになる．IVDFG 上で，メソッド getName() の *ret* からメソッド foo(String) の *param* へ辺が接続されているため，

$$ret \text{ String getName()} \rightarrow param \text{ void foo(String)}$$

となることが直ちに判明する．このように，メソッド接続関係の追跡は辺が接続されている限り実行され，経路でどのようなメソッドを経由したかという情報は保持されない．

3.2.2 制御辺によるメソッド接続関係

メソッド接続関係は，制御辺によっても定義される．制御的なメソッド接続関係の例を図 10 に示す．図 10 のプログラムにおいて，2 行目で呼び出されているメソッド execute() が実

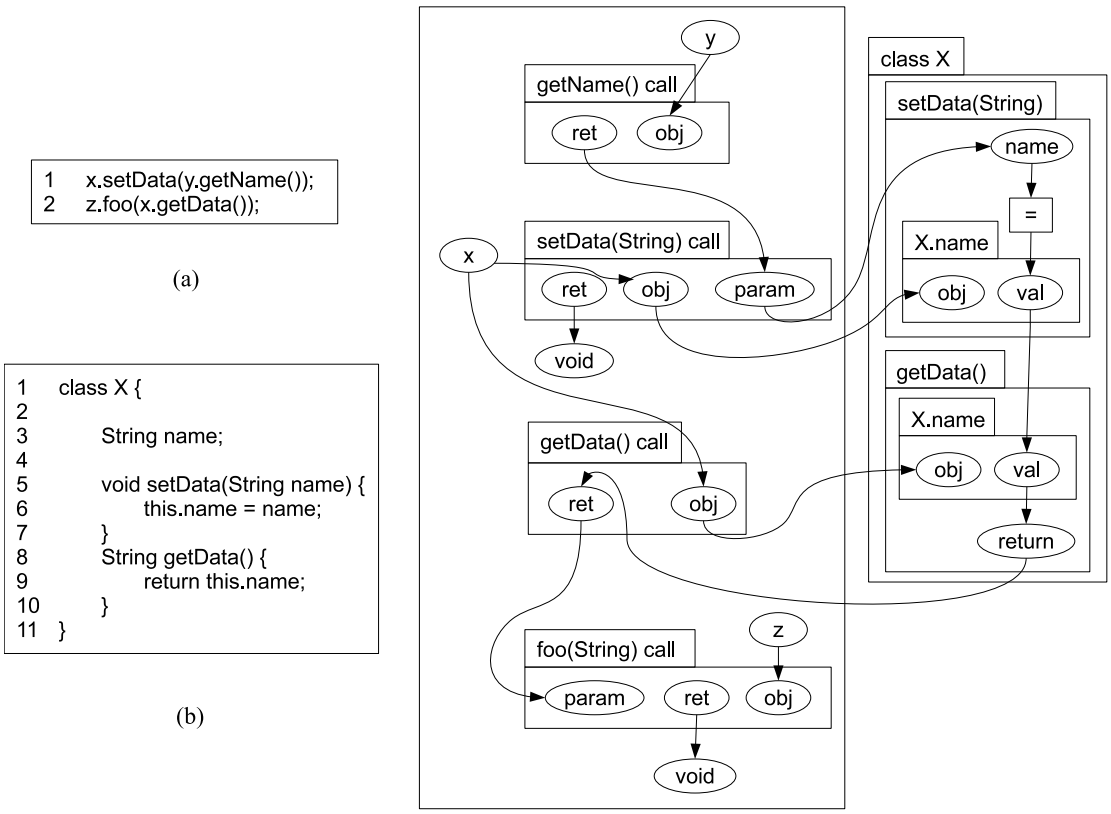

図 9: 複数のメソッド間におけるメソッド接続関係，(a) メソッドの呼び出し，(b) 呼び出されているメソッド，(c) IVDFG

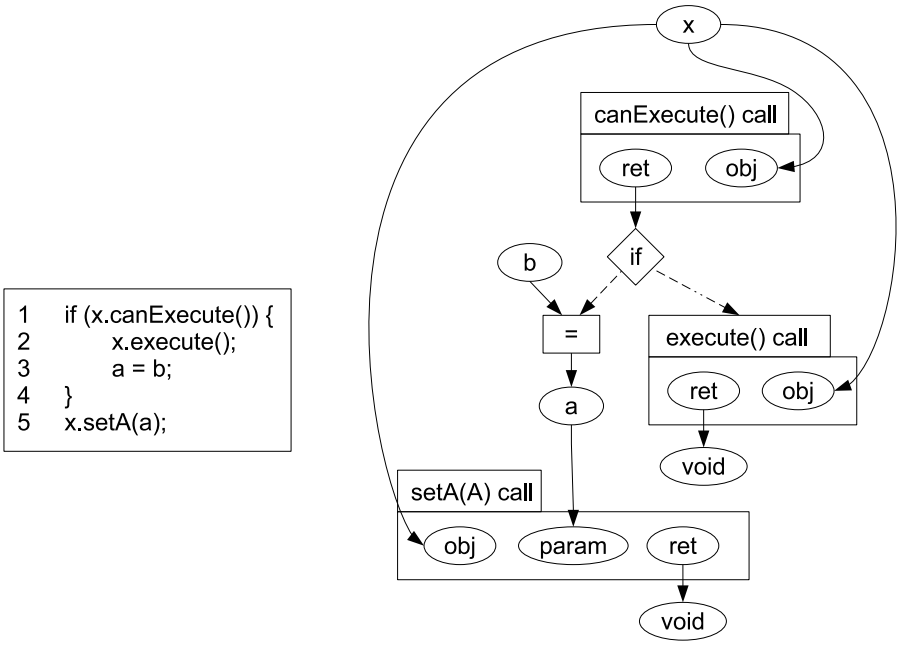

図 10: 制御的なメソッド接続関係

行されるかどうかは，1 行目で呼び出されているメソッド canExecute() の戻り値によって左右される．このとき，メソッド呼び出し canExecute() の戻り値はメソッド呼び出し execute() 自身と制御的な接続関係があると表現し，

$$ret \text{ boolean canExecute()} \xrightarrow{\text{if}} \text{void execute()}$$

のように表す．また，3 行目の代入文で値がセットされている変数 a は，5 行目で呼び出されているメソッド setA(A) の引数となっている．そのため，1 行目のメソッド呼び出し canExecute() の戻り値によって，5 行目で引数として使用されている変数 a は何らかの影響を受けることになる．したがって，メソッド呼び出し canExecute() の戻り値はメソッド呼び出し setA(A) の引数と制御的な接続関係があり，

$$ret \text{ boolean canExecute()} \xrightarrow{\text{if}} param \text{ void setA(A)}$$

となる．制御的な接続関係も，IVDFG 上で条件頂点からの制御辺をたどることによって容易に抽出できる．

3.2.3 メソッド接続関係の算出法

これまでの定義をまとめると，IVDFG 上でメソッド・コンストラクタの ret からメソッド・コンストラクタの obj あるいは $param$ に辺が接続されており，かつそれらが IVDFG

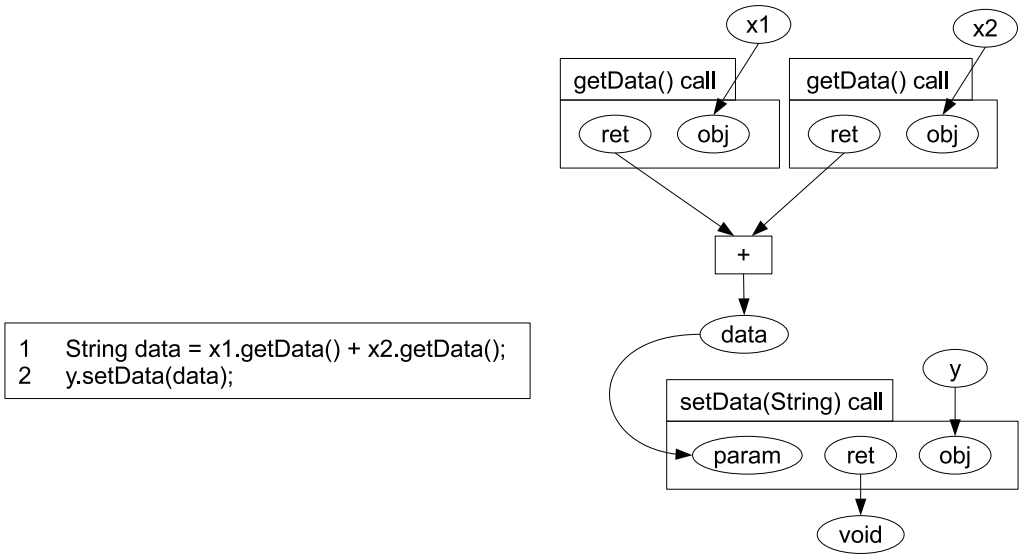

図 11: メソッド接続関係の算出

における始端・終端の要素であった場合，一つのメソッド接続関係として数えられる．しかし，このままではプログラムの構成によっては冗長なメソッド接続関係が抽出される可能性があるため，さらに定義を追加する．

これまでの定義から明らかなように，IVDFG 上では演算頂点によってのみデータフローの合流が起こる．そのため，この演算頂点に同じメソッド呼び出しが複数接続されていると，それらがすべてメソッド接続関係として抽出されてしまう．図 11 に例を示す．図 11 (a) のプログラムは，異なるインスタンス変数で同じメソッド getData() を 2 度呼び出し，その戻り値を連結してメソッド呼び出し setData(String) の引数として使用している．このプログラムの IVDFG は図 11 (b) のようになるが，これをそのまま解析すると

$$ret \text{ String getData()} \rightarrow param \text{ void setData(String)}$$

のメソッド接続関係が 2 回出現していると抽出される．しかし，このプログラムからは，メソッド呼び出し getData() の戻り値がメソッド呼び出し setData(String) の引数になっているという関係が分かればよいため，同じメソッド接続関係を 2 回抽出する必要はない．そのため，演算頂点によってデータフローの合流が起こると，それぞれのフローに含まれているメソッド呼び出しの戻り値を調べ，同じものがあれば先に出現したもののみを考慮して残りは無視する．同じメソッド・コンストラクタかどうかの判定は，パッケージ名，戻り値の型，引数の型の完全限定名を比較することで行う．

制御的なメソッド接続関係でも冗長な接続関係が抽出されることがある．図 12 にその例を示す．図 12 (a) のプログラムは，if 文の各ブロックでメソッド setData(Data) を呼び出

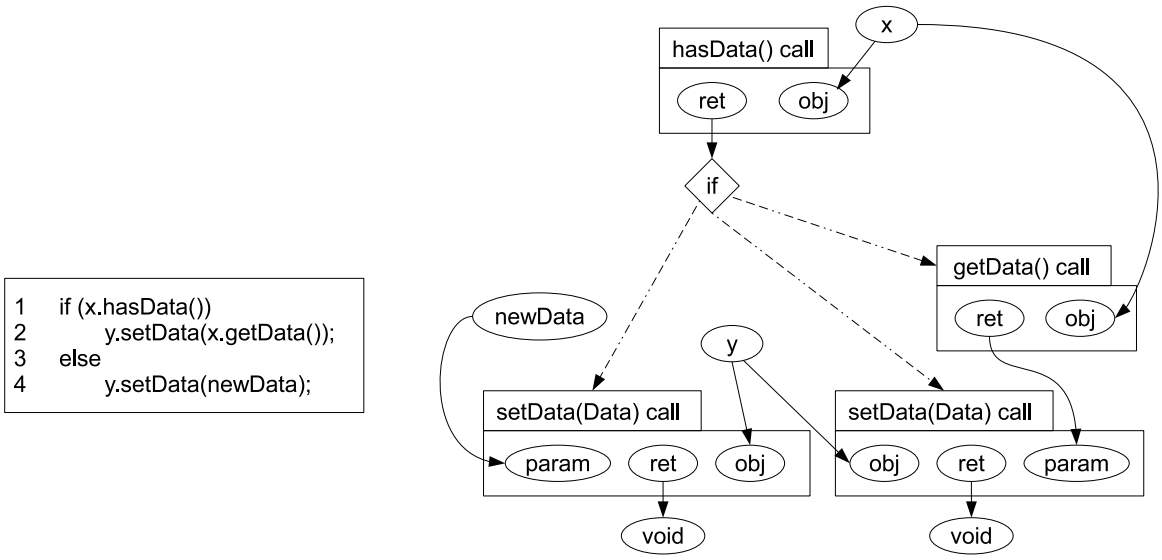

図 12: 制御的なメソッド接続関係の算出

している．このプログラムの IVDFG は図 12 (b) のようになるが，これをそのまま解析すると

$$ret \text{ boolean hasData}() \xrightarrow{\text{if}} \text{void setData(Data)}$$

の制御的な接続関係が 2 回出現したと抽出される．しかし，ここでもメソッド呼び出し hasData() の戻り値がメソッド呼び出し setData(Data) と制御的な接続関係であることが分かれば十分であるため，同じ接続関係を 2 回抽出する必要はない．したがって，制御的な接続関係を抽出する場合は，制御文のブロックに同じメソッド呼び出しが 2 個以上出現する場合，先に出現したもののみを考慮し残りは無視する．

4　システムの実装

本章では，本研究で作成したシステムについて説明する．また，IVDFG からメソッド接続関係を抽出するアルゴリズムに関する説明もここで行う．

4.1　システムの構成

図 13 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，その解析結果から構築ルールに従って文単位で行う．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．

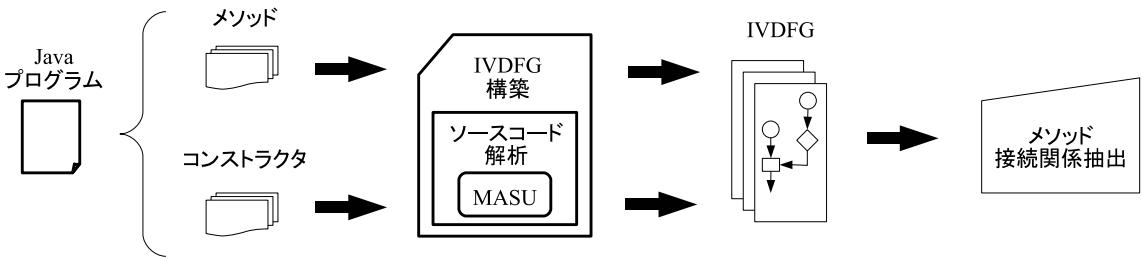

図 13: システムの構成

4.2　メソッド接続関係抽出アルゴリズム

IVDFG からメソッド接続関係を抽出する前に，まず IVDFG を効率よく走査できるように IVDFG の縮約を行う．たとえば，

$$ret\ \mathrm{getX}() \rightarrow \mathrm{x}$$
$$\mathrm{x} \rightarrow param\ \mathrm{setX}(\mathrm{X})$$

という IVDFG は，変数 x が他に登場しなければ

$$ret\ \mathrm{getX}() \rightarrow param\ \mathrm{setX}(\mathrm{X})$$

のように縮約することができる．

メソッド接続関係を抽出するためには，IVDFG 上で始端となる頂点と終端となる頂点を知る必要がある．始端と終端は，どちらか一方が判明すれば，もう一方は辺を接続されてい

る限りたどることによって知ることができるため，本アルゴリズムでは，実装が容易な始端の探索を行う．ある頂点が始端であるかどうかは，次のようにして判別できる．まず，その頂点を仮に始端であると見なし，その頂点を根として子を調査していき，子として登場した頂点はすべて始端ではないと見なす．すべての頂点を調べた後に始端であるとなっている頂点が始端である．もし，始端であると見なした頂点が実際には始端でなかったとしても，その頂点は始端ではないのでいずれ別の頂点を根とした場合の子として登場するため，誤って始端として検出されることはない．また，無駄な計算を省くために，根から子を走査する途中で始端ではないと見なされた頂点に到達すると，それ以降の子は調べない．

しかし，この方法では始端から始まる閉ループがあった場合，始端が始端と見なされないという問題がある．たとえば，次のような閉ループとなっている IVDFG を考える．各記号は頂点を表しており，各頂点はメソッド接続関係に関連する要素とする．

$$
\begin{aligned}
\bigcirc &\rightarrow \times \\
\times &\rightarrow \triangle \\
\triangle &\rightarrow \bigcirc
\end{aligned}
$$

○は実際には始端なのだが，○を根として子を走査すると，×，△と頂点をたどり，やがて○に到達し，○は子として出現したため始端ではないとみなされてしまう．ここで，この IVDFG がなぜ閉ループとなっているかを考えると，○を根として子に○が出現したためである．したがって，根と同じ頂点が子として現れた場合は，その頂点は始端ではないと見なす処理をスキップすることでこの問題を回避できる．

始端となる頂点が判明すると，その頂点がメソッド・コンストラクタの戻り値ならばその頂点を根として子をすべて訪問し，葉に到達すると，葉がメソッドのオブジェクト頂点あるいはメソッド・コンストラクタの引数ならば，根と葉のメソッド接続関係を記録する．また，根から子を走査する途中で条件頂点を訪問すると，その条件頂点の制御辺をすべて調べ，メソッド・コンストラクタ呼び出しあるいはオブジェクト頂点，引数頂点が含まれていれば，根とそれらの頂点との制御的なメソッド接続関係を記録する．このとき，無駄な計算を省くために，一度訪問した頂点には，その頂点を葉まで走査した場合に到達するすべての葉の情報を持たせる．

5　適用実験

本章では，提案手法の有効性を評価するために行った実験の内容とその結果について説明する．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

5.1　実験内容

39 種類の小中規模な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，以下の三つ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
- あるメソッドの戻り値を条件として，あるメソッドを実行するといった情報が得られるか

5.2　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．実行時間は，LOC 数万〜十数万程度のソフトウェアで数秒〜数十秒，十数万〜50 万程度のソフトウェアで数分〜十数分，50 万以上のソフトウェアで数十分〜150 分となった．抽出されたメソッド接続関係の分布を図 14 に示す．図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 14 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 15 に示す．図のように，実際の出力ではメソッド・コンストラクタの要素は

接頭語 _ < メソッド・コンストラクタ >

表 1: 実験に使用したソフトウェア

ソフトウェア名	バージョン	規模 (LOC)
ANTLR	3.0.1	70,845
Apache Ant	1.7.0	198,394
Apache Batik	1.6	297,320
Apache Cocoon	2.1.11	505,715
Apache Commons Collections	3.2.1	59,490
Apache Commons Logging	1.1.1	13,335
Apache Log4J	1.2.15	52,765
Apache Lucene	2.3.2	161,680
Apache MyFaces	1.2.4	101,072
Apache POI	3.1	297,466
Apache Struts	1.2.7	157,990
Apache Tomcat	6.0.14	322,971
ArgoUML	0.24	294,466
ASM	3.1	56,822
Azureus	3.0.3.4	552,295
BerkeleyDB Java Edition	3.2.76	223,921
BioJava	1.5-beta2	300,578
Cabos	0.8.1	294,600
FreeMind	0.8.1	102,414
GanttProject	2.0.7	69,469
H2 Database	2008-08-28	150,352
HSQLDB	1.8.0.10	157,388
ImageJ	1.43n	86,566
iText	2.1.3	170,542
JBoss	4.2.3 GA	696,761
JDK	5.0	885,887
JEdit	4.3pre11	168,872
JFreeChart	1.0.9	282,363
JGraph	5.12.1.1	36,509
JHotDraw	7.0.9	92,564
Maven	2.0.9	60,416
OpenCms	7.5	410,374
PDFRenderer	2008-09-01	27,896
SableCC	3.2	35,545
Soot	2.2.4	381,357
Spring Framework	2.5.5	487,177
SVNKit	1.1.8	112,202
Trove	2.0.4	9,237
XMind	3.1.1	165,421

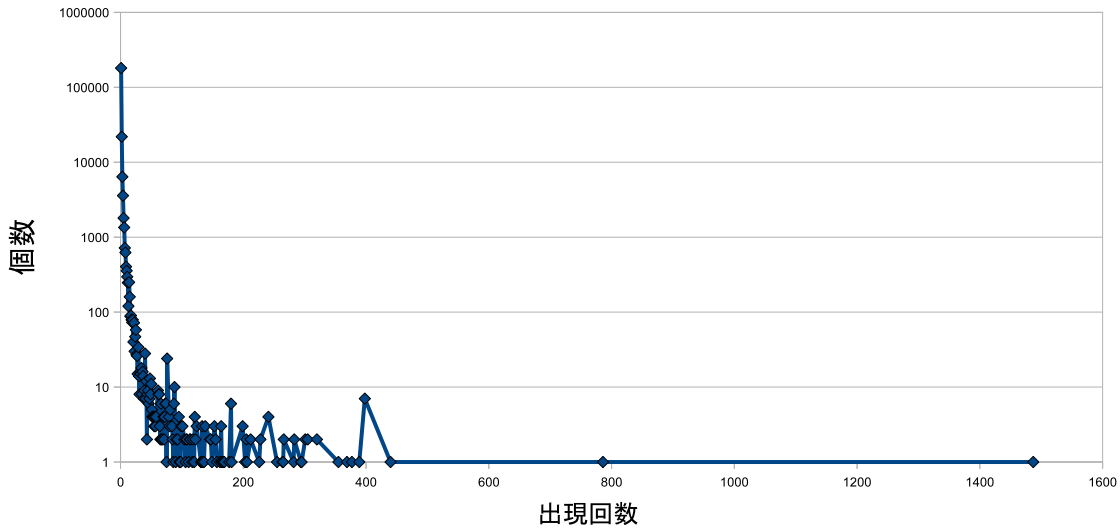

図 14: 抽出された接続関係の分布

というフォーマットで表現される．接頭語は，オブジェクト頂点なら obj，引数頂点なら param，戻り値頂点なら ret となる．param の後の数字は何番目の引数であるかを表している．また，メソッド名は

クラス名: 戻り値の型 メソッド名（引数 1 の型，引数 2 の型，...）

というフォーマットで表現される．紙面の都合上で省略しているが，実際にはクラス名，戻り値の型，引数の型は完全限定名で記述される．同様に，コンストラクタは

クラス名: コンストラクタ名（引数 1 の型，引数 2 の型，...）

となる．図 15 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一まとまりの単位で利用する可能性が高いことが分かる．そこで，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．

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 15: 抽出された接続関係の上位 10 個

5.3 メソッド接続関係から読み取った利用例

本節では，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 16 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 16 (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一まとまりで利用されていることが判明した．図 16 (b) と図 16 (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．

また，本手法によって，複数のメソッドに渡って一まとまりで利用されていたメソッドの組も抽出できた．図 17 は，Apache POI 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 17 (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一まとまり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，ま

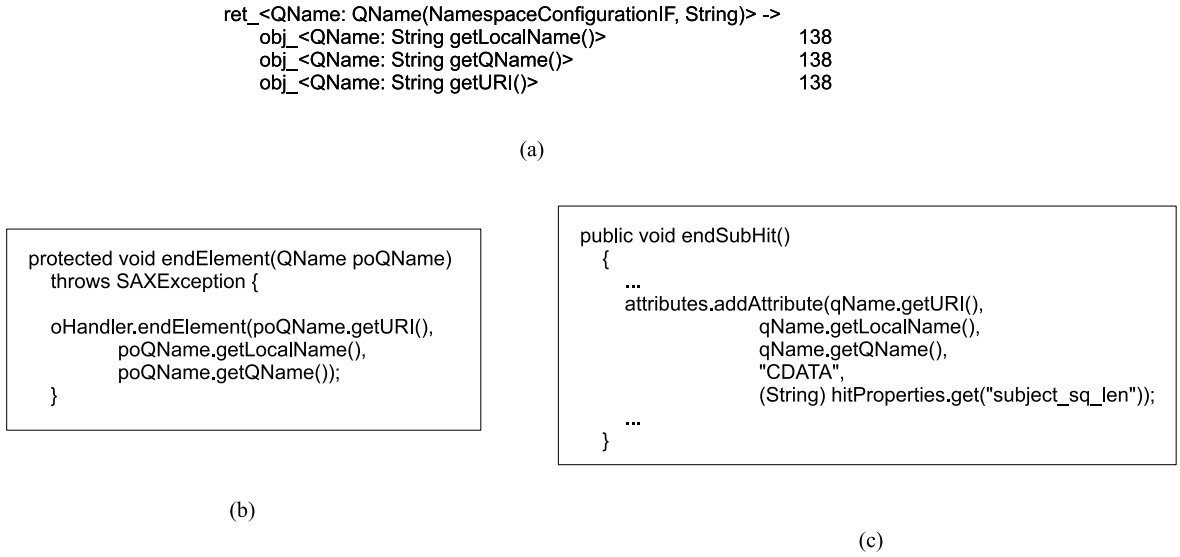

図 16: BioJava から抽出したメソッドの利用例

ず図 17 (b) のメソッド createSpContainer(HSSFSimpleShape, int) において，インスタンス変数 userAnchor によってメソッド isHorizontallyFlipped() と isVerticallyFlipped() が実行されている．その後，userAnchor は図 17 (c) のメソッド createAnchor(HSSFAnchor) に引数として渡され，メソッド createAnchor(HSSFAnchor) 内で getAnchorType()，getCol1() などのメソッドを呼び出している．このようにして，複数のメソッドに渡って計 18 個のメソッドが一まとまりで利用されていた．

さらに，本手法では利用順序・条件がそれぞれ異なるがセットで利用されているようなメソッドの組も抽出できた．図 18 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 18 (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一まとまり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 18 (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 18 (c) のメソッド addGenICGeneratedFiles(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，これらの三つのメソッドは，実際に 14 個のメソッドで利用順序・条件を問わず一まとまりで利用されていた．

```
ret_<HSSFClientAnchor: HSSFClientAnchor(int, int, int, int, short, int, short, int)>     ->
    obj_<HSSFAnchor: boolean isHorizontallyFlipped()>                                      19
    obj_<HSSFAnchor: boolean isVerticallyFlipped()>                                        19
    obj_<HSSFAnchor: int getDx1()>                                                         19
    obj_<HSSFAnchor: int getDx2()>                                                         19
    obj_<HSSFAnchor: int getDy1()>                                                         19
    obj_<HSSFAnchor: int getDy2()>                                                         19
    obj_<HSSFAnchor: void setDx1(int)>                                                     19
    obj_<HSSFAnchor: void setDx2(int)>                                                     19
    obj_<HSSFAnchor: void setDy1(int)>                                                     19
    obj_<HSSFAnchor: void setDy2(int)>                                                     19
    obj_<HSSFClientAnchor: int getAnchorType()>                                            19
    obj_<HSSFClientAnchor: int getRow1()>                                                  19
    obj_<HSSFClientAnchor: int getRow2()>                                                  19
    obj_<HSSFClientAnchor: short getCol1()>                                                19
    obj_<HSSFClientAnchor: short getCol2()>                                                19
    obj_<HSSFClientAnchor: void setAnchorType(int)>                                        19
    obj_<HSSFClientAnchor: void setCol2(short)>                                            19
    obj_<HSSFClientAnchor: void setRow2(int)>                                              19
```

(a)

```
private EscherContainerRecord createSpContainer(HSSFSimpleShape hssfShape, int shapeId)
{
    ...
    HSSFAnchor userAnchor = shape.getAnchor();
    if (userAnchor.isHorizontallyFlipped())
      ...
    if (userAnchor.isVerticallyFlipped())
      ...
    anchor = createAnchor(userAnchor);
    ...
  }
```

(b)

```
public static EscherRecord createAnchor( HSSFAnchor userAnchor )
{
    if (userAnchor instanceof HSSFClientAnchor)
    {
        HSSFClientAnchor a = (HSSFClientAnchor) userAnchor;
        ...
        anchor.setFlag( (short) a.getAnchorType() );
        anchor.setCol1( (short) Math.min(a.getCol1(), a.getCol2()) );
        anchor.setDx1( (short) a.getDx1() );
        anchor.setRow1( (short) Math.min(a.getRow1(), a.getRow2()) );
        anchor.setDy1( (short) a.getDy1() );
        ...
    }
    ...
}
```

(c)

図 17: Apache POI から抽出した複数のメソッド間に渡るメソッドの利用例

```
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 18: 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

6　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，一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では抽出できないような，複数のメソッドに渡って一まとまりで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．

また，制御的な接続関係に関しては，あるメソッドの戻り値を条件として別のメソッドを実行するといった接続関係を抽出できると期待したが，今回の実験で調査した 2,000 個からはそのような接続関係は抽出できなかった．実際にはそのような接続関係も抽出されていたが，メソッドの利用例として有用なものは含まれていなかった．

今回の実験結果から，本システムの別用途としてプログラムの欠陥検出への応用が考えられる．ソフトウェアの異なるバージョン間で接続関係を比較し，旧バージョンで一まとまりで利用されていた，すなわち接続関係の出現回数が同じだったメソッド・コンストラクタの組が，新バージョンではそのようになっていなかった場合，誤ったデータフローが形成されている可能性が考えられる．このような情報を自動的に抽出することができれば，ソフトウェアの異なるバージョン間におけるコンポーネント接続の誤りを検出し，開発者に警告することができる．あるいは，必ず同時に実行するべき操作に関するメソッドの接続関係から欠陥を検出することも考えられる．たとえば，open-close，start-end，enter-exit などの名を冠するメソッドの組は，必ず同時に利用しなければならない可能性が高い．このようなメソッドの組は，本システムの出力では出現回数が同じ接続関係の組として抽出されるはずである．実際に，今回調査した 2,000 個の接続関係にもそのような類のメソッドの組が見られ

た．すなわち，開発中のプログラムにおいてそれらのメソッドの出現回数が異なっていれば，open したが close し忘れている，あるいは逆に open していないなどの欠陥を容易に検出できると考えられる．

7　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，接続関係からメソッドの適切な利用例の情報を分析した．

実験では，用途の様々な 39 種類の小中規模なソフトウェアに対して，実装したシステムを適用し，抽出されたメソッドの接続関係からメソッドの利用例として適切な情報を分析できるかどうかを検証した．その結果，利用例として一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，一まとまりで利用することが多いメソッドの組のうち，いずれか一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考察の余地がある．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．これらを考慮するようになると，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

本研究の全過程を通して，常に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井上克郎教授に心より深く感謝致します.

本研究を通して，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申し上げます.

本研究を通して，常に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御礼申し上げます.

最後に，本研究にあたって，有意義な御指導，御助言および，適用実験へのご協力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ソフトウェア工学講座井上研究室の皆様に深く感謝致します.

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 James D. 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] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. 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.