修士学位論文

題目

データフロー情報を用いたコードナビゲーションツールの実装と評価

指導教員
井上 克郎 教授

報告者
悦田 翔悟

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

データフロー情報を用いたコードナビゲーションツールの実装と評価

悦田 翔悟

内容梗概

オブジェクト指向プログラミングでは，開発者が注目するコード片を理解するためには，関連する他のコード片を読解する必要がある．特にデータフローに関する調査など，プログラムの詳細を理解する作業では，ソースコード上での移動が多発する．ここで言うソースコード上での移動とは，開発者が現在閲覧しているコード片から，別のコード片を閲覧し始めるまでに必要な操作のことであり，例えば，ファイルを開き，カーソルを読解する行に移す一連の操作を 1 回の移動と考える．関連するコード片はプログラム中に遍在しており，ソースコード上での移動をツールによって支援する必要性が主張されている．これまで開発者の移動を支援するためのコードナビゲーションツールが多数提案されてきたが，データフロー調査に特化したツールは提案されていなかった．

そこで，本研究では，軽量なデータフロー解析技術を用いて，データフロー調査に特化したコードナビゲーションツールを提案する．具体的には，開発者がエディタ上で注目している識別子をクエリとし，その識別子を起点としたデータフローグラフを開発者に提示することで，複数のコード片を横断したデータフローの調査を支援する．

提案手法の有効性を確認するために，12 名の学生を対象に，プログラム理解作業を課題として，提案手法を実装したツールの有無による対照実験を行った．実験の結果として，ツールを使用した場合の方が，ツールを使用しなかった場合より，同一の作業時間でより多くのコード片を調査できることが分かり，提案手法と実装したツールの有効性を確認できた．

主な用語

プログラム理解 (Program Comprehension)
オブジェクト指向プログラミング (Object-Oriented Programming)
コードナビゲーション (Code Navigation)
可視化 (Visualization)
プログラムスライシング (Program Slicing)

目 次

1 はじめに

オブジェクト指向プログラミングでは，開発者が注目するコード片を理解するためには，関連する他のコード片へ移動し，読解する必要がある．例えば，注目しているメソッドの詳細を理解するためには，そこから呼び出されているメソッド，参照されているフィールドについて，それらの定義部や，他の参照箇所へ移動して，プログラム読解を行い，最初に注目していたメソッドとの関連性を調査する必要がある．なお，本論文では，コード片およびソースコード上での移動を以下のように定義する．

コード片 開発者が読解の対象とするソースコードの一部分とコメントなどの周辺情報．以下に例を示す．

- クラス定義
- メソッド定義，および内部
- フィールド定義

ソースコード上での移動 エディタ，その他ツールを用いて，開発者が現在閲覧しているコード片から，他のコード片を閲覧し始めるまでに必要な操作．例えば，ファイルを開き，カーソルを読解する行に移す，などである．

データフローに関する調査など，プログラムの詳細理解を必要とする作業では，ソースコード上での移動が多発し，保守作業における開発者の移動のコストが問題視されている [19, 23]．これまで開発者の移動を支援するために多数のコードナビゲーションツールが提案されてきたが，データフロー調査に特化したツールは提案されていなかった．

データフローなどプログラムの詳細を理解する時の支援手法として，プログラムスライシングを用いる方法が古くより提案されているが [11]，解析コストが高く，大規模ソフトウェアに対しては適用が難しいとされてきた．筆者らの研究グループでは，プログラムスライシングの近似計算手法を用いた軽量なデータフロー解析を提案しており [14]，本研究では，[14] の手法によって収集したデータフロー情報を用いて，データフローの調査支援に特化したコードナビゲーションツールを提案する．具体的には，開発者がエディタ上で注目している識別子をクエリとし，その識別子を起点としたデータフローグラフを開発者に提示することで，複数のコード片を横断したデータフローの調査を支援する．

提案手法の有効性を確認するためにツールとして実装し，12 名の学生を対象に，プログラム理解作業を課題として，ツールの有無による対照実験を行なった．実験の結果として，ツールを使用した場合の方が，ツールを使用しなかった場合より，同一の作業時間でより多くのコード片を調査できていることが分かり，ツールの有効性が確認できた．

本論文の構成は次のとおりである．まず，2 章で背景としてプログラム理解作業におけるソースコード上での移動が起こる要因，統合開発環境 Eclipse [7] や既存手法によるナビゲーション機能の紹介，およびデータフロー調査時の問題点について説明を行う．3 章では解析に用いる手法，および提案手法の詳細について説明し，4 章では実装したツールについて述べる．5 章では実装したツールを用いて行った適用実験とその結果，考察について述べ，6 章で関連研究を，最後に，7 章でまとめと今後の課題を記す．

2　背景

ソフトウェアの保守作業では，プログラム理解に多くの時間が費やされる．特に，開発者が不慣れなソフトウェアを保守対象とする場合や，プログラムの詳細な理解を必要とする作業では，ソースコードの閲覧と移動が作業時間の多くを占めると言われている [19, 23]．この要因として，多くのソフトウェアで用いられているオブジェクト指向プログラミングでは，開発者が注目するコード片を理解するために，そのコード片と関係のあるクラス，フィールド，メソッドなど，他のコード片を調査する必要があることが挙げられる．さらに，継承や多相性の適用など，オブジェクト指向特有の識別子間の関係を考慮する必要があり，移動にかかるコストが大きくなる．識別子間の関係で主要なものを以下に示す．

- **継承関係** クラスとクラスの関係．あるクラスには自身の親クラス，子クラスが存在しうる．オブジェクトが参照するメソッドやフィールドは，自身の親クラスから提供されている場合がある．
- **呼び出し関係** メソッド，コンストラクタなどクラス内のメンバとメンバの関係．各メンバは自身のコード片中で他のメンバを呼び出す，あるいは他のメンバのコード片中で呼び出される．メソッドの動作を理解するには，呼び出し関係のあるメソッドの内部を調査しなければならない場合がある．
- **変数のデータフロー** フィールド，ローカル変数，仮引数など変数と変数の関係．代入式を介して，右辺の変数の値が左辺の変数の値として定義される．変数がどのような値をもつかは，データフロー関係をもつ他の変数を調査する必要がある．

これらソースコード上に遍在する識別子間の関係を把握しながらプログラムを読解するには，ツールによる支援が必要だと指摘されている [24].

2.1　プログラム理解支援ツール

Storey らによると，プログラム理解支援ツールの機能は以下の 3 つのカテゴリに分類できる [24].

- **Extraction** 構文解析やデータ収集など．
- **Analysis** クラスタリング，コンセプトの対応付け，機能の特定，変換，ドメイン解析，プログラムスライシング，メトリクスの計算を行うための静的解析や動的解析など．
- **Presentation** ソースコードエディタ，ブラウザ，ハイパーテキスト，プログラムの可視化など．

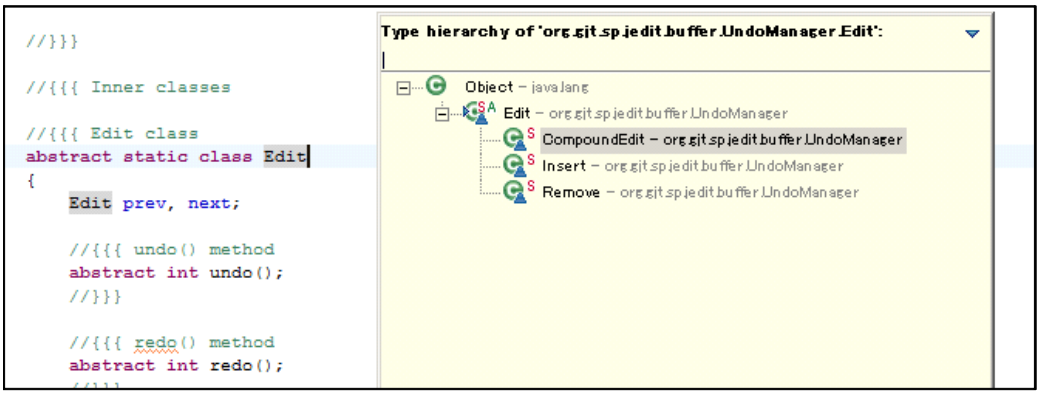

図 1: Eclipse におけるクラス階層の閲覧機能

統合開発環境やリバース・エンジニアリング用のツールでは上記の機能を複数もつこともある．本論文では，プログラム理解作業で最も利用される Presentation に特に注目する．以降では，Presentation に属するプログラム理解支援ツールの例として，2.1.1 項で統合開発環境 Eclipse のプログラム理解支援機能について，2.1.2 項で既存のコードナビゲーションツールについて説明する．

2.1.1 統合開発環境 Eclipse によるプログラム理解支援

Eclipse は代表的な統合開発環境の一つであり，エディタと連動した多彩なプログラム理解支援機能をもつ．

クラス階層の閲覧機能 Java プログラムでは継承を用いて，クラス間に親子関係をもつことができる．開発者はプログラムの動作理解時に動的束縛やオーバーライドなどを考慮しながらプログラム読解を行う必要があるため，注目するクラスと親子関係をもつクラスにも注意を払う必要がある．

Eclipse ではマウス操作やショートカットキーを用いてクラスの継承関係を閲覧することができる．図 1 は Eclipse 上で選択した Edit クラスに対して，Object クラスまでのスーパークラスと自身のサブクラスを階層的に表示している．Eclipse のクラス階層の閲覧機能はツリービューの形式をとっており，多層的な継承関係を容易に辿ることが可能である．クラス階層の閲覧機能を用いることで，注目しているクラスの継承関係を把握しながらプログラム理解を進めることができる．

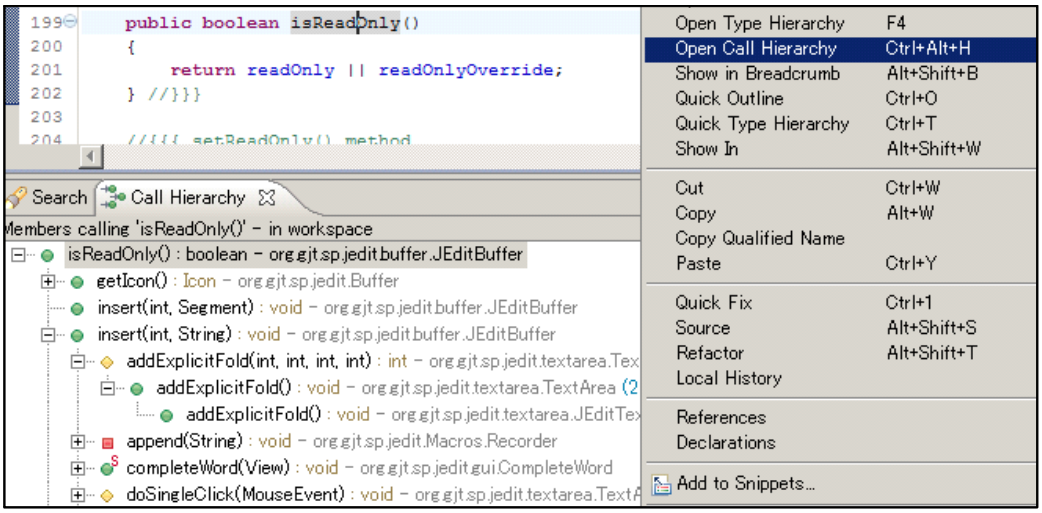

図 2: Eclipse における呼び出し元の閲覧機能

定義の閲覧，移動機能　Java プログラムでは，あるメソッドの動作を理解しようとする時，そのメソッドが呼び出しているメソッドについても，調査が必要な場合がある．Eclipse では，エディタ上でマウス操作やショートカットキーを用いて，呼び出されるメソッド定義へ容易に移動することができる．また，Declaration ビューでは，エディタ上で，注目するメソッド呼び出しにマウスカーソルを合わせることで，呼び出されるメソッド定義を閲覧することができる．Declaration ビューを使うことで，開発者は呼び出し先のコード片と呼び出し元のコード片を見比べたり，呼び出し先のコード片を詳細に読解する必要があるかどうか瞬時に判断することができる．これらの機能により，開発者のメソッドからメソッドへの移動の負荷が軽減される．

呼び出し関係の検索機能　Eclipse では，プログラム内で定義されているフィールド，メソッドやコンストラクタについて，それらが呼び出されている箇所をマウス操作やショートカットキーを用いて検索することができる．図 2 では，エディタ上で選択した isReadOnly() メソッドが参照されている箇所を検索し，一覧で表示している．クラス階層の閲覧機能と同様に，呼び出し関係の閲覧機能もツリービュー形式をとっており，推移的な呼び出し関係を簡単に辿ることが可能である．注目しているメソッドが，なぜ呼ばれたか，実引数として実際にどの型のオブジェクトが渡されるかなど，メソッドが呼ばれる原因や状況を調査する場合に有用な機能である．

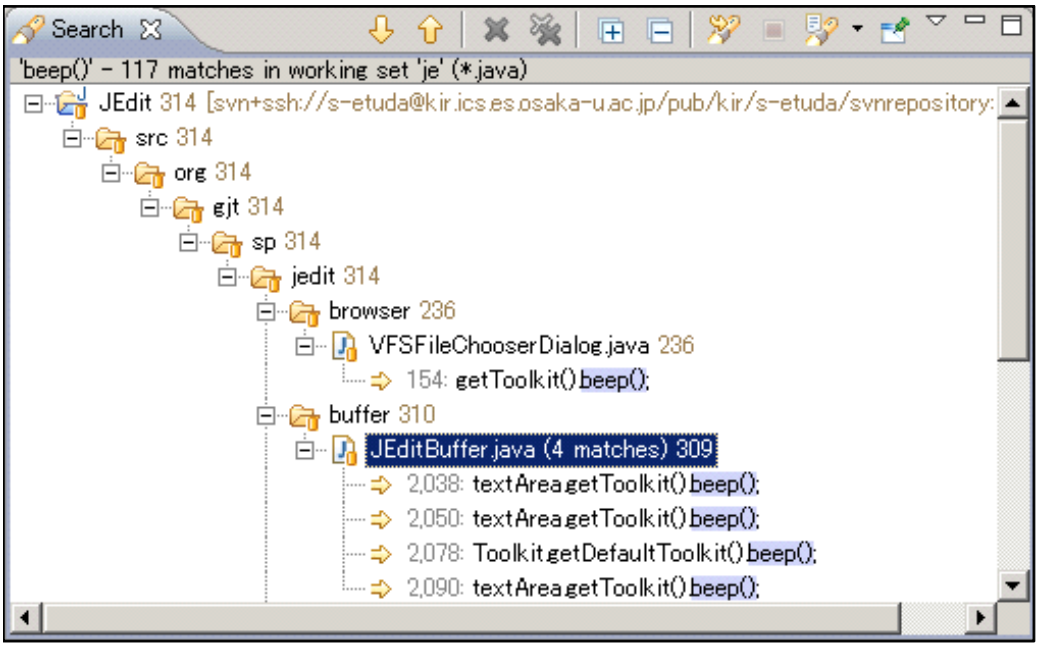

図 3: Eclipse における検索機能

検索機能　Eclipse では，ファイル内に登場する文字列を検索する機能に加えて，プログラム内で定義されている各識別子について検索する機能がある．図 3 は注目するプロジェクトに対して特定の文字列を検索した結果を表示している．結果はパッケージ階層に応じて出力される．注目する文字列に対してファイルを横断的に調査したい場合に有用な機能である．

識別子の強調表示機能　Eclipse のエディタ上で注目する識別子にマウスカーソルを合わせることで，ファイル内の同じ識別子が強調表示される．図 4 では，エディタ上で選択した変数 abbrev を強調表示している．この機能により，開発者は注目するコード片内で識別子が参照，変更されている箇所を瞬時に探し出すことができる．

統合開発環境 Eclipse を用いることで，開発者はエディタ上でのプログラムの読解に加えて，これらの理解支援機能を用いながらプログラム理解を進めることができる．

2.1.2　コードナビゲーションツール

本論文では，コードナビゲーションを，**プログラム理解のために，開発者の注目しているコード片から，関連する別のコード片への移動を支援する機能**とする．すなわち，2.1.1 項で述べた Eclipse によるプログラム理解支援機能は，コードナビゲーションの役割を果たす

```
String abbrev;

//{{{ Handle abbrevs of the form abbrev#pos1#pos2#pos3#...
if(lineText.charAt(pos-1) == '#')
{
    wordStart = lineText.indexOf('#');
    wordStart = TextUtilities.findWordStart(lineText,wordStart,
        buffer.getStringProperty("noWordSep") + '#');

    abbrev = lineText.substring(wordStart,pos - 1);

    // positional parameters will be inserted where $1, $2, $3, ...
    // occurs in the expansion

    int lastIndex = 0;
    for(int i = 0; i < abbrev.length(); i++)
    {
        if(abbrev.charAt(i) == '#')
        {
            m_pp.addElement(abbrev.substring(lastIndex,i));
            lastIndex = i + 1;
        }
    }

    m_pp.addElement(abbrev.substring(lastIndex));

    // the first element of pp is the abbrev itself
    abbrev = m_pp.elementAt(0);
    m_pp.removeElementAt(0);
} //}}}
//{{{ Handle ordinary abbrevs
```

図 4: Eclipse における識別子の強調表示機能

ものである．

また，これまでクラスの継承やメソッド，コンストラクタの呼び出し関係に注目して，多数のコードナビゲーションツールが提案されている．例を次に示す．

Fluid Source Code Views [6]　オブジェクト指向プログラムでは，メソッドの定義部と呼び出し部が互いに断片化し，動作を把握することが難しくなっている．Fluid Source Code Views では，メソッド呼び出し部から対応する定義部へ実行の流れを負担なく読めるように，ソースコード上のメソッド呼び出しで定義部の実装を折りたたみ表示する．これにより，必要に応じて，呼び出し先のメソッドを開いて読むことができ，動作理解をスムーズに行うことができる．

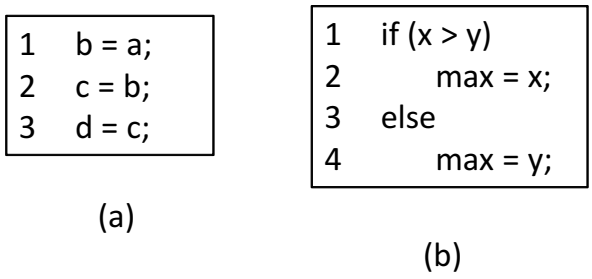

図 5: データフローの特徴，(a) 推移的な関係の例， (b)1 対多の関係の例)

Code Bubbles [4]　メソッドの動作を理解したい時は，メソッド呼び出し先のコード片を推移的に調査する必要がある場合や，呼び出し元と呼び出し先のコードを見比べたい場合がある．既存のエディタはファイル単位でプログラムを閲覧することを前提としており，複数のコード片を同時に見比べることには不向きだった．Code Bubbles では，開発者が注目するコード片をバブルとして定義し，バブル間の呼び出し関係を階層的，推移的に表示することが可能である．これにより，開発者は関連する複数のバブルを見比べながら，プログラム理解を進めることができる．

ソフトウェアナビゲーションマップ [27]　コードナビゲーションツールとして，プログラムの構造や，呼び出し関係を可視化して開発者に提示する手法が多数提案されている [3, 9]. 上原らは，プログラムの大局的な流れと処理の詳細の双方を理解するために，プログラム中のメソッドのクラス階層，メソッド呼び出し関係などを，各メソッドの重要度に応じて，多粒度で可視化する手法を提案している [27]．この手法ではメソッドの重要度をフィールドアクセスの有無，自身の複雑さ，現在注目しているメソッドとの関係性の強さなどを用いて計算し，開発者が次に注目するコード片を推測し，そのコード片を細粒度で，それ以外のコード片を粗粒度で可視化することで開発者をナビゲートしている．

これらの手法は，クラスの継承関係，メソッドの呼び出し関係に基づいて，開発者をナビゲートしている．プログラムの詳細理解が必要なタスクでは，変数のデータフローに注目して，開発者をナビゲートする必要があり，次節でこれについて述べる．

2.2　データフロー調査の支援

開発者がプログラムの詳細理解を必要とするタスクでは，注目するコード片に含まれる変数のデータフローを調査する必要がある．閲覧中のコード片内については，プログラムの読

解や Eclipse の識別子の強調表示機能を用いることで容易にデータフローを理解することができる．しかし，オブジェクト指向プログラムでは，メソッド呼び出しやフィールドアクセスなど，クラス，メソッドを横断したデータフローが存在するため，開発者は注目しているコード片の読解に加えて，関連するコード片へ移動し，移動先でプログラム読解を行う必要がある．

データフローには，次の 2 つの特徴がある．

- **推移的な関係** データフローは複数の代入文を介して，変数から別の変数，さらに別の変数に値が代入されるといった，推移的な関係が存在する．例えば，図 5 の (a) では，1 行目で変数 a の値が b に代入され，2 行目で b の値が c に，3 行目で c の値が d に代入されている．このコード片において，a の値は推移的に d に代入されている．

- **1 対多の関係** コード片内で複数のデータフローをもつことで，代入先，代入元がそれぞれ複数存在する．例えば，図 5 の (b) では，1 行目の条件式が真の場合は，2 行目で x の値が max に代入され，条件式が偽の場合は，4 行目で y の値が max に代入される．このコード片において，max は代入元を複数もつことになる．

推移的なデータフローを探索するためには，開発者はソースコード上での移動を何度も繰り返す必要がある．また，1 対多の関係により複数のデータフローパスを調査する必要があるため，ソースコード上での移動がプログラム理解作業における大きな負担となっている．そのため，移動の軽減に加えて，すべてのパスを網羅的に調査したり，調査すべきパスに優先順位をつけるための支援ツールが必要だと言われている [10]．Eclipse のプログラム理解支援機能や既存のコードナビゲーションツールは，データフローの調査支援に特化したものではなく，開発者に提示される情報はデータフロー情報よりも粒度が粗い．データフローの調査を支援するためにはコード片の内部を解析する必要があるため，より詳細なプログラム解析，可視化手法が必要である．

詳細な情報を扱うための支援手法として，プログラムスライシングを用いる方法が古くより提案されており [11]，保守プロセスにおける機能追加や変更，デバッグ支援に対して有効と言われている [12, 17, 22]．プログラムスライシングの課題として，解析コストの増加によるスケーラビリティの問題が指摘されており，大規模ソフトウェアに対して正確なプログラムスライシングを行うことは難しい．筆者らの研究グループでは，プログラムスライシングの近似計算手法を用いた軽量なデータフロー解析を提案しており [14]，本研究では，[14] の手法によって収集したデータフロー情報を用いて，データフローの調査支援に特化したコードナビゲーションツールを提案する．データフローの調査の起点は，開発者の注目する変数が変更された時点で新しいものとなる．よってプログラム理解作業中に，何度も細かいデータフローの調査が行われると考えられ，開発者は気軽にツールを使えることが望ましい．ま

た，Eclipse の既存のプログラム理解支援機能と本手法を併せて使用したいという要望から，本手法は Eclipse の plugin として実装している．これにより，エディタと本手法を連動させることができ，開発者はエディタ上でのプログラム読解作業中に，スムーズに本手法を利用することができる．開発者にデータフロー情報を提示するツールとして，Code Surfer があるが [1]，筆者らはオブジェクト指向言語の一つである Java で書かれた大規模ソフトウェアを解析対象としており，軽量なデータフロー解析手法を基盤として用いている点，Eclipse plugin として実装している点が異なる．

3　提案手法

本研究では，開発者が注目している識別子をクエリとし，その識別子を起点としたデータフローグラフを開発者に提示することで，複数のコード片を横断したデータフローの調査を支援する．はじめに，筆者の研究グループで提案している軽量なデータフロー解析について説明し，次に提案手法について述べる．

3.1　変数間データフローグラフ

筆者の研究グループでは，プログラムスライシングの近似計算手法を用いて軽量に構築することができる，変数間データフローグラフ (IVDFG：Inter-Variable Data Flow Graph) を提案している [14]．IVDFG 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 (PDG : Program Dependence Graph) 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・演算単位であること，制御フロー情報を考慮せずにグラフを構築するといったことが挙げられ，データフロー解析に特化したグラフとなっている．IVDFG の頂点には変数頂点，演算頂点，条件頂点の 3 種類がある．

変数頂点　プログラム中のローカル変数とメソッド・コンストラクタの仮引数に対応して生成される頂点．また，演算の中間データを表現するために，仮想的な変数頂点が生成される．グラフ上では，楕円形で表される．

演算頂点　+，∗ などの算術演算子，<，== などの比較演算子に代表される各演算子に対応して生成される頂点．グラフ上では，長方形で表される．

条件頂点　if 文や for 文などの制御文に対して一つ生成される頂点．グラフ上では，ひし形で表される．

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，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，プログラム文から得られるデータフローおよび制御情報を使って，変数頂点を接続していくことでで構築される．以降では，各文に対して構築される IVDFG について説明する．

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．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．

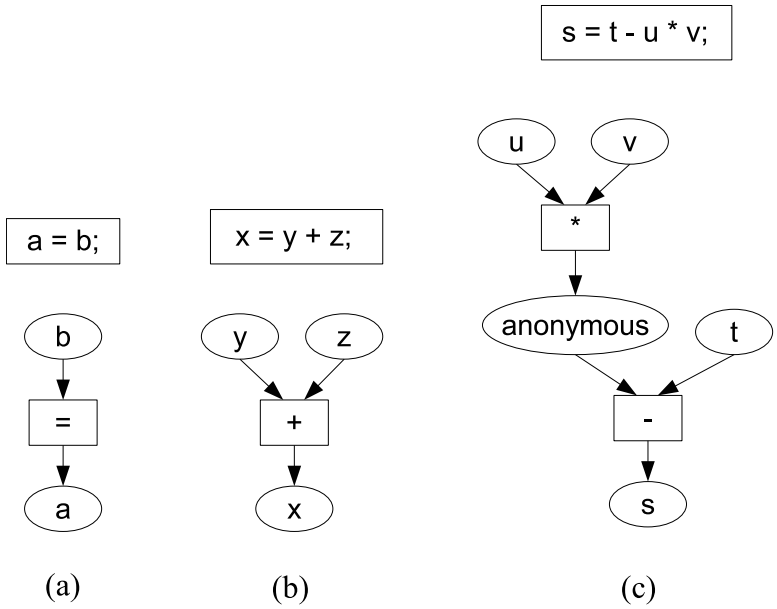

図 6: 代入文の IVDFG，(a) 直接代入文の IVDFG，(b) 二項演算を介する代入文の IVDFG，(c) 複数の二項演算を介する代入文の IVDFG

図 6(a) に直接代入文の IVDFG の例を示す．図 6(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．図 6(b) に二項演算を介する代入文の IVDFG の例を示す．図 6 (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．図 6 (c) に右辺で 2 個の二項演算を介する代入文の IVDFG の例を示す．図 6 (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 という仮想的な変数頂点を生成してこれに接続する．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．右辺に 3 個以上の二項演算がある場合も同様に行う．

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 などの条件頂点が一つ生成される．その後，制御文の条件式から条件頂点へ辺を接続する．制御文はそれぞれ条件式の評価結果が真の時，偽の時に実行されるブロック，または文を個別に持つが，そのブロック，文の中に登場する演算の集合，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，評価結果によって実行されるかどうかが決定するため，条件頂点から各演算に対して制御辺を接続する．

図 7 に制御文の IVDFG の例をいくつか示す．たとえば，図 7 (a) は if 文の IVDFG である．図 7 (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

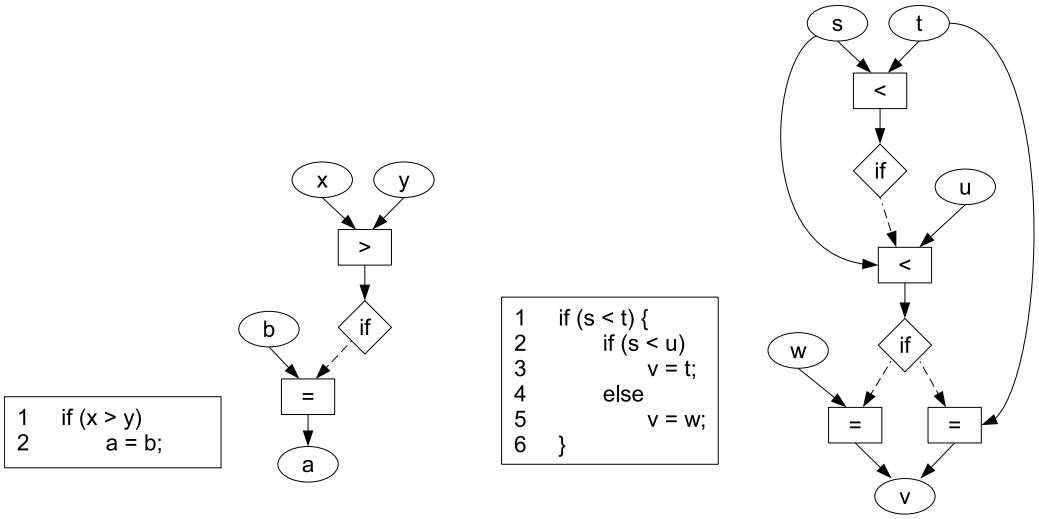

図 7: 制御文の IVDFG，(a) if 文の IVDFG，(b) 入れ子構造になっている if 文の IVDFG

おり，if から条件式の影響を受ける文の演算子 “=” に対して制御辺を接続する．なお，図 7(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 7(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．

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，メソッド呼び出しではインスタンス変数を表す．また，コンストラクタ呼び出しでは生成されるオブジェクトのクラスを表す．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．戻り値が void のメソッドの場合は，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．

図 8 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，コンストラクタの戻り値 ret は代入文における右辺値となっているため，“=” を介してローカル変数 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

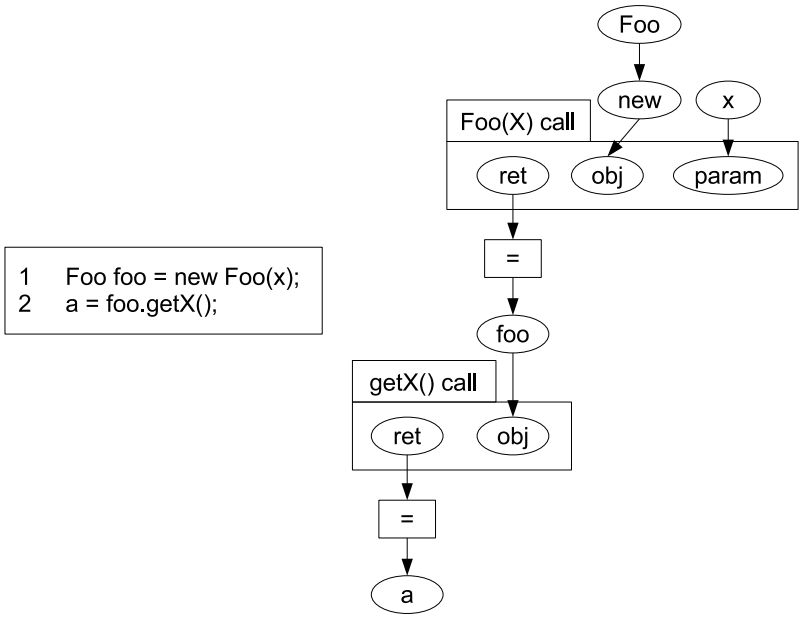

図 8: メソッド・コンストラクタ呼び出しの IVDFG

foo をレシーバとしてメソッド getX() を呼び出しているため， foo からのデータフロー辺を 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．

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を，val はフィールドの値をそれぞれ表す.

メソッド，コンストラクタの定義の 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，メソッドの戻り値を表現する仮想的な変数頂点 $return$ を生成し，return 文によって返される式から辺を接続する．$return$ は変数頂点の一種として扱う．図 9 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．

また，メソッド・コンストラクタ呼び出しとメソッド・コンストラクタ定義の間のデータフローを表すための辺を接続する必要がある．図 10 にメソッド呼び出しとメソッドの接続例を示す．まず，図 10 (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 10 (b) の 5 行目を見ると，メソッドの仮引数は z であること

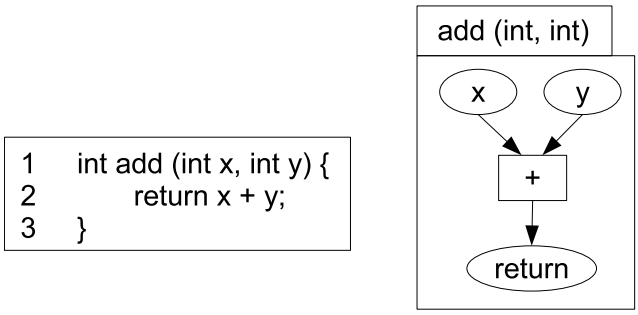

図 9: return 文があるメソッド定義の IVDFG

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，図 10(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，図 10(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 obj へ辺を接続する．最後に，図 10(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 グラフの可視化

本手法では，3.1 節で示した変数間データフローグラフ全体の中から開発者が注目している変数に関わるノード，エッジのみを抽出し，可視化する．また，可読性を向上させるために，クラス，メソッド，コンストラクタ定義でノードをグループ化する．

細粒度のグラフを可視化する際には，ノード，エッジ数が多くなり，可読性を下げることが課題とされている [20]．そこで，可視化するノードの種類，数に制限を加えることでグラフの可読性を高める．ここでは，可視化するノードの種類についてのみ述べ，次節以降でノード数の制限方法について述べる．

ノード数を削減するために，以下のノードのみを可視化し，残りのノードは省略する．

- フィールド
- メソッド，コンストラクタ呼び出し
- 配列操作
- ローカル変数
- 仮引数

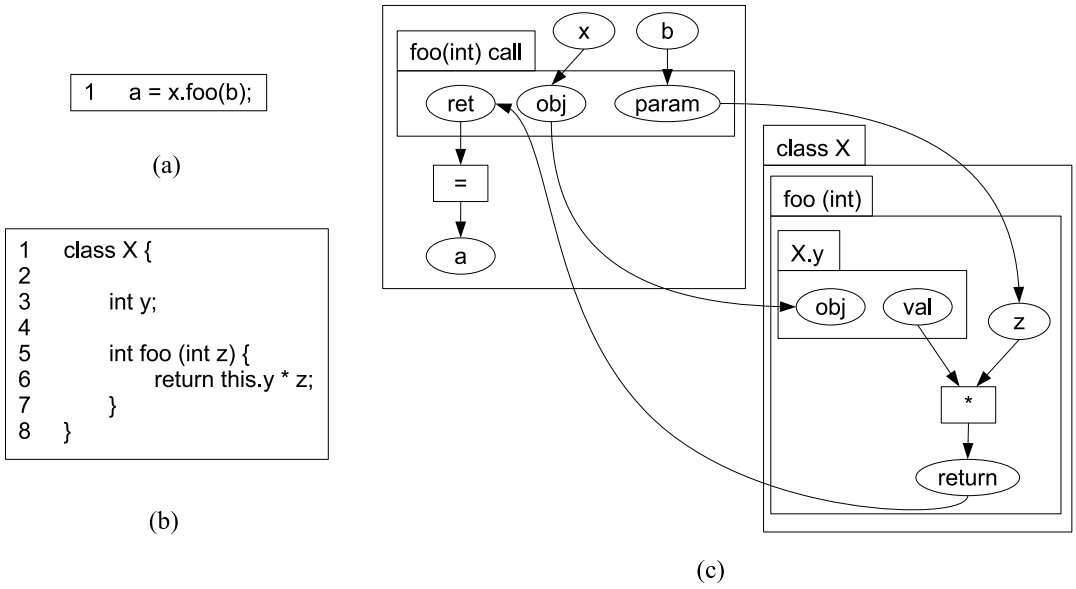

図 10: メソッド呼び出しとメソッドの IVDFG，(a) メソッド呼び出し，(b) 呼び出されているメソッド，(c) メソッド呼び出しとメソッドの接続

- リテラル
- $return$ ノード
- 条件頂点

演算頂点については，エッジにラベルとして記述することで，識別子間でどのような演算が行われたか示す.

3.3 グラフの探索

変数間データフローグラフの中から，開発者が注目している識別子をクエリとして，データフローの探索を行い，可視化すべきノードを見つける．開発者はエディタ上で簡単なマウス操作を行うことで，可視化したい識別子を選択し，クエリを発行する．探索には，データフロー辺の順方向に辿ってデータの代入先を探索する Forward 探索，データフロー辺を逆方向に辿ってデータの代入元を探索していく Backward 探索の 2 種類を用いる．

開発者が注目しているコード片内のデータフローは，エディタ内でプログラム読解を行うことで調査が可能なため，注目しているコード片以外のノードとデータフローをもつ可能性があるノードを探索のクエリとする．したがって，メソッド，コンストラクタ定義，フィー

ルド，メソッド，コンストラクタ呼び出しが該当する．以降では，クエリとなる各ノードに対して，それぞれの探索方法について述べる．

メソッド，コンストラクタ定義　開発者があるメソッド，コンストラクタを読解中に，他のコード片とのデータフロー調査を行いたい時にクエリとして使われる．したがって，このクエリに対して調査すべき探索の起点となるノードは，仮引数，メソッド，コンストラクタ内で参照するフィールド，代入するフィールド，戻り値である．以下，それぞれのノードを基点とする探索方法について述べる．

- **仮引数** 仮引数に何が代入されるかは，閲覧中のコード片から読み取ることはできないため，Backward 探索を行う．
- **参照するフィールド** 参照したフィールドに何が代入されていたかは，閲覧中のコード片から読み取ることはできないため，Backward 探索を行う．
- **代入するフィールド** 代入したフィールドがどこで使用されるかは，閲覧中のコード片から読み取ることはできないため，Forward 探索を行う．
- **戻り値** 戻り値がどこで使用されるかは，閲覧中のコード片から読み取ることはできないため，Forward 探索を行う．

フィールド　開発者が注目しているコード片に含まれるフィールドと他のコード片とのデータフロー調査を行いたい時にクエリとして使われる．よって，注目するフィールドを起点として Forward 探索，Backward 探索の両方を行う．

メソッド，コンストラクタ呼び出し　開発者が注目しているコード片に含まれるメソッド，コンストラクタ呼び出しと呼び出されるメソッド，コンストラクタ定義とのデータフロー調査を行いたい時にクエリとして使われる．したがって，このクエリに対して調査すべき探索の起点となるノードは，実引数，メソッド，コンストラクタ呼び出しの戻り値である．以下，それぞれのノードを基点とする探索方法について述べる．

- **実引数** 実引数がどこで使用されるかは，閲覧中のコード片から読み取ることはできないため，Forward 探索を行う．
- **メソッド，コンストラクタ呼び出しの戻り値** 戻り値がどのように計算されるかは，閲覧中のコード片から読み取ることはできないため，Backward 探索を行う．

3.3.1 探索の打ち切り

2.2 節で述べたように，ある変数は他の多くの変数とデータフローの関係をもつ場合があり，可読性を確保するためには，ノード数の合計を一定数以下に保つ必要がある．本研究ではフラクタルビュー [16] に基づいた，以下のルールを用いて探索を打ち切ることで，提示するノードを一定数以下に保つ．

1. 探索の起点となるノードのポイントを 1 とし，最初の親ノードとする
2. 親ノードとデータフローがある，かつ未探索のノードを子ノードとみなしポイントを計算する
3. 子ノードのポイントが閾値以上なら，新たな親ノードとみなす
4. 親ノードが新しく検出さなくなるまで，２．３．を繰り返す
5. 親ノードとして検出されたノードを可視化する

ノードのポイントには [16] で用いらている fractal value を用いた．fractal value の算出方法は，以下の式で表される．

$$\begin{cases} F_{v_{focus}} = 1 \\ F_{v_{child\_of\_x}} = r_x \times F_{v_x} \end{cases} \qquad r_x = CN_x^{-1/D}$$

$F_{v_{focus}}$ は起点となるノードのポイント，$F_{v_{child\_of\_x}}$ は親ノード F_{v_x} の子ノードのポイント，C は減衰定数，N_x は親ノードがもつ子ノードの数，D はフラクタル次限定数である．本手法では，C，D ともに値を 1 としている．グラフを N 分木とみなし，閾値を F_v とした場合，グラフ全体のノード数 M は以下の式で表される．

$$M = \frac{F_v^{-1} - 1/N}{1 - 1/N}$$

この式より，各ノードのエッジ数とノードの全体数を決めることで，閾値を決定することができる．例えば，エッジ数の平均を 5 と仮定し，ノードの全体数 30 程度にしたい場合は，閾値は 0.04 に設定すればよい．

3.4 コードナビゲーション

本手法における，コードナビゲーションの手順について述べる．開発者は注目するコード片内のデータフローについては，エディタ上でソースコードを読解することで把握することができる．他のコード片とのデータフローについては，注目しているコード片内の識別子をクエリとして，グラフを表示し，グラフ上でデータフローを調査する．グラフ上でのデータフローの調査は，以下の点でエディタ主導の移動が伴う調査に比べて有利である．

- 代入の推移的な関係をソースコード上で移動することなく把握することができる．移動の負担削減に加えて，注目しているデータフローの末端をチェックすることで，調査の結果が開発者が求めるものに関係するか予測でき，調査の優先順位をつけることができる．

- 代入の呼び出し先，呼び出し元が複数ある場合，分岐が起こったグラフを保存，再度閲覧することで，網羅的な探索を行うことができる．また，オーバーライドを考慮してデータフローを表示するため，クラスの継承関係，オーバーライドの有無の調査の手間が省ける．

- 注目するコード片に関わるクラス，メソッド，フィールドが多数表示されるため，より抽象度が高いレベルでプログラム同士の関係性を把握することができる．

Pinzger らによる調査によると，ソフトウェアをグラフ構造として可視化する際，最初はグラフの情報量を少なめにして提示し，開発者の注目箇所に応じて，グラフを拡張し，情報量を増やす手法が有効だと言われている [20]．本手法では，グラフ上のメソッド，コンストラクタ定義，フィールド，メソッド，コンストラクタ呼び出しを新たなクエリとして開発者が選択することで，グラフの拡張を行う．また，グラフ上の注目する識別子をエディタ上で確認できるようにするために，簡単なマウス操作でノードに対応するソースコードをエディタで閲覧できるようにする．これらの機能より，開発者はエディタとグラフの双方を使ったデータフロー調査を行うことができる．

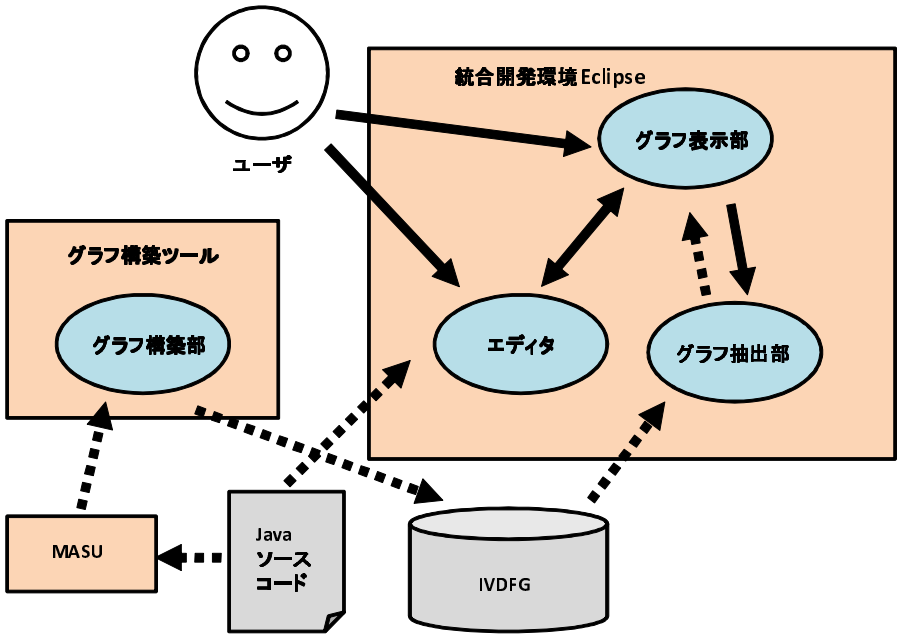

図 11: 本研究で実装したツールのアーキテクチャ．実線は制御の流れを，点線はデータの流れを表す．

4　実装

提案手法をツールとして実装した．ツールはグラフ構築部，グラフ抽出部，グラフ表示部からなり，グラフ構築部は既存ツールを改良し [12]，グラフ抽出部，グラフ表示部は Eclipse plugin として新たに実装した．ツールのアーキテクチャを図 11 に示す．

4.1　グラフ構築部

グラフ構築部では，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，メトリクス計測プラグインプラットフォーム MASU [26] のソースコード解析モジュールを利用して，ソースコードの解析を行う．その後，MASU の解析結果から構築ルールに従って文単位で IVDFG の構築を行う．

本手法では，以下の点の改良を行った．

- IVDFG をクラス階層に基づいて構造化した．
- クラスノードに継承関係情報を，メソッドノードにオーバーライド情報を付加した．
- グラフ抽出部，表示部で使用する諸情報を付加した．

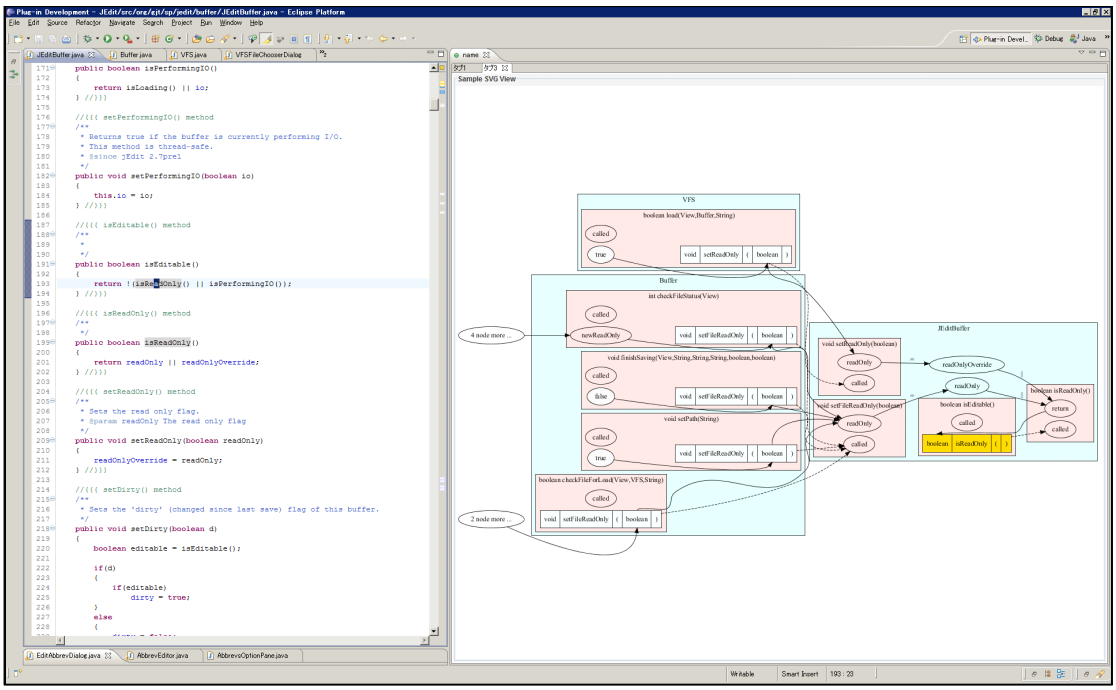

図 12: グラフ表示部：エディタ上で JEditBuffer クラスの isEditable() メソッド呼び出しを選択し，グラフを表示した様子.

4.2 グラフ抽出部

グラフ表示部から渡されたクエリを起点として，IVDFG の全体からユーザに提示する部分グラフを抽出し，グラフ表示部へ渡す．具体的には，部分グラフを表現する SVG ファイル [25] をグラフ可視化ツール Graphviz[8] を使って生成し，グラフ表示部へ渡している．

4.3 グラフ表示部

本ツールでは，グラフを表示したい識別子をエディタ上でダブルクリックするか，右クリックでメニューを開き，Open Graph の項目を選択することで，識別子情報をグラフ表示部へ渡すことができる．グラフ表示部は識別子情報を渡されると，探索用のクエリを生成し，グラフ抽出部へ渡す．その後，グラフ抽出部から返された部分グラフを，オープンソースの SVG 描画ライブラリ Batik [2] を使って表示する．

ツールのスクリーンショットを図 12 に示す．図 12 は，エディタ上で JEditBuffer クラスの isEditable() メソッド呼び出しの識別子をダブルクリックして，グラフを表示した場面である．グラフ上では，クエリとなった isEditable() メソッド呼び出しノードが黄色く強調表示されている．

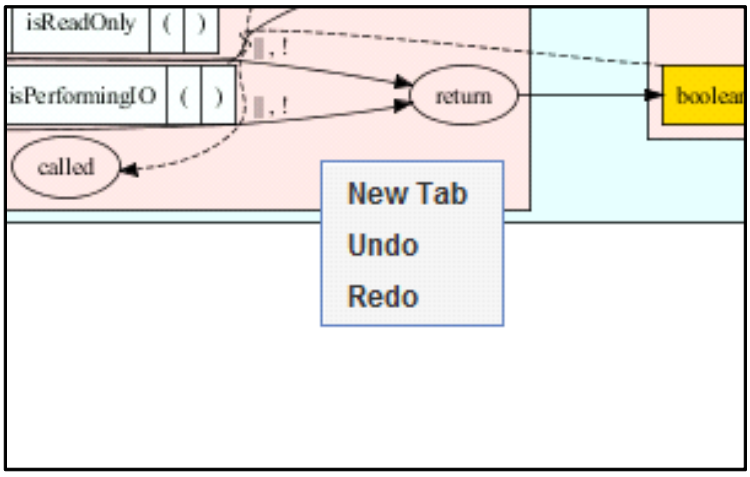

図 13: グラフ表示部：右クリックメニュー

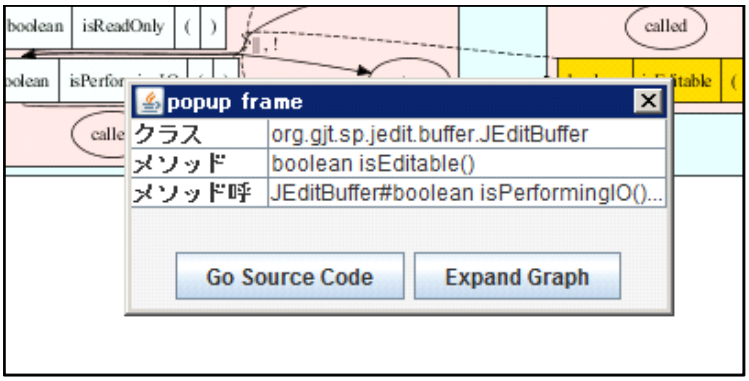

図 14: グラフ表示部：中クリックメニュー

グラフはタブ付きのビューで開かれ，マウスで操作でき，左ドラッグで移動，ホイールスクロールで拡大縮小する．また，グラフ上でマウスの右ボタン（図 13），ノード上でマウスの中ボタン（図 14）をクリックすることで以下のメニューを開くことができる．

右クリックメニュー

- **NewTab** 現在開いているグラフをコピーして，新しいタブで開く．
- **Undo** 一つ前の状態のグラフへ戻す．
- **Redo** Undo で戻った状態の時に，Undo を実行する以前の状態へグラフを戻す．

中クリックメニュー

- **Go Source Code** 選択されたノードに対応する識別子をエディタで開く.
- **Expand Graph** 選択されたノードをクエリとし，新しくグラフを開く.

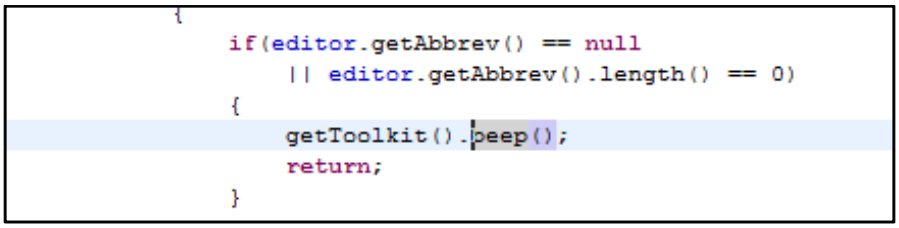

図 15: 実験課題：EditAbbrevDialog クラスに現れる beep() メソッド

5　適用実験

データフロー調査に対する提案手法の有効性を検証するために，プログラム理解作業において，実装したツールの有無によりソースコード上での移動が支援されるかどうか実験を行った．

5.1　実験課題

課題内容　実験では，課題として Java で書かれたオープンソースソフトウェアの jEdit 4.3.2 [13] を対象にし，ビープ音が鳴る原因を調査してもらった．jEdit では，図 15 に示すように，beep() メソッドが呼び出されることでビープ音が鳴る．よって，課題では beep() メソッドが呼ばれる原因，すなわち直前の条件文が true になる原因を調べる．被験者はこの時，editor.getAbbrev() のメソッド定義を参照し，戻り値として null か，空文字を返す条件を調査することになる．このように，被験者はデータフローを逆上りながら，原因となる箇所を調査していく．解答用紙には，調査の結果，最終的に原因と考えた箇所に加えて，調査の途中で参照したメソッド，フィールドについても記述してもらった．なお，課題によっては原因箇所が複数考えられる場合もある．

実験の準備　beep() メソッドは jEdit 4.3.2 のソースコード内の 117 箇所から呼ばれている．事前に筆者が原因の調査を行い，同程度の難易度，作業量となるような課題を 2 つ用意し，それぞれを課題 a，課題 b とした. 課題として用いた beep() メソッド呼び出しを以下に示す.

課題 a　EditAbbrevDialog クラスの内部クラス ActionHandler，actionPerformed(ActionEvent) メソッド，153 行目の beep()

課題 b　JEditBuffer クラス，undo() メソッド，2043 行目の beep()

それぞれの課題を，被験者としてソフトウェア工学講座に所属する大学院生 9 名，大学生 3 名の計 12 名に解答してもらった．被験者は調査環境として，24 インチ，解像度 1920 ×

1200 ピクセルのディスプレイを使い，作業は Eclipse3.6 で，他に解答用紙とボールペンを与えて作業を行った．ツールを用いることで，ソースコード上での移動が支援されるかどうかを評価するために，上記環境に加えてツールを使用した場合と，使用しなかった場合の 2 パターンで作業を行った．ただし，ツールを使用する場合，対象ソースコードの解析，グラフの構築は事前に行った状態で作業を始めた．課題に対する学習効果などを考慮し，被験者と課題，ツール使用の有無は表 1 ように割り当てた．課題の詳細については，付録として載せている．

実験の進め方　はじめにツールの使用方法の説明，タスクの説明，例題を用いた練習を合わせて 30 分で行った．その後，課題 a，課題 b をそれぞれ表 1 の順番で 30 分ずつ解答してもらった．

5.2　評価基準

本実験では，被験者がどの程度データフローを調査できたかを評価する．まず，筆者がビープ音が鳴る原因箇所を定め，beep() メソッドが呼ばれる探索の起点から原因箇所までデータフロー辺を逆向きに辿るパスを正解パスと定めた．被験者がそれぞれの課題で，時間内に正解パスをどの程度調査できたかを採点し，評価する．

1 つの課題について複数の正解が出現する場合は，すべての正解集合を解答できれば満点とする．解答の数が足りない，探索パスの途中までしか解答できていない場合は，部分点を与える．得点は 0 から 1 の実数とし，その算出方法は以下の式のとおりである．

$$Score = \sum_{v \in V} weight(v, m) \frac{|A \cap path(v, m)|}{|path(v, m)|}$$

表 1: 学生の課題の割り当て表

作業を行った被験者	課題 1 回目	課題 2 回目
被験者 1，2，3	課題 a（ツール有り）	課題 b（ツールなし）
被験者 4，5，6	課題 a（ツールなし）	課題 b（ツール有り）
被験者 7，8，9	課題 b（ツールなし）	課題 a（ツール有り）
被験者 10，11，12	課題 b（ツール有り）	課題 a（ツールなし）

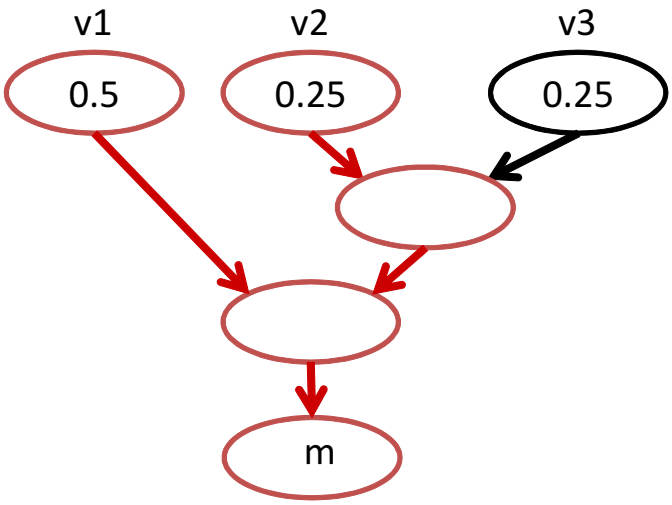

図 16: データフロー調査の例，赤いエッジが調査済みのパスを表す．

$weight(v, m)$ は，3.3.1 節で述べた fractal value [16] を用いる．A は被験者が解答したパスの集合，V は原因箇所の集合，m は探索の起点，$|path(x, y)|$ はノード x から y までのデータフローパスに含まれるエッジの数である．

図 16 にデータフロー調査の例を示す．この例では，ノード m を基点とする backward 探索を行った結果であり，正解のパスとして v1 から m まで，v2 から m まで，v3 から m までが存在する．赤いエッジが探索されたデータフローパスを，黒いエッジが探索されなかったデータフローパスを表している．fractal value の定義に従って原因箇所のノードの weight を算出すると，v1 は 0.5，v2，v3 は 0.25 となる．m から v1，m から v2 のデータフローパスは，すべてのエッジを探索できているので，それぞれ 0.5，0.25 のスコアが与えられる．m から v3 のデータフローパスは，途中までしかパスを探索できていないので部分点として $0.25 \times 2/3 = 0.16$ のスコアが与えられる．よってトータルスコアは $0.5 + 0.25 + 0.17 = 0.92$ となる．

評価方法として，課題を完了させるまでにかかった時間を比較する方法もあるが，被験者の実力によって解答時間に大きな差が生じ，ツールの有無よりも大きな影響を与える可能性がある．データフローの調査範囲を採点する場合，例え被験者が時間内に課題を完了させることができなくても，部分点を与えることができるので，被験者の実力の差による影響を抑えることができる．また，制限時間を設けることで，課題に集中して取り組むことが見込める．以上の理由から，実験ではデータフローの調査範囲を視点することで評価を行った．

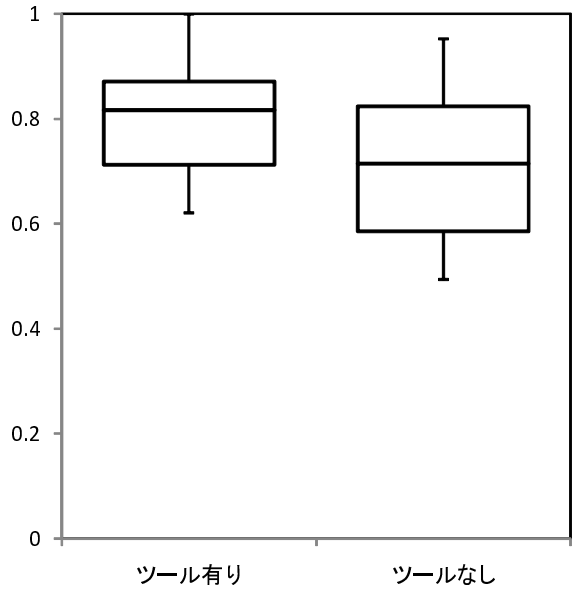

図 17: スコアの分布

5.3 結果

表 1 に従って，学生 12 人に作業を行ってもらったところ，各課題のスコアは表 2 に示す結果となった．被験者のスコア分布を図 17 に示す．表 2，図 17 に示すようにツールを使用した場合の方が，ツールを使用しなかった場合よりも，スコアの平均値が高くなるという結果が得られた．

この違いが統計的に有意であるかどうか調べるために，ノンパラメトリック検定の一種であるウィルコクソンの符号順位和検定を行った．帰無仮説，対立仮説は以下のように設定し，有意水準 0.05 で片側検定を行った．

帰無仮説 ツールを使用した場合と，使用しなかった場合では，スコアに差はない

対立仮説 ツールを使用した場合の方が，使用しなかった場合よりも，スコアが高い

検定結果は，P 値が 0.009 となり 0.05 以下であるため，帰無仮説を棄却できる．よって，有意水準 0.05 において，ツールを使用した場合の方が，ツールを使用しなかった場合よりも，スコアが高い傾向にあることが分かった．

5.4 考察

作業の全体を通して，被験者はツールが表示するグラフを用いてデータフロー調査を行っていた．よって，本実験においてツールの有無による，調査方法には大きな差が生じた．

5.4.1 ツールの有効性について

課題 a では null が代入される箇所を探索する作業が必要だったが，開発者はグラフ上に null が出現しているのを見つけると，起点へのデータフローが起こりうるか調査するなど，現在注目している条件に応じて，優先順位をもって調査を行えていた．それに比べて，ツールがなしの時にスコアが悪かった被験者は，複雑なロジックを含むパスの調査や，不正解のパスの調査に時間がかかっており，正解のパスを十分に調査する時間が不足する場合が多かった．このように，特定の値や，特定のクラス，フィールドなど，調査の目標が定まっている作業の場合などに，探索の優先順位をつけるという意味で本ツールは有効性を発揮した．

パスが分岐している箇所では，タブ機能を用いて閲覧中のグラフの状態を保存しておくことで，網羅的にすべてのパスを調査することができていた．被験者はあるパスの調査を終了

表 2: 各被験者のスコア

作業を行った被験者	スコア	
	ツール有り	ツールなし
被験者 1　（課題 a：有り．　課題 b：なし）	0.857142857	0.78125
被験者 2　（課題 a：有り．　課題 b：なし）	1	0.722916667
被験者 3　（課題 a：有り．　課題 b：なし）	1	0.620833333
被験者 4　（課題 a：なし．　課題 b：有り）	0.875	0.857142857
被験者 5　（課題 a：なし．　課題 b：有り）	0.708333333	0.428571429
被験者 6　（課題 a：なし．　課題 b：有り）	0.620833333	0.571428571
被験者 7　（課題 b：なし．　課題 a：有り）	0.733333333	0.714285714
被験者 8　（課題 b：なし．　課題 a：有り）	0.858333333	1
被験者 9　（課題 b：なし．　課題 a：有り）	0.816666667	0.714285714
被験者 10　（課題 b：有り．　課題 a：なし）	0.714285714	0.589583333
被験者 11　（課題 b：有り．　課題 a：なし）	0.857142857	0.722916667
被験者 12　（課題 b：有り．　課題 a：なし）	1	0.908333333
平均値	0.836755952	0.719295635
中央値	0.857142857	0.71860119

した後，直前の分岐まで戻る必要がある．グラフがある場合は容易に直前の分岐に戻ることができていたが，グラフがない場合は探索の起点である beep() メソッドに戻ってから，直前の分岐まで，一度辿ったパスをもう一度辿り直す被験者が多かった．特に，作業後半になってエディタ中で開いているファイル数が多くなっていた場合や，ファイルの行数が多くて，先程まで閲覧していたコード片を見つけるのが手間な時に，このような行動が目立った．開発者は，ソースコード上で移動を行って注目するコード片が変わると，先程まで調査した内容を忘れてしまう場合があると言われている．本実験でも，多くの被験者が直前の分岐に戻った時や，複雑なロジックを読解した後に，自分が先程までどのような作業を行っていたか，どのような探索条件に注目していたかを失念する場面が見られた．「**私はさっきまで何をしていたのだっけ？**」「**私はここで何を調査するはずだっけ？**」と被験者はたびたび呟いていた．この時，被験者は再確認のために探索してきたパスを見直すという行動を取るのだが，ツールが有りの場合では，グラフを使って自分が辿ってきたパスを確認することで，容易に確認作業が行われていた．このように，分岐が起こるデータフローを網羅的に調査する場合や，自分が探索してきたパスの再確認を行う作業で本ツールは有効性を発揮した．

5.4.2 被験者のスキルとスコアについて

被験者の合計スコアを比較すると，最大で 1.8 倍の差が見られた．プログラムの詳細理解が必要な作業では，開発者は対象プログラムに関する自身のメンタルモデルに基づいて，仮定ベースで調査を行うと言われており [23, 28]，熟練者と初級者はプログラムの調査方法や理解スピードに大きな差があると指摘されている [18].

本実験の課題でも開発者の開発経験やドメイン固有の知識の有無によって，スコアに大きな差が見られた．特に今回の課題では，GUI 部品に関する知識の差異が大きな影響を持っていた．また，被験者の中でも熟練の開発者は，Eclipse の操作方法を確立しており，ツールがない状態でも，高速でデータフロー調査を進めており，ツールの有無に関わらずスコアが高くなった．

5.4.3 制御フローを考慮しないことによる影響

IVDFG では，制御フローを考慮せずにグラフを構築している．そのため実際にはデータ依存関係がないノードの間にもエッジが引かれる可能性がある．図 18 に例を示す．例の場合では，z の値が x に代入されることはないが，IVDFG 上ではグラフのように推移的なデータフローが存在しているように表される．このように，ある代入文の代入元の変数が，以後の代入文で別の値を代入されると IVDFG のグラフで，実際には起こりえない推移的なデータフローが表示される．

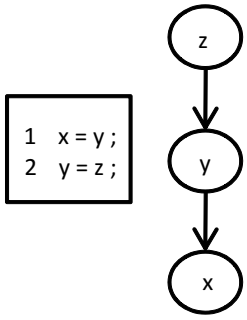

図 18: 実際には起こりえない推移的なデータフローの例

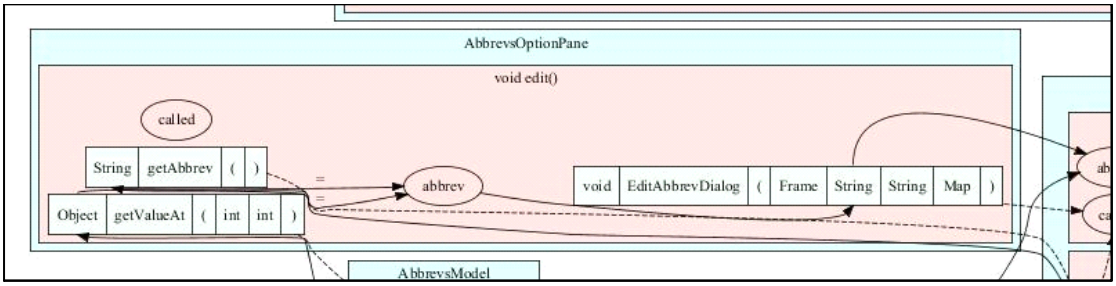

図 19: 実際には起こりえない推移的なデータフローの実例：グラフ

本実験の調査に使われたグラフ上では，このような例が 1 件存在した．該当箇所を図 19, 図 20 に示す．図 20 では，EditAbbrevDialog(Frame, String, String, Map) コンストラクタの第二引数の abbrev には，コンストラクタ呼び出しの後，dialog.getAbbrev() メソッドの戻り値が代入されているのが分かる．図 19 のグラフ上では，abbrev を介して，dialog.getAbbrev() から EditAbbrevDialog コンストラクタ呼び出しに推移的なデータフローがあるように表されている．課題では，本来 dialog.getAbbrev() メソッドを調査する必要はないが，ツールを使った被験者 8 名の中で 1 名だけが，詳しく調査をしていた．その他の被験者は，edit() メソッドのソースコードを閲覧して，制御フローを確認していたために，dialog.getAbbrev() メソッドを調査しなかった．該当箇所が，探索の終盤であったこと，メソッド呼び出しの戻り値が絡む複雑な処理だったことから，被験者はソースコードを確認したと考えられる．

5.4.4 ツールの課題

グラフを使ったデータフロー調査で，新しいメソッドに到達するたびに，グラフ上での調査を中断し，エディタでソースを確認している被験者がいた．本ツールの使用目的が移動コストの削減ということから，本来は，委譲メソッドなど単純なメソッドはエディタ上での確

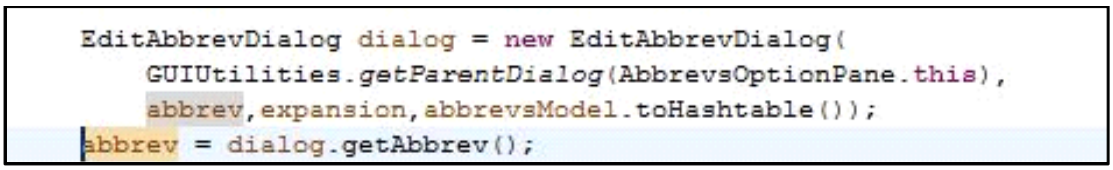

図 20: 実際には起こりえない，推移的なデータフローの実例：ソースコード

認なしに，グラフ上でのデータフロー調査を続けることが望ましい．一方で，制御フローが考慮されないことによって実際には起こりえない推移的なデータフローが表示される可能性が存在するので，全くソースコードを見ずに探索を行うのも危険である．どのようなグラフを閲覧している時にソースコードを確認した方が良いかは，今後検討する必要がある．対策の一例として，グラフ上でソースコードをポップアップとして表示するなど，簡単に閲覧できる機能があれば，過剰なソースコードの確認，推移的な代入が正しいかどうかの判定が容易に行えると考えれる．また，ローカル変数を介した代入を表すエッジでは，代入された行数を表示するなどの対策も考えられる．

また，本手法ではグラフ操作のほとんどをマウスで行うものであったが，キーボードによるショートカットキーを対応付けるなど，ユーザビリティの改善を求める被験者もいた．

また，探索を行うクエリの拡張や探索範囲を限定することにより，開発者の移動のコストを更に削減できると考えている．例えば，副作用調査のように，あるメソッド呼び出しによって変更を受けるフィールドを調査するなどの場合は，クエリとして，注目するフィールドとメソッドを指定することが望まれる．探索範囲の条件として，現在開発者がエディタ上で閲覧しているクラス群に限ったり，関連するパッケージ階層に限るなどの機能も有効であると考えらる．

このように既存機能に加えて，ユーザビリティの向上，ナビゲーション機能の充実が求められる．

5.5 妥当性の脅威

課題の一般性 本実験では，データフロー調査が主要な作業となる課題を設定した．より多彩な作業を必要とする一般的な作業，例えばプログラムの機能追加や修正作業でツールの有効性を確認するべきである．

被験者の力量 被験者 12 人はすべて学生だった．企業の開発者など，熟練の開発者に対してもツールの支援機能が十分に有効かどうか確認するべきである．

制限時間の設定 事前調査から，課題完了の目安を 30 分弱と判断した．ツールがない状態で

のスコアの平均値が 0.72 であることから妥当であると考えられるが，同等の課題に対して，制限時間を長くした状態，あるいは短くした状態でも実験結果が変わらないか確認するべきである．

6　関連研究

コードナビゲーションツールとして，データフロー以外にも，様々な観点で，開発者のソースコード上の移動を支援する研究が行われている．

考察でも述べたが，クエリの充実や開発者の関心に応じて探索範囲を限定させる手法が有効だと考えている．クエリベースのナビゲーションツールとしては，Eclipse 上で開発者が自然言語ベースで調査内容を入力できる手法や [29]，予め開発者が質問するであろう項目を列挙し，多数の解析ツールを駆使してそれらの質問に答える手法 [5] が提案されている．これらの手法により，開発者は現在閲覧しているコード片から，注目すべき別のコード片の情報を得ることができ，探索コストが削減できると考えられる．また，Kersten らは，Degree Of Interest（DOI）というメトリクスを用いて，開発者が関心を持っているクラスやメソッドを Eclipse 上で強調表示させる手法を提案している [15]．DOI では本ツールにも応用可能であり，表示するノードのフィルタリング機能として用いることができる．

また，これまでプログラム理解支援を目的としてプログラムの可視化手法が数多く提案されてきた [9]．クラス階層や呼び出し関係など，プログラムの構造や関係性に注目した可視化や行数，複雑度，変更履歴などの各種メトリクスに基づいてプログラムの特徴を可視化する手法が代表的である．こららの手法はプログラム理解の初期における，アーキテクチャの理解や全体の特徴を知るためには有効である．しかし，解析に時間がかかったり，実際にソースコード上を閲覧している時に知りたい情報を瞬時に引き出せないなど，詳細なプログラム理解を行う時にこのような可視化手法は相性が悪かった．本研究では，プログラムの詳細理解時にグラフを利用できるよう，Eclipse plugin として実装し，可視化したグラフとエディタとの連動性を高めた．

また，動的解析を用いてプログラムの動作を可視化したり，ソースコードに実行時の型情報を付加する研究 [21] も行われている．動的解析にかかるコストは大きいが，実行時の型情報を利用することができれば，グラフのフィルタリング機能に利用することができると考えている．

7 おわりに

オブジェクト指向プログラミングでは，開発者が注目するコード片を理解するためには，関連する他のコード片へ移動，読解する必要がある．特にデータフローに関する調査など，プログラムの詳細を理解する作業では，移動が多発する．関連するコード片はプログラム中に遍在しており，これまで開発者の移動を支援するために多数のコードナビゲーションツールが提案されてきたが，データフロー調査に特化したツールは提案されていなかった．そこで，本研究では，軽量なデータフロー解析技術を用いて，開発者が注目する識別子に対するデータフローを可視化するコードナビゲーションツールを提案，実装した．

適用実験では，12 名の学生に対して，プログラム理解作業を課題としたツールの有無による対照実験を行い，ツールの有効性を示した．実験の様子から，以下の点がデータフロー調査において有効に働いたと考察した．

- 推移的なデータフローを瞬時に確認できること．
- データフローパスが分岐した時に，グラフを起点として網羅的な探索が行えたこと．
- グラフを見直すことで，過去に調査したデータフローを確認しやすかったこと．

本手法では，解析にはプログラムスライシングの近似手法を用いており，制御フローを考慮せずにグラフを構築している．そのため，実際には起こりえない推移的なデータフローがグラフに表示される可能性がある．今後の課題として，グラフに表示されたデータフローが推移的に起こりえない可能性がある場合，開発者にソースコードでの確認を促す機能を追加すること，ユーザビリティ，ナビゲーション機能のさらなる充実が挙げられる．

ツールの実用性を向上させるには，ソースコードの解析にかかる時間はなるべく少ない方がよい．本ツールでは，解析ツールとして外部ツールを用いているが，Eclipse が内部情報として保持しているソースコードの AST 情報を利用できれば，解析コストの削減に繋がると考えられる．

謝辞

本研究の全過程を通じて，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いたします．

本研究の全課程を通じて，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 准教授に心より深く感謝いたします．

本論文を作成するにあたり，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 石尾 隆 助教に心より深く感謝いたします．

評価実験の実施にあたり，被験者として快く協力してくださっ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 学生の皆様に心より深く感謝いたします．

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] P. Anderson, T. Reps, T. Teitelbaum, and M. Zarins. Tool support for fine-grained software inspection. *IEEE Softw.*, 20:42–50, July 2003.

[2] Batik. Batik - java svg toolkit(`http://xmlgraphics.apache.org/batik/`).

[3] J. Bohnet and J. Döllner. Visual exploration of function call graphs for feature location in complex software systems. In *Proceedings of the 2006 ACM symposium on Software visualization*, pp. 95–104, 2006.

[4] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pp. 455–464, 2010.

[5] B. de Alwis and G. C. Murphy. Answering conceptual queries with ferret. In *Proceedings of the 30th international conference on Software engineering*, pp. 21–30, 2008.

[6] M. Desmond, M.-A. Storey, and C. Exton. Fluid source code views. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pp. 260–263, 2006.

[7] Eclipse. Eclipse(`http://www.eclipse.org/`).

[8] Graphviz. Graphviz(`http://www.graphviz.org/`).

[9] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 108–124, 1997.

[10] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th international conference on Software engineering*, pp. 392–411, 1992.

[11] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 39:229–243, April 2004.

[12] T. Ishio, S. Kusumoto, and K. Inoue. Debugging support for aspect-oriented program based on program slicing and call graph. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pp. 178–187, 2004.

[13] jEdit. jedit programmer's text editor(`http://www.jedit.org/`).

[14] 柳, 石尾, 井上. 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. 情報処理学会研究報告 第 167 回ソフトウェア工学研究発表会, 第 29 巻, 2010.

[15] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pp. 159–168, 2005.

[16] H. Koike. Fractal views: a fractal-based method for controlling information display. *ACM Trans. Inf. Syst.*, 13:305–323, July 1995.

[17] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Softw. Engg.*, 7:49–76, March 2002.

[18] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. In *Proceedings of the the 6th joint meeting of the European software engineering confeRence and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 361–370, 2007.

[19] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pp. 185–194, 2010.

[20] M. Pinzger, K. Graefenhain, P. Knab, and H. C. Gall. A tool for visual understanding of source code dependencies. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pp. 254–259, 2008.

[21] D. Rothlisberger, M. Harry, A. Villazon, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Augmenting static source views in ides with dynamic metrics. In *Proceedings of the 2009 IEEE International Conference on Software Maintenance*, pp. 253 –262, 2009.

[22] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pp. 112–122, 2007.

[23] J. Starke, C. Luce, and J. Sillito. Searching and skimming: An exploratory study. In *Proceedings of the 2009 IEEE International Conference on Software Maintenance*, pp. 157 –166, 2009.

[24] M.-A. Storey. Theories, methods and tools in program comprehension: Past, present and future. In *Proceedings of the 13th International Workshop on Program Comprehension*, pp. 181–191, 2005.

[25] W3C. Scalable vector graphics(`http://www.w3.org/Graphics/SVG/`).

[26] 三宅, 肥後, 井上. メトリクス計測プラグインプラットフォーム masu の開発. ソフトウェアエンジニアリング最前線 2008, pp. 63–70, 2008.

[27] N. Uehara, T. Kobayasi, T. Osuka, S. Ymamoto, and K. Agusa. Software comprehension support by multi-grained visualization. *Computer Software*, 27:2_112–2_117, 2010.

[28] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Trans. Softw. Eng.*, 18:1038–1044, December 1992.

[29] M. Würsch, G. Ghezzi, G. Reif, and H. C. Gall. Supporting developers with natural language queries. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pp. 165–174, 2010.

付録

適用実験の課題内容と正解について説明する．

課題の説明　課題 a，b 共にビープ音が鳴る原因を調査するものである．5.1 節で述べたとおり，被験者は beep() メソッドが呼ばれる直前の条件から探索を開始していくことになる．被験者には，最終的に原因と考えた箇所に加えて，調査の途中で参照したメソッド，フィールドについても解答用紙に記述してもらった．

データフローの探索をどこまで続けるかについては，ダイアログのボタンを押すなどのユーザの操作，ファイルやバッファに関する条件まで到達すれば，そこを原因とみなし，探索を打ち切るよう指示した．

課題 a　EditAbbrevDialog クラスの内部クラス ActionHandler，actionPerformed(ActionEvent) メソッド，153 行目の beep() メソッドが呼ばれる原因を調査する．課題 a の解答を表 3 に示す．探索深度とは，起点から探索条件を見つけるまでにかかるソースコード上での移動回数である．末尾に*が付いている条件は，データフロー探索の末端を示している．末尾に (不適) が付いている条件は，ビープ音がなる条件としては不適切な条件で，スコアには加算されない．不適切な条件とは，その条件を満たしたまま，beep() メソッドが呼ばれることはないことが，ソースコードの探索のみで分かる場合に該当する．これらの項目について調査が行われたかどうかを判定し，5.2 節で述べた評価基準に基づいて，採点を行った．

課題 b　JEditBuffer クラス，undo() メソッド，2043 行目の beep() メソッドが呼ばれる原因を調査する．課題 b の解答を課題 a の解答と同様の記述方法で表 4 に示す．

表 3: 課題 a の解答

探索深度	ビープ音が鳴る条件
1	editor.getAbbrev の戻り値が null か空文字
2	abbrev.getText の戻り値が null，空文字
3	abbrev.setText の引数が null，空文字
4	AddAbbrevDialog のコンストラクタで生成された値を保持 (不適)
4	EditAbbrev.init の引数が null，空文字
5	EditAbbrevDialog のコンストラクタの引数が null，空文字
6	AbbrevsOptionPane の actionPerformed が呼ばれる
7	AbbrevsOptionPane で add ボタンが押される *
6	AbbrevsOptionPane の内部クラス edit の abbrevsMode.getValueAt の戻り値が null，空文字 (不適)

表 4: 課題 b の解答

探索深度	ビープ音が鳴る条件
1	isEditable の戻り値が false
2	isReadOnly の戻り値が true
3	setReadOnly の引数が true
4	VFS.load で getCapabilities が WRITE_CAP *
3	setFileReadOnly の引数が true
4	Buffer.setPath で vfs.getCapabilities が VFS.WRITE_CAP *
4	Buffer.checkFileStatus で書き込み不可能な時 *
4	Buffer.checkFileForLoad で書き込み不可能な時 *
2	isPerformingIO がの戻り値が true
4	setLoading の引数が true
5	Buffer.load で setLoading(true) の状態が保持
6	Runnable.run が別スレッドで実行される *
3	setPerformingIO の引数が true
4	Buffer.save で setLoading の引数が true
5	Runnable.run が別スレッドで実行される *