Vol.2020-SE-206 No.14
2020/11/27

機能依存の理解におけるデルタ抽出の有効性について

神出 稔正[1,a)]　新田 直也[1,b)]

概要：既存のソフトウェアに新たな機能を追加するには，そのソフトウェアの内部で機能間の依存関係がどのようにして実現されているかを理解する必要がある．しかしながら，既存の機能依存検出技術は主に機能依存の有無を判定することを目的としており，依存の仕組みの理解においてどこまで有効であるかは明らかではない．そこで，機能依存の理解の支援を目的にソースコードを自動抽出する手法を提案する．提案手法は動的解析手法のデルタ抽出を用いており，機能依存の有無を判定する既存手法よりも広範囲のソースコードを抽出することが可能である．また，被験者を用いた実験により，提案手法を用いて抽出したソースコードは，既存の手法を適用して抽出したソースコードよりも機能依存の理解度への寄与が大きいことを確認した．さらに，機能の依存を媒介する動的構造として準静的構造を定義し，デルタ抽出によって準静的構造が効率良く抽出できることを確認した．

1. はじめに

既存のソフトウェアに対する機能の追加は，対象となるソフトウェアの規模が大きくなればなるほど難しくなる．その理由の一つとして，新しく追加する機能には，既存の機能との連携が求められることが多い点が挙げられる．新しい機能と既存の機能を正しく連携させるためには，まず既存の機能がそのソフトウェア内部でどのような仕組みで依存しあっているのかについて理解する必要がある．

ソフトウェア実行時の機能間の依存関係を特定する手法として，機能依存検出技術 [3], [6] が研究されている．機能依存検出技術は，プログラムを実行したときに複数の機能間に依存関係が存在しているか否かを判定することを目的としており，主に回帰テストを実施する範囲を調べるために用いられる．しかしながら，依存の有無を判定するための機能依存検出技術が，依存の仕組みを理解する上でどこまで有効であるかについては明らかではない．たとえばオブジェクトフロー [3] では，ある機能の実行で参照されたオブジェクトが別の機能の実行によって参照されるまでに，どのようにして受け渡しされてきたかを追跡することができる．機能間の依存関係の有無を検出するだけなら，そのようなオブジェクトの流れを追跡するだけで十分であるが，一般にあるオブジェクトの受け渡しには他の複数のオブジェクトが関わるため，単独のオブジェクトの追跡だけでは依存の仕組みの理解には不十分である可能性が高い．

そこで本研究では，デルタ抽出 [5] を用いて機能依存の仕組みの理解に必要なソースコードを抽出することを考える．デルタ抽出は 2 つのオブジェクトがどのようにして関連付けられたかを示すソースコードを抽出する動的解析技術で，対象となるオブジェクトだけでなく関係するすべてのオブジェクトを追跡することができるという特長を持ち，一般にオブジェクトフローよりも広い範囲のソースコードを抽出することができる．本稿では被験者を用いて，オブジェクトフローによって抽出したソースコードと，デルタ抽出によって抽出したソースコードの違いが，機能依存の理解度に与える影響について調べる実験を行った．その結果，3 つのオープンソースソフトウェアの 4 つの機能依存のうちの 3 つにおいて，デルタ抽出で抽出したソースコードの方がオブジェクトフローで抽出したソースコードよりも理解度を向上させることがわかった．さらに，機能依存を媒介する動的構造として，準静的構造というオブジェクトグラフの部分構造の存在を仮定し，3 つのオープンソースソフトウェアにおいて実際に準静的構造が機能依存を媒介していること，およびデルタ抽出によって準静的構造を効果的に抽出できることを確認した．

2. 機能依存とオブジェクト

ソフトウェアの実行において，通常，ある機能の実行はそれ以前の別の機能の実行に依存する．例えば図形エディタを考えたとき，図形削除機能によってどの図形が削除されるかは，それ以前の図形選択機能の実行においてどの図形を選択したかによって決まる．このような機能間の

[1] 甲南大学大学院 自然科学研究科
Graduate School of Natural Science, Konan University
[a)] m2024001@s.konan-u.ac.jp
[b)] n-nitta@konan-u.ac.jp

実行時の依存関係を**機能依存**という．本研究では，ソフトウェアの内部で機能依存がどのようにして実現されているかを理解することについて考える．本節では例として，オープンソースの UML エディタである ArgoUML の，クラス図上におけるクラス選択機能と，クラス削除機能の間の機能依存について取り上げる．ArgoUML ではクラス図上に配置されたクラスをクリックすると，そのクラスが選択状態になる．このとき，プログラム内部では，図 1 に示した `ModeSelect` クラスの `mousePressed()` メソッド (以下，`ModeSelect#mousePressed()` のように書く) が呼び出される．`ModeSelect#mousePressed()` の中では，クリックしたクラスに相当する `FigClass` クラスのインスタンスが，160 行目の代入文で変数 `underMouse` によって参照され，187 行目の呼び出しによって選択状態になる．ここで，`underMouse` の型である `Fig` クラスは，`FigClass` の祖先クラスに相当する．この状態で Delete キーを押すと，選択状態にあるクラスが削除される．このとき，図 2 に示した `ActionRemoveFromDiagram#actionPerformed()` が呼び出され，選択状態にある `FigClass` クラスのインスタンスが，86 行目で変数 `f` によって参照され，89 行目の呼び出しによって削除される．

ここで，図 2 の削除機能のコードの実行が，図 1 の選択機能のコードの実行に依存するには，図 2 中の変数 `f` と図 1 中の変数 `underMouse` が `FigClass` クラスの同じインスタンスを参照している必要があることに注意して欲しい．ただし，ここに示したコードだけでは，これら 2 つの変数が常に同じインスタンスを参照するかどうかは不明である．例えば，図 1 の 156 行目で，フィールド `editor` で参照されている `Editor` クラスのインスタンスと，図 2 の 83 行目で，変数 `ce` で参照されている `Editor` クラスのインスタンスが同一である保証はない．仮にこれらが，同じ `Editor` クラスのインスタンスを参照していたとしても，図 2 の 156 行目を実行したときと，図 2 の 85 行目を実行したときで，`Editor#getSelectionManager()` の呼び出しが，`SelectionManager` クラスの同じインスタンスを返すかどうかはわからない．

```
53: public class FigModifyingModeImpl
                      extends ModeImpl
                      implements FigModifyingMode {
       :
61:    protected Editor editor;
       :
179: }
```

```
70: public class ModeSelect
                     extends FigModifyingModeImpl {
        :
116:    public void mousePressed(MouseEvent me) {
           :
156:       SelectionManager sm
                    = editor.getSelectionManager();
           :
160:       Fig underMouse = editor.hit(selectAnchor);
           :
187:       sm.select(underMouse);
           :
198:    }
       :
362: }
```

図 1 ArgoUML のクラス選択機能で実行されるコード (一部)

```
60: public class ActionRemoveFromDiagram
                       extends AbstractAction {
        :
82:    public void actionPerformed(ActionEvent ae) {
83:       Editor ce = Globals.curEditor();
          :
85:       List<Fig> figs
               = ce.getSelectionManager().getFigs();
86:       for (Fig f : figs) {
             :
89:         f.removeFromDiagram();
            :
94:       }
95:    }
96: }
```

図 2 ArgoUML のクラス削除機能で実行されるコード (一部)

3. デルタ抽出とオブジェクトフロー

前節で見たように，2 つの機能が依存するには，それらの機能が少なくとも 1 つ以上の同一オブジェクトにアクセスしている必要がある．**オブジェクトフロー** [3] は，注目したオブジェクトへの参照がプログラムの実行に伴ってどのように受け渡しされてきたかを追跡する動的解析技術であり，オブジェクトフローを適用することで，実行時の機能間の依存の有無を検出することが可能である．例えば，前節の機能依存の例では，図 2 の 89 行目の変数 `f` で参照されている `FigClass` クラスのインスタンスが，どのようにしてここに来たかを実行と逆向きに追跡することで，図 1 の 160 行目の変数 `underMouse` への代入文にたどり着くことができる．しかしながらオブジェクトフローでは，対象としたオブジェクトの追跡しか行われない．したがって，オブジェクトフローによるコードの抽出のみでは，なぜそのオブジェクトがここまでたどり着くことができたのかを理解するには不十分であると考えられる．例えば前節の例に関して，オブジェクトフローによって，`FigClass` クラスのインスタンスがどのようにして受け渡しされてきたかの経路はわかるが，それぞれの機能によって参照されている `Editor` クラスのインスタンスや，`SelectionManager` クラスのインスタンスなどの別のオブジェクトの関係がどうなっているのかはわからない．

いっぽうデルタ抽出 [5] は，2 つのオブジェクトが関連付けられた経緯を示すコードと，その関連付けに関わったすべてのオブジェクトおよびそれらが成すオブジェクトグラフを抽出する技術である．例えば前節の例では，図 2 の 89 行目で，ActionRemoveFromDiagram のインスタンス (this オブジェクト) と，FigClass クラスのインスタンスが，局所変数 f によって関連付けられるまでの経緯と，その関連付けに関わった Editor クラスのインスタンスや SelectionManager クラスのインスタンスが同時に抽出される．そのため，デルタ抽出によるコードの抽出が，なぜ 2 つの機能を依存させることができるのかという機能依存の仕組みの理解にとって，より効果的であると期待される．デルタ抽出と機能依存の具体的な関係については，4.2 節で詳しく述べる．

4. オブジェクトグラフの準静的構造

4.1 機能が依存するための条件

2 節の例において，クラス選択機能とクラス削除機能の間で，なぜ FigClass クラスの同一インスタンスを共有できるのかについて考えてみよう．2 節でも説明したように，図 1 中の変数 underMouse と図 2 中の変数 f によって FigClass クラスの同じインスタンスが参照されることが保証されるためには，図 1 の 156 行目と図 2 の 85 行目で SelectionManager クラスの同じインスタンスが，また図 1 の 156 行目と図 2 の 83 行目で Editor クラスの同じインスタンスがアクセスされることが保証されればよい．実際にクラス選択機能とクラス削除機能を実行してみると，これらのインスタンスが 2 つの機能間で常に一致していることを確認することができる．これら 2 つの機能の実行時にそれぞれアクセスされるオブジェクトの一部をオブジェクトグラフとしてまとめたものを図 3 に示す．この図において，楕円はオブジェクトを表し，矩形はクラスオブジェクトを表す．また，オブジェクト間の矢印はフィールドによる参照を表し，楕円もしくは矩形内部の小矩形はメソッドの実行を，メソッド実行間の破線矢印は呼び出し関係を表す．図から，2 つの機能間で同一の Editor オブジェクトが共有されていることがわかるが，これは図 1 の 156 行目で参照される Editor クラスのインスタンスと，図 2 の 83 行目で参照される Editor クラスのインスタンスが常に一致していることを示している．ここで，この Editor クラスのインスタンスが等しく，さらにそこから先の FigClass クラスのインスタンスに至るまでの構造が，クラス選択機能の実行からクラス削除機能の実行までの間変わらないため，これらの機能間で同一の FigClass クラスのインスタンスを共有できることがわかる．ただし，Editor クラスのインスタンスから FigClass クラスのインスタンスに至る構造のうち，ArrayList クラスのインスタンスから先の部分は，クラス選択機能の実行によって生成されたものであることに注意が必要である．この部分はクラス選択機能の実行後は，クラス削除機能の実行まで変わらない．このように，

(1) 依存先の機能の起点となるオブジェクトから共有対象となるオブジェクトに至るアクセスパスの一部が依存先機能の実行によって生成され，

(2) 依存元の機能の起点となるオブジェクトから共有対象となるオブジェクトに至るアクセスパスと途中で合流し，

(3) さらにそれらのパスが成す構造が依存先機能の実行直後から依存元機能の実行直前までの間不変に保たれる

ことが，それらの機能が依存する条件であると考えられる．

4.2 機能依存とデルタ抽出

図 3 で示した構造がデルタ抽出を用いてどのように抽出されるかについて説明する．まず，図中の ActionRemoveFromDiagram のインスタンスと FigClass クラスのインスタンスは，図 2 の 89 行目で局所変数 f によって関連付けられるため，関連付けの経緯を表すデルタを抽出すると，図 3 のオレンジ色の破線で囲まれた部分が抽出される．さらにこの抽出された部分を見ると，SelectionClass クラスのインスタンスと FigClass クラスのインスタンスが，フィールド contents によってクラス選択機能の実行時に関連付けられたことがわかるため，次にその関連付けの経緯を表すデルタを抽出すると，今度は図中の青色の破線で囲まれた部分とその左側の灰色の部分が抽出される．このようにしてデルタ抽出を繰り返すことによって，機能が依存するための条件となるオブジェクトグラフの部分構造を取り出すことができる．

4.3 準静的構造

一般に，プログラム実行中の新たなインスタンスの生成や，フィールドへの代入によるオブジェクト間参照の生成などによって，オブジェクトグラフの構造は常に変化し続ける．いっぽう，4.1 節で説明したように，実行時に機能が依存するためには，オブジェクトグラフの構造の一部が一定期間変化せず保たれ続ける必要がある．ここでは，ソフトウェアが起動している間中，変化しないオブジェクトグラフの構造を**準静的構造**と定義する．具体的には，準静的構造とは**準静的参照**からなるオブジェクトグラフの部分構造であり，準静的参照とは以下のすべての条件を満たすフィールド参照のことをいう．

- ソフトウェアの起動処理が終了して以降，ソフトウェアの終了処理が開始するまでの期間で，値が変化した場合は変化する直前の値が必ず null である，
- ソフトウェアの終了処理が開始する時点でメモリ中に

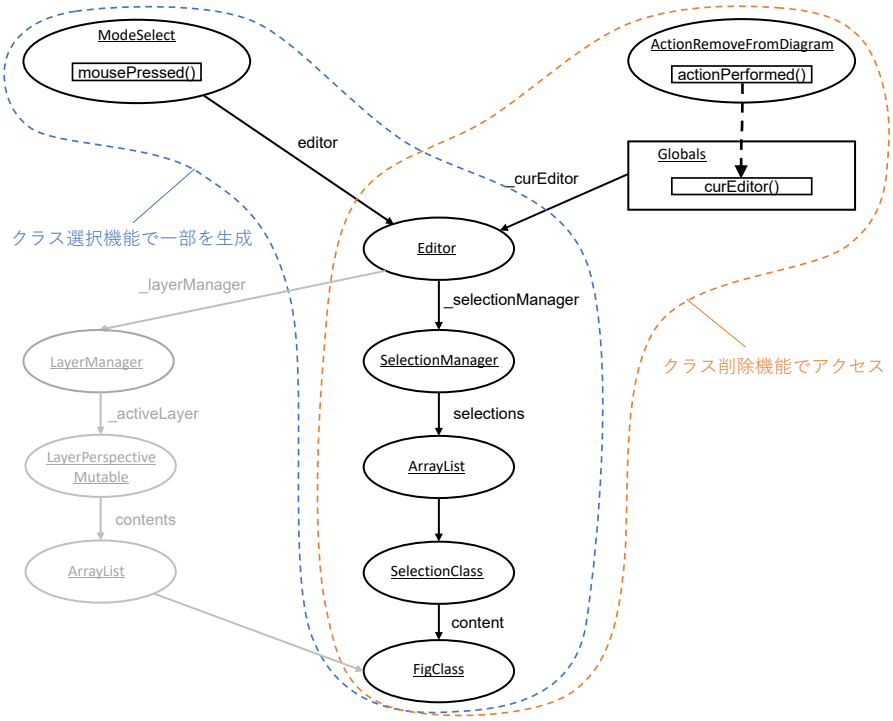

図 3 ArgoUML のクラス削除機能とクラス選択機能の依存関係

残されている．

定義の 1 つ目の条件は次の 2 つの条件と等価である．

- ソフトウェアの起動処理終了時には null 値を持ち，その後ソフトウェアの終了処理開始時までの期間に null 以外の同じ値でのみ更新される，
- ソフトウェアの起動処理が終了した時点で null 以外の値を持ち，その後ソフトウェアの終了処理開始時までの期間で一度も別の値で更新されない．

定義ではフィールドの宣言時に一度 null が代入された後に実際の参照先の値が代入される場合を考慮して，null での初期化を無視している．また，プログラムの実行中に何度も同じ値で更新される可能性があるが，これは参照先が不変であるため準静的参照とみなす．機能間の依存と関係がないことから，機能の実行が行われない起動処理および終了処理中の値の更新は準静的の判定対象外とする．また，プログラム実行中にガベージコレクションによってオブジェクトが消滅し，それに伴ってオブジェクト間参照が消滅する可能性がある．これらのオブジェクト間参照は，ソフトウェアの終了処理開始時点ではメモリ中に残されていないため準静的参照とはみなさない．

配列は配列の要素番号をフィールドとして考え，配列から要素への仮想的なオブジェクト間参照が存在すると考えることによって適用可能である．Java コレクション型についても同様である．また，要素番号を持たない集合である HashSet なども，仮想的なフィールドが存在するものとして扱うことにする．なお，この準静的構造の定義はソフトウェアが起動している間中の不変性を仮定しているため，機能の実行間の不変性のみを仮定する機能依存のための条件よりも強いものとなっている．

5. リサーチクエスチョン

前節までの議論で用いた仮説の検証のため，以下のリサーチクエスチョンを立てる．3 節で説明したように，オブジェクトフローでは注目したオブジェクトへの参照がどのようにして受け渡されてきたかを追跡するのに比べ，デルタ抽出では 2 つのオブジェクトがどのように関連付けられたのかと，その関連付けに関わったすべてのオブジェクトの関与を抽出することが可能である．そこで，RQ1 と RQ2 では機能依存に関してデルタ抽出とオブジェクトフローの比較を行う．

RQ1: オブジェクトフローとデルタ抽出で，機能依存の抽出範囲に差があるか？

RQ2: デルタ抽出によって抜き出したソースコードによる機能依存の仕組みの理解と，オブジェクトフローによって抜き出したソースコードによる機能依存の仕組みの理解に差があるか？

RQ1 では実際のオープンソースソフトウェアの機能依存を対象にデルタ抽出およびオブジェクトフローのそれぞれを適用することによって抽出範囲の比較を行う．抽出範囲に差が見られた場合，RQ2 ではこの抽出範囲の差が機能依存の仕組みの理解に影響を与えるか否かを被験者を用いた実験によって評価する．

また 4 節において機能が依存するための条件として，各機能の起点となるオブジェクトから共有対象となるオブ

ジェクトに至る 2 つのパスが途中で合流し，そのオブジェクトグラフの構造が一定期間不変である必要があるという仮説を立て，さらにより強い条件として準静的構造の存在を仮定した．以下のリサーチクエスチョンではその検証とデルタ抽出との関係の調査を行う．

RQ3: 全オブジェクト間参照中の，準静的参照が占める割合はどの程度か？

RQ4: 抽出したデルタに含まれるオブジェクト間参照中の準静的参照が占める割合はどの程度か？

RQ5: 抽出したデルタに含まれるオブジェクト間参照のうち，準静的ではないものはなぜか？

RQ6: デルタ抽出によって，機能の起点となるオブジェクトから機能間で共有されているオブジェクトまでの参照パスを抽出できているか？

オブジェクトグラフの構造はプログラムの実行に伴って変化すると考えられるため，起動している間中構造が変わらない準静的参照の割合はそれほど多くないと予想される．そのため RQ3 では，各対象のアプリケーションに対して全オブジェクト間参照数と準静的参照数を算出しその割合を求める．RQ4 では，デルタ抽出を用いて抽出したオブジェクト間参照が準静的であるか否かを確認する．デルタ抽出がオブジェクトグラフ中から均等にオブジェクト間参照を抽出しているのであれば，この割合は RQ3 と変わらないことが予想される．RQ5 では RQ4 のうち，準静的参照ではなかったものについて議論を行う．デルタ抽出で抽出した構造がたとえ準静的であったとしても，それだけでは機能が依存することはできない．そこで，RQ6 では，機能の起点となるオブジェクトから機能間で共有されているオブジェクトまでの参照パスを抽出できているかについて確認を行う．

6. 評価実験

デルタ抽出を用いて抽出したソースコードの，機能依存の理解における有効性を評価するために有効性評価実験を行う．また，機能依存と準静的構造の関係を調べるために，準静的構造に関する解析を行う．対象アプリケーションとして，UML エディタの ArgoUML ver. 0.34[*1]，テキストエディタの jEdit ver. 4.3[*2]，2 次元グラフィックフレームワークの JHotDraw ver. 7.6[*3] を用いる．これらのアプリケーションはいずれも Java で記述されたオープンソースソフトウェアである．これらの対象ソフトウェアから複数の機能を選択し，それらの機能間の依存関係に対し，デルタ抽出およびオブジェクトフローを適用してソースコードの抽出を行った．本実験で対象とした機能依存を表 1 に示す．

[*1] https://github.com/argouml-tigris-org/argouml/
[*2] http://www.jedit.org/
[*3] https://sourceforge.net/projects/jhotdraw/

表 1 対象とした機能依存

	アプリケーション	依存先機能	依存元機能
D1	ArgoUML	クラス選択機能	クラス削除機能
D2	ArgoUML	クラス配置機能	クラス選択機能
D3	jEdit	文字選択機能	文字削除機能
D4	JHotDraw	図形選択機能	図形移動機能
D5	JHotDraw	矩形配置機能	図形選択機能

表 2 抽出したソースコードの行数の比較

機能依存	D1	D2	D3	D4	D5
デルタ	155	120	70	131	137
オブジェクトフロー	92	78	55	75	91

表 3 抽出したソースコードのフィールド数の比較

機能依存	D1	D2	D3	D4	D5
デルタ	8	11	5	13	15
オブジェクトフロー	3	5	3	8	10

表 4 抽出したソースコードのメソッド数の比較

機能依存	D1	D2	D3	D4	D5
デルタ	17	15	8	17	18
オブジェクトフロー	14	9	6	9	11

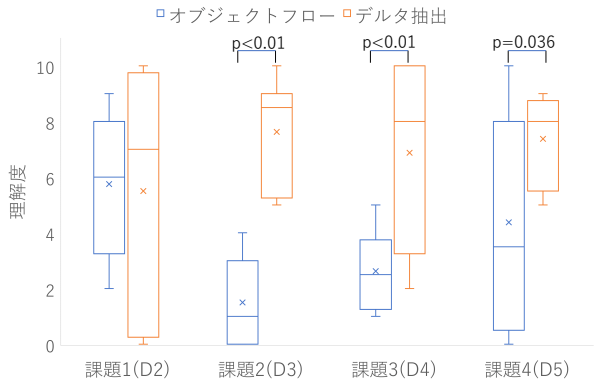

図 4 有効性評価実験の結果

6.1 有効性評価実験

RQ1 および RQ2 の調査のために有効性評価実験を行った．本実験では，まず RQ1 を確認するために，対象ソフトウェアの機能依存それぞれにオブジェクトフローおよびデルタ抽出に基づいてソースコードの抽出を行った．それぞれの機能依存において抽出したソースコードの行数を表 2 に，フィールド数を表 3 に，メソッド数を表 4 に示す．3 節で説明した通り，オブジェクトフローとデルタ抽出では，ソースコードの抽出範囲に違いがみられた．全ての機能依存で，デルタを用いた方が抽出されたコードの行数が多く，より広範囲のソースコードが抽出されたことがわかる．また，デルタ抽出では追跡対象のオブジェクトが切り替わることによって，より多くのメソッドおよびフィールド参照を抽出することもわかる．

次に RQ2 の調査のために，被験者を用いた実験を行った．実験は，オブジェクトフローまたはデルタ抽出で抽出

表 5　全オブジェクト間参照数と準静的参照数

	ArgoUML	jEdit	JHotDraw
オブジェクト間参照数	10658	11489	7613
準静的参照数	6839	6517	2717
割合	64.17%	56.72%	35.69%

したソースコードのいずれかを被験者に見てもらい，それによる機能依存の理解度を 0 から 10 の 11 段階で回答してもらった．具体的には，以下のようなアンケートを行った．たとえば，2 節で説明した ArgoUML のクラス選択機能とクラス削除機能の依存 (D1) の場合，選択中の `FigClass` クラスのインスタンスを参照する変数 `underMouse` と，削除対象となる `FigClass` クラスのインスタンスを参照する変数 `f` がある．これに対し，被験者に「なぜ `underMouse` と同じ値が `f` に必ず代入されるのか？」について，抽出されたソースコードから理解できるかを回答してもらった．なお，被験者には web 上でソースコードを提示し，メソッドの呼び出し先や戻り先等の実行時の情報をリンクとして付与した．これにより，被験者はプログラムの実行の流れに即してソースコードを読むことが可能となる．

実験全体の流れについて説明する．被験者にはまず実験に慣れてもらうことを目的に，練習課題としてデルタ抽出に基づいて抽出した ArgoUML の機能依存 (D1) を見てもらい，その後，機能依存の理解度をアンケートに回答してもらう．この練習課題は実験の結果に含めない．練習課題終了後は，同様の流れでそれぞれの抽出手法の利用が 2 例づつとなるように，D2~D5 の計 4 つの課題で繰り返す．すべての課題終了後は実験全体に関するアンケートに回答してもらう．

被験者は Java の実務経験が 3 年以上という条件でクラウドソーシングサービスで募集を行った．集まった被験者 20 名のうち，各課題の後のアンケートで 1 度でも『わからない Java の構文が有った』や『課題の意味を理解できなかった』を選択した被験者，および，全ての課題において同じ理解度を選択 (全ての課題で 10 を選択など) しその理由について言及していない被験者の計 4 名を最終的な集計から除外した．実験は順序効果を考慮して 8 名ずつの 2 グループに分け，グループ毎に抽出手法を用いる順序を変えて行った．被験者には抽出手法の数や，どの抽出手法を用いたのかがわからないようにした．なお，課題はいずれも D2~D5 の順序で行った．実験の結果を図 4 に示す．図中の×印は平均値を示している．各課題において，非等分散を仮定した片側 t 検定を行った結果，課題 2 と課題 3 では 2 つの手法間に有意水準 0.01，課題 4 では有意水準 0.05 の有意差が認められた．いっぽうで，課題 1 では有意差が認められなかった．

6.2　準静的構造解析

RQ3~RQ6 の調査のために対象アプリケーションに対して準静的構造の解析を行った．本解析では，動的解析プラットフォーム [8] を用いて収集した実行トレースの情報をもとに行った．実行トレースの収集ために各アプリケーションで標準シナリオを設定した．たとえば ArgoUML の場合，以下の手順で操作を行った．

(1) アプリケーションを起動する．

(2) ツールバーのクラス配置モードを選択し，クラスを配置する (クラス配置機能)．

(3) (2) で配置したクラス図形の選択状態を一度外す．

(4) ツールバーの選択モードを選んだ後，再度 (2) のクラス図形をクリックし選択する (クラス選択機能)．

(5) Delete キーを押し，(4) で選択した図形を削除する (クラス削除機能)．

(6) アプリケーションのウィンドウの閉じるボタンをクリックし，アプリケーションを終了する．

jEdit，JHotDraw に関しても同様に実行シナリオを設定し，実行トレースを収集した．以後の解析ではこのようにして収集された実行トレースを用いることにより，解析プログラムの実行を繰り返しても，オブジェクト ID など同一の条件にて解析を行うことが可能となる．

RQ3 の調査のために，実行トレース中に記録されたオブジェクト間の全参照数と，準静的参照数を算出した．対象となる準静的参照は 4 節の定義に基づき，ソフトウェアの起動処理終了後からソフトウェアの終了処理開始時点までの期間を準静的参照判定の対象とした．ただし解析の簡単化のため，Java 標準のコレクション型へ追加されたオブジェクトへの参照は，その後の処理に関わらず準静的参照とした．また，ソフトウェアの終了処理の開始時点のオブジェクトグラフの状態を再現するためにエスケープ解析を行った．結果を表 5 に示す．各アプリケーションによって異なるものの，全オブジェクト間参照中の準静的参照が占める割合は 35%~65%程度であった．

次に RQ4~RQ6 の調査のために，抽出したデルタに含まれるオブジェクト間参照が準静的参照であるか否かを調べた．本解析では，3 アプリケーション 5 課題で抽出した機能依存に含まれる全オブジェクト間参照を対象とした．なお，抽出されたオブジェクト間参照のうち，複数の機能で抽出されたものは 1 つとして数えた．その結果を表 6 に示す．全体として抽出したデルタに含まれるオブジェクト間参照の 93.10%が準静的参照であった．いっぽうで，以下の 2 つのオブジェクト間参照は準静的ではなかった．

- ArgoUML の `LayerManager.activeLayer`,

表 6　抽出したデルタに含まれる準静的参照数

	ArgoUML	jEdit	JHotDraw	計
オブジェクト間参照数	10	4	15	29
準静的参照数	9	4	14	27
割合	90.00%	100.00%	93.33%	93.10%

- JHotDraw の `DefaultDragTracker.transformedFigures`,

これらのオブジェクト間参照が準静的でない理由 (RQ5) については，次の節で議論する．

RQ6 については機能特有の処理が始まるオブジェクトを起点とし，機能間で共有されているオブジェクトまでの参照パスを抽出できているかについて，抽出したソースコードを目視で確認した．その結果，デルタ抽出に基づいて抽出したすべての機能依存で，参照パスが抽出できていることを確認した．

7. 考察

7.1 リサーチクエスチョンに対する回答

3 節で予想した通り，デルタ抽出とオブジェクトフローでは，抽出範囲に差があることが確認できた (RQ1)．これは，オブジェクトフローではある単一のオブジェクトを追跡対象とするのに対し，デルタ抽出では 2 つのオブジェクトの関連付けに関わったすべてのオブジェクトを追跡対象とすることに起因する．

オブジェクトフローとデルタ抽出の抽出範囲の差が，機能依存の仕組みの理解に及ぼす影響を確認するために実験を行った．その結果，4 つのうち 3 つの機能依存において，2 つの手法の抽出範囲の差が機能依存の仕組みの理解に影響を与えていることを確認できた (RQ2)．デルタ抽出を用いた手法では，2 つのオブジェクトがどのようにして関連付けられたのかと，その関連付けに関わったすべてのオブジェクトを抽出することが可能である．そのため，依存先の機能で保持されたオブジェクトが，依存元の機能でどのようにして取得されたかを知ることができ，機能依存の仕組みの理解につながったと考えられる．いっぽうで，課題 1 で有意差が認められなかったのは被験者に提供したアプリケーション実行時の情報が不足していたことが理由だと考えられる．被験者には実行時の情報として，メソッドの呼び出し先や戻り先などの情報は提供したが，抽出したソースコード中の変数の値については提供をしなかった．この点は課題 1 終了後のアンケートにおいて，デルタ抽出に基づいて抽出したソースコードを見た複数の被験者から指摘されている．変数の値は通常のデバッガでも確認できるものであるため，抽出したソースコードをデバッガ上で表示することによってこの問題は解決すると考えられる．

準静的構造解析の結果，全オブジェクト間参照中の準静的参照が占める割合は ArgoUML で 64.17%, jEdit で 56.72%, JHotDraw で 35.69%となった (RQ3)．また，デルタ抽出に基づいて抽出したオブジェクト間参照は 93.10%が準静的参照であった (RQ4)．一般にオブジェクトグラフの動的構造はプログラムの実行に伴って絶えず変化するため，準静的参照が占める割合はそれほど高くないと予想していた．しかし，ArgoUML と Edit では準静的参照が占める割合が 50%を超えており，実行中に生成されたオブジェクト間参照の半数以上が，プログラムの終了処理開始時点まで準静的参照として残り続けていることが判明した．また全オブジェクト間参照中の準静的参照が占める割合が最も低かった JHotDraw でも 35%を超えていた．いっぽうで，デルタ抽出に基づいて抽出したオブジェクト間参照では全体の 93%が準静的参照であり，ほぼすべてのオブジェクト間参照が準静的参照であった．デルタ抽出が，オブジェクトグラフ中からオブジェクト間参照を均等に抽出しているのであれば，この割合は RQ3 の割合と変わらないはずであるが，RQ4 の割合は RQ3 の割合より明らかに高く，デルタ抽出は準静的参照を選択的に抽出しているといえる．RQ5 では RQ4 のうち準静的参照ではなかったオブジェクト間参照について確認した．JHotDraw では図形移動機能で使用されるフィールド `DefaultDragTracker.transformedFigures` が準静的参照ではなかった．このフィールドを確認したところ，メソッドの呼び出しの度に生成されたインスタンスが代入されていることが判明した．また，ArgoUML では `LayerManager.activeLayer` が準静的参照ではなかった．ArgoUML では，図形を複数のレイヤに配置することができる．この `LayerManager.activeLayer` は現在アクティブなレイヤを参照するため，アクティブなレイヤを切り替える度にこの変数の値も変化することから，明らかに準静的ではない．しかしながら，アクティブなレイヤが変わらない限りはクラス選択機能で選択したクラスが，クラス削除機能で削除される．RQ6 では機能依存の条件である，機能の起点となるオブジェクトから共有対象となるオブジェクトに至る参照パスが抽出されているかを確認した．その結果デルタ抽出に基づいて抽出したすべての機能依存でこの参照パスが抽出できていた．いっぽうで，オブジェクトフローでは単一のオブジェクトしか追跡されないため，この参照パスの全体を抽出することができない．そのためデルタ抽出は機能依存の仕組みの理解に有効であると考えられる．

準静的構造は，機能依存が成立するための条件である，機能の実行間で構造が変化しないというものよりも強い定

義となっている．しかしながら，機能依存で用いられるオブジェクト間参照のほとんどすべてが準静的参照であった．これは，ソフトウェアの設計者が機能を依存させる際に，この準静的構造を意図して使っているためではないかと考えられる．また，RQ5 で確認したように，準静的ではない参照についてはその理由を説明することが可能である．今後の課題として，準静的構造とソフトウェアアーキテクチャの動的側面との関係について解明していく予定である．

7.2 妥当性の検証

有効性評価実験では，実験への慣れが抽出手法の評価に影響を与えないよう順序効果を考慮して行った．まずは実験結果に含まれない練習課題 (D1) を実施し，不慣れな初回の課題が結果に大きく影響を与える可能性を低減させた．被験者には使用した抽出手法がいくつあり，各課題においてどの抽出手法を用いたのかがわからないようにして実験を行った．また，Java の経験年数が長い被験者や短い被験者が，それぞれの課題において特定の抽出手法に偏ることがないように考慮した．これらのことから，内的妥当性が満たされていると考えられる．さらに，被験者は Java の実務経験が 3 年以上を条件に，クラウドソーシングサービスで募集を行った．これは，学内の学生に協力してもらう方法や，特定の企業に派遣を依頼する方法と比べて，標本の偏りが少ないといえ，外的妥当性が高いと考えられる．ただし，課題数，対象アプリケーション数についてはより多い方が望ましいと考えられる．

8. 関連研究

機能依存に関する研究はあまり多くはない．Salah ら [6] は，ある機能の実行で生成されたオブジェクトが後の別の機能の実行で利用されることを検出して，機能間の実行時の依存関係を特定する手法を提案した．Lienhard ら [3] は，ある機能の実行から別の機能の実行までのオブジェクトの参照の流れを追跡することによって，より多くの機能依存を特定することを可能にした．本研究は機能依存の特定に加えて，理解に必要な情報を抽出することを目的としている．そのため，デルタ抽出 [5] を用いて，機能依存に関わるより広い範囲のソースコードが抽出できるようにした．ソースコードの抽出手法としては，動的スライス [1], [7] が良く知られているが，抽出範囲が広くなるため，機能依存の理解には適さない．

機能追加に関して，アプリケーションフレームワークの拡張方法に関する研究が行われている．Heydarnoori ら [2] は，サンプルアプリケーションの複数の実行トレースを解析して，アプリケーションフレームワークの拡張例としてテンプレートを生成する手法を提案した．新田ら [4] は，非対称スライスを用いて，サンプルアプリケーションの単独の実行トレースからテンプレートを生成可能な手法を提案した．ただし，機能追加を行うために必要な既存の機能間の依存関係の理解に関する研究は今のところ見当たらない．

9. おわりに

機能依存の理解を目的として，理解に必要なソースコードをデルタ抽出に基づいて抜き出す手法の構築を行い，被験者を用いて評価実験を行った．その結果，機能依存を理解する上では，本手法で抜き出したコードの方がオブジェクトフローで抜き出したコードよりも有効であることがわかった．また，オブジェクトグラフの部分構造として準静的構造を定義し，機能依存を理解するための枠組みとして準静的構造が有効であることを示した．今後の課題として，準静的構造の定義をより一般化するとともに，ソフトウェアアーキテクチャの動的側面の理解との関係について解明していく予定である．

謝辞 研究の初期段階で準静的構造に関する調査を行ってくれた卒業生の藤大之氏に感謝いたします．また，実験に協力してくださった被験者の皆様に感謝いたします．この研究の一部は，私立大学等経常費補助金 特別補助「大学間連携等による共同研究」による．

参考文献

[1] Agrawal, H. and Horgan, J. R.: Dynamic program slicing, *Proc. the Conference on Programming Language Design and Implementation*, pp. 246–256 (1990).
[2] Heydarnoori, A., Czarnecki, K. and Bartolomei, T.: Supporting framework use via automatically extracted concept-implementation templates, *Proc. 23rd European Conference on Object-Oriented Programming (ECOOP'09)*, pp. 344–368 (2009).
[3] Lienhard, A., Greevy, O. and Nierstrasz, O.: Tracking Objects to Detect Feature Dependencies, *Proc. 15th International Conference on Program Comprehension*, pp. 59–68 (2007).
[4] Nitta, N., Kume, I. and Takemura, Y.: Identifying mandatory code for framework use via a single application trace, *Proc. 28th European Conference on Object-Oriented Programming (ECOOP'14)*, pp. 593–617 (2014).
[5] Nitta, N. and Matsuoka, T.: Delta extraction: An abstraction technique to comprehend why two objects could be related, *Proc. the 31st International Conference on Software Maintenance and Evolution*, pp. 61–70 (2015).
[6] Salah, M and Mancoridis, M: A hierarchy of dynamic software views: from object-interactions to feature interacions, *Proc. IEEE International Conference on Software Maintenance (ICSM 2004)*, pp. 72–81 (2004).
[7] Tallam, S., Tian, C. and Gupta, R.: Dynamic slicing of multithreaded programs for race detection, *Proc. 24th IEEE International Conference on Software Maintenance*, pp. 97–106 (2008).
[8] 石谷涼，新田直也：オンラインおよびオフライン動的解析プラットフォームの開発とそのオブジェクトフロー解析への応用，情報処理学会研究報告，2019-SE-202(12) (2019).