Vol.2011-SE-171 No.12
2011/3/14

変数間データフローグラフを用いたソースコード間の移動支援

悦田翔悟[†1]　石尾隆[†1]　井上克郎[†1]

プログラムの動作を理解するには，制御およびデータの流れを複数の手続き単位に渡って追跡していく作業が必要である．本研究では，軽量なデータフロー解析技術を用いて，開発者が現在注目しているソースコード片に関係したソースコードの情報を提示する手法を提案する．具体的には，開発者がエディタ上で選択した識別子を起点としてデータフローグラフを探索し，グラフの要約を可視化するとともに，開発者が選んだグラフの頂点に対応するソースコードを表示可能とすることで，複数の手続きを横断したデータフローの調査を支援する．提案手法を Eclipse プラグインとして実装し，12 名の学生を対象に実験を行った結果，提案手法によって，同一時間で多くの手続きを調査可能となることを確認した．

Source Code Navigation Support Using Inter-Variable Data-Flow Graph

Shogo Etsuda,[†1] Takashi Ishio[†1] and Katsuro Inoue[†1]

To understand the behavior of a program, developers must investigate control-flow and data-flow among procedures. In this research, we propose a source code navigation approach using a lightweight data-flow analysis. Our approach uses a data-flow graph viewer visualizing data-flow paths surrounding the selected identifier when a developer selects an identifier on a Java editor. The viewer also enables the developer to open a source file corresponding to a vertex in the graph. We have conducted an experiment with 12 students. The result shows students working with Eclipse extended with our approach could investigate data-flow paths more efficiently than students working with a regular Eclipse.

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

1. はじめに

プログラムの動作を理解するには，制御およびデータの流れを複数の手続きに渡って追跡していく作業が必要である．ある変数の値をどの文が代入するか，また，その文はどのような条件で実行されるか，というように，推移的な追跡を行っていくことになる[5]．

開発者にとって，制御フローやデータフローの探索は，日常的に実施される業務であり[9]，時間を要する作業でもある[13]．これに対して，ソースコード上での開発者の「移動」を支援するために，様々な手法が提案されている． ここでの移動とは，現在注目しているソースコード片に関係しているソースコードの情報を探索し，エディタやその他ツールによって，それらのコードを表示する操作のことである．本研究と同様にデータフローに基づいてソースコードを探索する手法としてはプログラムスライシング[3] があるが，解析には制御フロー解析およびデータフロー解析が必要であり，計算に多くの時間とメモリを必要とすることから，日常的な開発には使用されていないのが現状である．

本研究では，Java プログラムを対象として，変数間データフローグラフ[11] を用いて，開発者が現在注目しているソースコード片に関係したソースコードの情報を提示する手法を提案する．本研究で使用する変数間データフローグラフ[11] は，我々の研究グループで考案した，プログラム依存グラフ[4] に対する近似計算用のグラフであり，プログラム実行時には起こりえないデータフローを抽出する可能性がある一方で，大規模なプログラムに対しても高速に構築できるという特徴を持つ．具体的な手法の手順としては，まず，事前にプログラムに対する変数間データフローグラフを構築しておき，開発者がソースコードエディタ上で選択した識別子を起点としてデータフローグラフの探索を行う．探索の結果得られたグラフに対して要約処理を適用したものを可視化することで，開発者はデータフロー関係をグラフ上で辿り，詳細を調べたいと選択したグラフ上の頂点から，対応するソースコードの表示を行う．これにより，複数の手続きを横断したデータフローの効果的な調査を可能とする．

提案手法の有効性を確認するために，本手法を Eclipse プラグインとして実装し，12 名の学生を対象に，適用実験を行った．この実験では，JEdit というテキストエディタにおいて，ある機能が実行できないと判定される条件を調査するというタスクを 2 つ実行した．この実験の結果，提案手法を用いることで同一時間内に多くの手続きを調査可能であり，その差が統計的に有意であることを示した．

本論文の構成は次のとおりである．2 章では，本研究の背景にある，ソースコードの閲覧に関する問題と既存研究を紹介する．3 章では提案手法の詳細について説明する． 4 章では

実装したツールを用いて行った適用実験の結果を示し，5 章でまとめと今後の課題を述べる．

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，移動が作業時間の多くを占めると言われている[13]．また，開発者にとっては，このような調査は頻繁に実施されるものである[9]．

移動が必要になる要因として，Java プログラムを対象に考えると，次の 3 つが挙げられる．

- **継承関係**．あるクラスには自身の親クラス，子クラスが存在しており，そのクラス内部に登場するメソッドやフィールドの参照は，親クラスから提供されている場合がある．
- **呼び出し関係**．メソッドは他のメソッドを呼び出すことができるため，あるメソッドの動作を理解するために，呼び出し関係にあるメソッドの内部を見なくてはならない場合がある．
- **データフロー**．フィールドに格納された値や，メソッドの引数として外部から渡された値がどのように計算されたかを知るには，それぞれ，フィールドへの代入を行ったメソッド，呼び出し元のメソッドを閲覧しなくてはならない．

このような状況に対して，数多くのプログラム理解支援ツールが提案されている．Storey らによると，プログラム理解支援ツールの機能は，情報を収集する（Extraction），解析する（Analysis），可視化する（Presentation）という 3 つに分類される[14]．このうち，情報の収集とは，構文解析などの基本的な解析技術のことである．これ以降では，解析および可視化の 2 つについて，既存手法と，本研究の特徴を示す．

2.1 解 析 技 術

開発者がプログラムの中を適切に「移動」するには，開発者が注目しているソースコードに対して，そこに関連したソースコードを探索する手法が必要である．

たとえば Eclipse は，開発者が注目している，すなわち，テキスト編集用のカーソルがある位置にある識別子に対して，以下の情報を表示する機能を提供している．

- クラス名に対して，そのクラスの持つ継承関係．
- メソッド名に対して，そのメソッドの宣言位置．
- メソッド名に対して，そのメソッドを参照しているメソッド．
- 変数名に対して，その変数を宣言している行．
- 変数名に対して，変数を参照している行．
- 任意の識別子に対して，その同一の識別子が出現している行．

メソッド名についての情報は，メソッド名から実際にその呼び出し先を見つける，あるいは逆に呼び出し側を見つける，いわゆる制御フローに関する情報を提供しているといえる．一方で，データフローについては，メソッドの呼び出し関係によって生じるもの以外を追跡する機能はサポートされていない．

データフローを扱う手法としてはプログラムスライシングを用いる方法が提案されており[4]，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[7),12)]．しかし，プログラムスライシングは，解析コストが高く，大規模プログラムに対しては適用が困難である．

本研究では，筆者らの研究グループで提案している変数間データフローグラフ[11] を用いる．変数間データフローグラフはプログラムスライシングの近似計算に当たり，制御フロー解析を省略することで，効率よくデータフロー解析を実行することができる．たとえば，本研究の実験で使用している JEdit は，10 万行を超えるプログラムであるが，これに対する変数間データフローグラフは，構文解析などを含めても 2 分程度で抽出することができる．従来，このグラフはメソッド間のデータフロー関係の探索にのみ用いており，プログラム理解に応用したのは本研究が初めてである．

2.2 可視化手法

可視化手法には，次のようなものがある．

2.2.0.1 Fluid Source Code Views[2)]

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，呼び出し部から定義部へ実行の流れを負担なく読めるように，メソッドの呼び出し部で，定義部の実装を折りたたみ表示する．これにより，メソッド呼び出し部で必要に応じて，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．

2.2.0.2 Code Bubbles[1)]

メソッドの動作を理解したい時，推移的にメソッド呼び出し先のコード片を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．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．

これらの手法は，開発者が調査を行っていく過程で閲覧したソースコードの関係を可視化

するための手法である．本研究によって得られるデータフロー解析手法では，グラフをそのまま可視化し，グラフから頂点を選ぶことで適切なソースコードをエディタに表示する方式を選択したが，このときの移動を，これらの手法で可視化することは可能である．

3. 提 案 手 法

本研究では，Java プログラムを対象として，開発者が注目している識別子 1 つを入力として，その識別子に関連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，次にデータフロー情報の提示方法について述べる．

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算出することができる，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[11)]．

変数間データフローグラフは，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

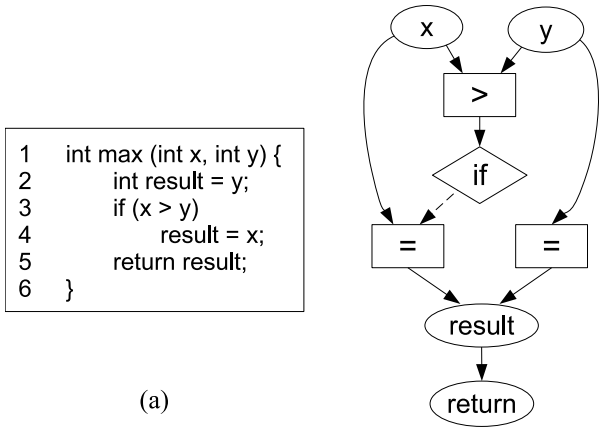

図 1 IVDFG の例，(a) メソッド max のプログラム，(b) (a) の IVDFG
Fig. 1 Example of IVDFG, (a) program of method max, (b) IVDFG of (a)

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．

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．

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

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．

3.2 データフローの提示

開発者がエディタ上で選択した識別子に対して，変数間データフローグラフの探索を行い，識別子周辺のデータフロー情報を可視化する．提案手法は，グラフの探索，グラフの可視化，そしてグラフの閲覧から構成される．

3.2.1 グラフの探索

本手法の入力は，開発者が，エディタ上で注目している識別子 1 つである．識別子には，クラス名，メソッド名や変数名などがあるが，本研究ではメソッド名と変数名のみを想定している．

このステップでは，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，可視化すべき頂点を見つける．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．なお，プログラムスライシングでは Two-Phase Slicing という辺の辿り方が一般的であるが，変数間データフローグラフは実行順序を無視した定義となっているため，単純な探索となっている．

変数が選択された場合は，その変数に関して，単純に Forward および Backward 探索を実施する．メソッドが選択された場合は，メソッドの引数，戻り値に関するデータフローを探索する．なお，開発者が現在注目しているコード片の中に含まれているデータフローは，エディタ内で直接ソースコードを読解することが可能であることから，注目している識別子がメソッド定義のときは，仮引数や，メソッド内部で参照しているフィールドに対する Backward 探索と，戻り値およびメソッド内部で代入するフィールドからの Forward 探索を行うことで，現在表示中のメソッドの周辺の情報だけを集める．

3.2.2 グラフの可視化

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頂点，辺を抽出し，可視化するが，細粒度のグラフを可視化する際には，頂点，辺の数が多くなり，可読性を下げることが課題とされている[10]．そこで，本研究では，可視化する頂点の種類，総数に制限を加えることでグラフの可読性を高める．

本研究で表示する頂点は，フィールド，メソッド呼び出し，ローカル変数，仮引数，リテラル，制御頂点のみとする．他の頂点については，グラフから取り除くが，演算を表現する頂点については，演算子の情報を辺のラベルとして残す．

提示する頂点を一定数に保つために，フラクタルビュー[6] に基づいた，以下のルールを用いて探索を打ち切る．

(1) 探索の起点となる頂点の fractal value を 1 とし，親頂点とする．
(2) 親頂点とデータフローがある，かつ未探索の頂点を子頂点とみなし fractal value を計算する．
(3) 子頂点の fractal value が閾値以上なら，新たな親頂点とみなす．
(4) 親頂点が新しく検出さなくなるまで，2．3．を繰り返す．
(5) 親頂点として検出された頂点を可視化する．

fractal value の算出方法は，以下の式で表される．

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

$F_{v_{focus}}$ は起点となる頂点の fractal value，$F_{v_{child\_of\_x}}$ は親頂点 F_{v_x} の子頂点の fractal value，C は減衰定数，N_x は親頂点がもつ子頂点の数，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，グラフ全体の頂点数 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，各頂点の辺数と頂点の全体数を決めることで，閾値を決定することができる．たとえば，辺数の平均を 5 と仮定したとき，表示される頂点の数を 30 程度としたい場合は，閾値を 0.04 に設定すればよい．

3.2.3 グラフの閲覧

開発者は，可視化されたグラフを閲覧することで，注目した識別子の周辺のデータフロー情報を知ることが可能である．Pinzger らによる調査によると，ソフトウェアをグラフ構造を可視化する際，最初はグラフの情報量を少なめにして提示しておき，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[10]．そこで，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開発者が選択したとき，それらの頂点からのデータフローを新たに探索す

Vol.2011-SE-171 No.12
2011/3/14

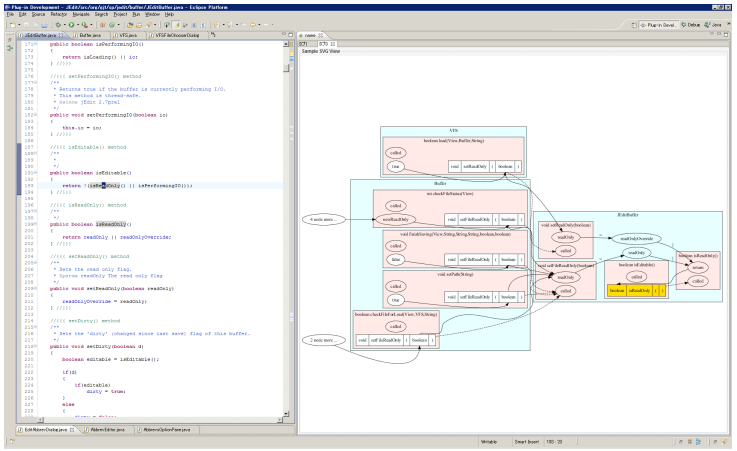

図 2 実装したツール：エディタ上で注目識別子を選択し，グラフを表示．
Fig. 2 A screenshot of the tool showing a data-flow graph for a selected identifier.

ることで，グラフの拡張を行うことを可能とした．また，グラフ上の注目する頂点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，頂点に対応するソースコードをエディタに表示することができる．

これらの機能により，開発者はエディタとグラフを連動させて，データフローの調査を行うことが可能となる．グラフ上でのデータフローの調査は，以下の点から，エディタ上で行う調査よりも移動のコストが削減されると考えられる．

- 代入の推移的な関係を一つのグラフから把握することができる．
- データフローに関連するメソッド呼び出しが複数存在する場合でも，グラフ上で漏れ無く把握することができる．

本手法では，エディタとグラフの連携が重要となることから，提案手法を Eclipse プラグインとして実装し，Eclipse の Java エディタと連携した動作を実現した．ツールのスクリーンショットを**図 2** に示す．図 2 の右側で開かれているビューが実装したグラフビューである．開発者は，エディタとグラフビューを使って，複数のコード片を横断して効果的にデータフロー調査を行うことができる．

4. 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，実装したツールによって開発者の移動が支援されるか実験を行った．

4.1 実 験 課 題

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 を対象とし，ビープ音が鳴る原因を調査してもらった．jEdit では，ユーザが指定した操作の実行に失敗すると，それをユーザに知らせるために，beep() メソッドを呼び出してビープ音を鳴らす．課題は，beep() メソッドが呼ばれる原因，すなわち直前の条件文が true になる原因を調査するというものである．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，調査の課程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，原因箇所とは，ユーザ操作による外部入力や，ファイル，バッファの状態に関わる条件とした．また，課題によっては原因が複数存在する場合もある．

課題 a は，EditAbbrevDialog クラスの 153 行目にある beep() メソッド呼び出しで，内部クラス ActionHandler の actionPerformed(ActionEvent) メソッドに含まれている．また，課題 b は，JEditBuffer クラスの 2043 行目の beep() メソッド呼び出しで，undo() メソッドに含まれている．

それぞれの課題を，被験者としてソフトウェア工学を専攻とする大学院生 9 名，大学生 3 名の計 12 名に解答してもらった．調査環境は，24 インチ，解像度 1920 × 1200 ピクセルのディスプレイ，調査は Eclipse 3.6 上で行い，他に解答用紙とボールペンを与えた．ツールの有効性を比較するために，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，ツール有りの場合，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．

実験では，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 30 分ずつ解答してもらった．

4.2 評 価 基 準

本実験は，採点方式で評価を行う．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．そして，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パスが出現する場合は，すべての正解パスを解答できれば満点とする．解答の数が足りない，正解パスの途中までしか解答出

Vol.2011-SE-171 No.12
2011/3/14

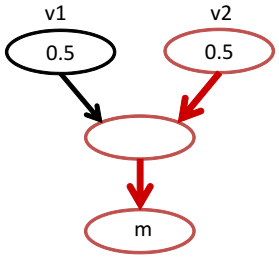

図 3 データフローパスの例.
Fig. 3 Example of data-flow paths.

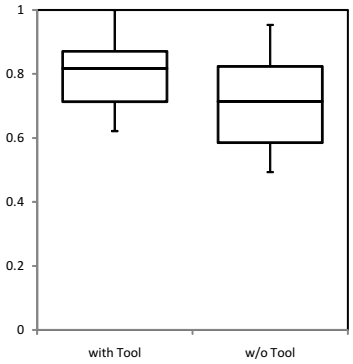

図 4 スコアの分布
Fig. 4 Distribution of score

来ていない場合は，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．

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

$weight(v)$ は，頂点 v の fractal value[6] である．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，$|path(x, y)|$ は頂点 x から y までのパスに含まれる辺の数である．

図 3 にデータフロー調査の例を示す．この例は，頂点 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 までが存在する．赤い辺が探索されたデータフローパスを，黒い辺が探索されなかったデータフローパスを表している．この場合，満点を 1 とすると正解が 2 つ存在するので，それぞれのパスを末端まで辿ることができれば，1 つのパスにつき 0.5 ずつスコアが与えられる．この例では，v1 から m までに関しては，途中までしか正解パス索ていないので部分点として $0.5 \times 1/2 = 0.25$ が与えられる．よってトータルスコアは $0.5 + 0.25 = 0.75$ である．

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．採点方式の場合，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．以上により，実験では採点方式を用いて評価を行った．

4.3 結　　果

学生 12 人に作業を行ってもらった．各被験者に対する課題の割り当て，スコアを**表 1** に示す．被験者のスコア分布を**図 4** に示す．図 4 中の with Tool はツールを使用した場合，w/o Tool はツールを使用しなかった場合を表す．表 1，図 4 に示すように，ツールを使用した方がスコアの平均値が高くなった．

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，ツールを使用した方がスコアが高い傾向にあることが分かった．

表 1 各被験者のスコア
Table 1 Score for each participant

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

4.4 考　　察

4.4.1 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，ツールを使っていない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ことが多かった．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，探索の優先順位を決定する際に本ツールは有効性を発揮した．

また，調査の途中でデータフリーパスが分岐していた場合，分岐した箇所のグラフを探索の起点とすることで，被験者は網羅的にすべてのパスを調査することが出来ていた．あるパスの調査を終了した後，被験者は一旦，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出来ていたが，グラフがない場合は最初の 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ことが多かった．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に手間がかかる時に，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，注目しているコード片が変わると，先程まで調査してい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．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必要があるが，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．

4.4.2 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，仮定を立てながら調査を行うと言われており[13),15)]，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[8)]．

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，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いものとなった．

4.4.3 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，エディタでソースコード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．

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頂点の間にも辺が引かれる可能性がある．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．ツールを使った被験者で 8 名中 7 名は，該当するコード片をソースコードをエディタで閲覧して，制御フローを確認していたために，調査に影響はなかった．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．

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辺では，代入文の行数を表示するなどの対策も考えられる．

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つけるなど，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などが望まれる．探索範囲を，現在開発者がエディタ上で閲覧しているクラス群に限定したり，関連するパッケージ階層に限定するなどの手法も有効であると考えらる．

4.5 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．ツールを使用しない場合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．

5. おわりに

本研究では，軽量なデータフロー解析手法を用いて，開発者が注目する識別子に関するデータフローを可視化するツールを提案，実装した．本ツールでは，開発者がエディタ上で識別子を選択するとデータフローグラフが表示され，データフローグラフからは追加のデータフローの探索や，対応するソースコードの表示を行うことが可能である．

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パスの探索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．

今後の課題として，グラフ上で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．

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) Bragdon, A., Reiss, S.P., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F. and LaViola, Jr., J.J.: Code bubbles: rethinking the user interface paradigm of integrated development environments, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pp.455–464 (2010).
2) Desmond, M., Storey, M.-A. and Exton, C.: Fluid Source Code Views, *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pp. 260–263 (2006).
3) Horwitz, S. and Reps, T.: The use of program dependence graphs in software engineering, *Proceedings of the 14th international conference on Software engineering*, pp.392–411 (1992).
4) Horwitz, S., Reps, T. and Binkley, D.: Interprocedural slicing using dependence graphs, *SIGPLAN Notice*, Vol.39, pp.229–243 (2004).
5) Ko, A.J. and Myers, B.A.: Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior, *Proceedings of the 30th ACM/IEEE International Conference on Software Engineering*, pp.301–310 (2008).
6) Koike, H.: Fractal views: a fractal-based method for controlling information display, *ACM Trans. Inf. Syst.*, Vol.13, pp.305–323 (1995).
7) Kusumoto, S., Nishimatsu, A., Nishie, K. and Inoue, K.: Experimental Evaluation of Program Slicing for Fault Localization, *Empirical Softw. Engg.*, Vol.7, pp.49–76 (2002).
8) LaToza, T.D., Garlan, D., Herbsleb, J.D. and Myers, B.A.: Program comprehension as fact finding, *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).
9) LaToza, T.D. and Myers, B.A.: Developers ask reachability questions, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pp.185–194 (2010).
10) Pinzger, M., Graefenhain, K., Knab, P. and Gall, H.C.: A Tool for Visual Understanding of Source Code Dependencies, *Proceedings of the 16th IEEE International Conference on Program Comprehension*, pp.254–259 (2008).
11) 柳　慶吾，石尾　隆，井上克郎：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， Vol.2010-SE-167, No.29, pp.1–8 (2010).
12) Sridharan, M., Fink, S.J. and Bodik, R.: Thin slicing, *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pp.112–122 (2007).
13) Starke, J., Luce, C. and Sillito, J.: Searching and skimming: An exploratory study, *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pp. 157 –166 (2009).
14) Storey, M.-A.: Theories, Methods and Tools in Program Comprehension: Past, Present and Future, *Proceedings of the 13th International Workshop on Program Comprehension*, pp.181–191 (2005).
15) Wilde, N. and Huitt, R.: Maintenance Support for Object-Oriented Programs, *IEEE Trans. Softw. Eng.*, Vol.18, pp.1038–1044 (1992).