Vol.2016-SE-194 No.2
2016/11/17

オブジェクト指向プログラムの性質に基づく対話的機能抽出手法の提案

藤岡 大樹[1,a)]　新田 直也[1,b)]

概要：本研究では, オブジェクト指向プログラムの性質を系統的に利用することによって, 対象機能に関連するソースコードを対話的に抽出する手法の構築を目指す. 具体的には, ソフトウェア偵察を用いて対象機能に関連するソースコードの一部分を抽出した後に, オブジェクト指向プログラムの性質を用いて関連するソースコード全体を網羅するように対話的に抽出範囲を拡大するアプローチをとる. 本稿では, jEdit のバージョン 4.3 で追加された機能のうち規模の大きいもの 3 つを対象に手作業で本手法を適用したところ, 関連するソースコードの 75%以上を網羅するよう抽出範囲を拡大することができたので報告する.

1. はじめに

大規模ソフトウェアの保守・再利用作業では, 対象となる機能が膨大なソースコード上のどのコンポーネントで実装されているかを調べることに多くの時間が費される. そこで, ソースコードの中から対象となる機能の実装箇所を効率よく特定するための**機能抽出技術**が広く研究されている. 代表的な機能抽出技術として, **ソフトウェア偵察** [1] を挙げることができる. ソフトウェア偵察は, 対象となる機能をユーザが実行したときの実行トレースと実行しなかったときの実行トレースを比較して, 実行したときのトレースにのみ含まれているメソッドをその機能の実装箇所として抽出する動的解析手法である. ソフトウェア偵察を用いることによって対象機能の実装箇所の一部分を精度良く取り出すことができる. しかしながら, 実行トレースを取得する際にユーザが行った操作方法やソフトウェアの内部状態に抽出精度が大きく依存する点, 実装箇所の広い範囲を網羅的に抽出できない点などが問題として挙げられる. 例えば, ある機能をユーザが実行する前に, プログラムの内部でその機能を初期化するためのコードが予め実行されている場合が少なくないが, そのような初期化コードはユーザが対象機能を実行した場合もしなかった場合も同様に実行されてしまうため, 原理的にソフトウェア偵察によって抽出することができない.

実装箇所のより広い範囲を抽出するための手法には, 多数の実行トレースを用いるアプローチ [2][3], 静的解析技術と組み合わせるアプローチ [4], 情報検索技術と組み合わせるアプローチ [5][6], それらを複合したアプローチ [7] などがある. 本研究では対象機能に関係しないコードの抽出を極力排除するという観点と, 抽出されたコードがなぜその機能に関係しているのかを説明できるようにするという観点から, 情報検索技術を使用せずに主に動的解析を用いて抽出範囲の拡大を試みる. 具体的には, ソフトウェア偵察によって得られた抽出範囲を, オブジェクト指向プログラムが持つ動的および静的性質を用いながら対話的に拡大していくことによって機能抽出を行う. なお, 本研究ではオブジェクト指向言語として Java を取り上げる. ユーザが最適な選択を行うという仮定の下で, 提案手法によってどこまで正解集合に近い解を得ることができるかを, jEdit のバージョン 4.3 で追加された機能のうち規模の大きいもの 3 つを対象に調査した. その結果, 少数の実行トレースを用いるだけで正解集合の 75%以上を網羅できることがわかった. 今後, これらの結果に基づいて手法の詳細な設計を行うと共に, より広い範囲を網羅できるよう手法を拡張していく予定である.

2. 機能抽出

一般にソフトウェアの機能はソースコード上の複数のコンポーネントに跨って実装される. このようなソフトウェアの性質を散乱という. ソフトウェアの規模が大きくなればなるほど各機能の実装はより広い範囲のコンポーネントに散乱される傾向にあり, そのような場合各機能の実装箇所を網羅的に特定することは非常に困難となる. そこで, 対

[1] 甲南大学大学院 自然科学研究科
Graduate School of Natural Science, Konan University
[a)] m1624002@s.konan-u.ac.jp
[b)] n-nitta@konan-u.ac.jp

象となる機能を実装しているコンポーネントを効率よく特定する**機能抽出技術**が広く研究されている．代表的な機能抽出技術として，**ソフトウェア偵察** [1] を挙げることができる．ソフトウェア偵察は，対象となる機能をユーザが実行したときのプログラム内部の情報を**実行トレース**として記録し，同じ機能を実行しなかったときの実行トレースと比較して実装箇所を特定する動的解析手法である．具体的には，これら 2 つの実行トレースのうち対象機能を実行したときの実行トレースのみで実行が確認されたメソッドを，その機能の実装箇所として抽出する．ソフトウェア偵察を用いることによって対象機能の実装箇所の一部分を精度良く取り出すことができるが，実行トレースを取得する際にユーザが行った操作方法やソフトウェアの内部状態に抽出精度が大きく依存する点，実装箇所の広い範囲を網羅的に抽出できない点などが問題として挙げられる．本研究では特に後者の問題に着目し，その改善を試みる．実装箇所のより広い範囲を抽出する方法として，利用する実行トレースの数を増やすアプローチ [2][3]，静的解析技術と組み合わせるアプローチ [4]，情報検索技術と組み合わせるアプローチ [5][6]，それらを複合したアプローチ [7] などが提案されている．実行トレースの数を増やすアプローチは多くの場合有効ではあるが，例えば対象機能を初期化するコードがプログラム起動時に必ず実行されるような場合，そのようなコードの抽出にはまったく効果がない．また，情報検索技術を用いることによって抽出範囲を広げることは可能であるが，逆に対象機能に関係しないコードまで抽出してしまう可能性がある．また情報検索技術では，なぜ抽出されたコードがその機能に関係しているのかを説明することができない．そこで本研究では，使用する実行トレースの数を最小限に留めつつ主に動的解析を用いて抽出範囲を広げることを考える．なお，ソースコード上のどの部分を対象機能の実装箇所と定義するかについてはいくつかの方法がある．例えば文献 [8] では “対象とする機能をプログラムから刈り取るときに削除するか修正しなければならないソースコードをその機能の実装箇所とする” という**刈り取り依存規則**が提案されているが，本研究ではそれとほぼ同等の “対象機能がそのプログラムに最後に追加されたと仮定したときに追加されるか修正されるソースコードをその機能の実装箇所とする” という**最新追加規則**を提案する．

3. 本研究のアプローチ

本研究では，2 つ以上の実行トレースに対してソフトウェア偵察を行った結果得られた抽出範囲を，オブジェクト指向プログラムが持つ動的および静的性質を用いながら対話的に拡大していくアプローチを採る．同時に，通常メソッドの単位で行うソフトウェア偵察をブロック単位でも行えるよう拡張する．以下，具体例を用いて本研究のアプローチを説明する．例として，jEdit のバージョン 4.3 で追加されたある機能を取り上げる．jEdit は Java で記述されたオープンソースのテキストエディタであり，着目するのはその中のアクティビティログの検索機能である (図 1 参照)．アクティビティログの検索機能はアクティビティログ画面中の文字列入力欄にキーワードを入力することによって実行される．そこで，文字列入力欄にキーワードを入力した場合としなかった場合の実行トレースを取り，それらに対してソフトウェア偵察を行う．そうすると，4 つのメソッドを抽出することができる．図 2 の `LogViewer` クラスの `setFilter()` メソッド (以下，`LogViewer#setFilter()` のように書く) がそのうちの 1 つである．すべてのメソッドは対象となる機能の追加において，新規に作成されたか，一部を変更されたか，全く変更されていないかのいずれかになるが，本研究では前節で紹介した最新追加規則を仮定しているため，この点に着目して抽出範囲を求める．まず，ソフトウェア偵察によって抽出されたメソッドはその機能の実行においてしか呼び出されないものであるため，その機能の追加において新規作成されたメソッドであると考える．`LogViewer#setFilter()` が新規作成されたメソッドであるならば，そのメソッドの呼び出し行である図 2 の 79 行目も新たに追加されたものであると考えられる．したがって，その行を含む無名クラスの `insertUpdate()` メソッド (以下，`LogViewer$1#insertUpdate()` のように書く) は一部を変更されたか新規に作成されたかのいずれかであることがわかる．ここで，この `LogViewer$1#insertUpdate()` の内容に着目すると，このメソッド本体にはこの呼び出し行しか含まれていないため，`LogViewer$1#insertUpdate()` 全体が新規作成されたものとみなしてもよいといえる．しかしながら，このような判断をどこまで一般化できるかが不明であるため，本アプローチでは一般的な断定が困難な判断についてはユーザに委ねることとする．この実行トレースにおける `LogViewer$1#insertUpdate()` の呼び出し元は `AbstractDocument#fireInsertUpdate()` で

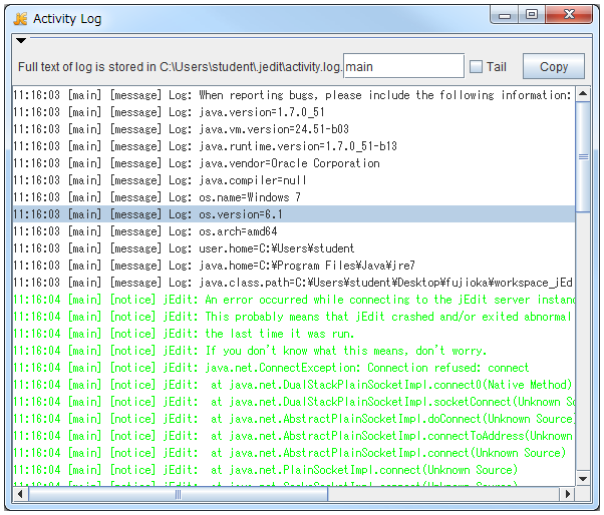

図 1 jEdit のアクティビティログ検索機能

```
          :
 39:   public class LogViewer extends JPanel implements DefaultFocusComponent,
 40:   EBComponent
 41:   {
          :
 43:      public LogViewer()
 44:      {
             :
 70:         filter.getDocument().addDocumentListener(new DocumentListener()
 71:         {
                :
 77:            public void insertUpdate(DocumentEvent e)
 78:            {
 79:               setFilter();
 80:            }
                :
 86:         });
           :
113:      }
          :
173:      private void setFilter()
174:      {
175:         listModel.setFilter(filter.getText());
176:         scrollLaterIfRequired();
177:      }
          :
386:   }
```

図 2 jEdit の LogViewer クラスのソースコード (抜粋)

```
         :
  43:   public abstract class FilteredListModel<E extends ListModel> extends AbstractListModel implements ListDataListener
  44:   {
           :
  50:      private Vector<Integer> filteredIndices;
           :
 220:      public int getTrueRow(int rowIndex)
 221:      {
 222:         if (filteredIndices == null)
 223:            return rowIndex;
 224:         return filteredIndices.get(rowIndex).intValue();
 225:      }
           :
 278:   }
```

図 3 jEdit の FilteredListModel クラスのソースコード (抜粋)

あるが, `AbstractDocument` は標準クラスであるためその内容を変更することができない. したがって, すでに `LogViewer$1#insertUpdate()` が新規作成であると判断されているため, この場合変更なしのメソッドから新規作成メソッドが呼び出されたことになる. このような呼び出しは, オブジェクト指向プログラムにおける動的束縛の仕組みによって実現される. 具体的には, 呼び出し元の `AbstractDocument` のインスタンス (実際にはその子クラスである `PlainDocument` クラスのインスタンス) が, 新規に作成された無名クラス `LogViewer$1` のインスタンスを実行時に参照しており, その参照を用いて `LogViewer$1#insertUpdate()` を呼び出したということになる. このとき, 標準クラスである `AbstractDocument` 自身が `LogViewer$1` のインスタンスを生成して参照したという可能性はない. なぜなら, 新規作成クラスのインスタンスを生成できるのは新規作成メソッドか変更メソッドのみであるからである. したがって `LogViewer$1` のインスタンスは, どこか別のクラスのメソッドで生成され, この `AbstractDocument` のインスタンスまで渡されてきたということがわかる. また, そのようなメソッドが新規作成メソッドか変更メソッドのいずれかであることもわかる. `LogViewer$1` のインスタンスが渡されてきた経路を実行と逆向きに辿ることによってそのようなメソッドを見つけ出すことができる. 実際には `LogViewer$1` のインスタンス

は, 図 2 の 70 行目の `Document#addDocumentListener()` の呼び出しによって渡されてきたことが図 2 のプログラムより容易に推測できるが, 一般には動的スライスなどの技術を用いて特定することになる. 70 行目の実行によって新規クラスのインスタンスが生成され渡されてきたことから, この行も新たに追加されたものであることがわかる. したがって, この行を含む `LogViewer#LogViewer()` も一部を変更されたか新規に作成されたかのいずれかであることがわかる.

ブロック単位のソフトウェア偵察は, 以下のような状況に対応するために導入する. 図 3 の `FilteredListModel` クラスの `getTrueRow()` メソッドは, 検索機能の実行の有無に関わらず, アクティビティログ画面を表示している状態で常に呼び出されるが, このメソッド中の 224 行目は検索機能を実行したときにのみ実行される (ここでは, この行を仮想的に 1 つのブロックとみなす). このことから, 222 行目の条件分岐および 224 行目が検索機能の追加によって付け足されたか, もしくはこのメソッド全体が新規作成されたと推測することができる. しかしながら, メソッド全体は検索機能の実行の有無にかかわらず実行されるため通常のソフトウェア偵察ではこのメソッドを抽出することができない. そこで, ソフトウェア偵察において実行の有無を確認する単位をメソッドからブロックにまで詳細化することによって, このブロック (224 行目) とそれを含むメソッドを抽出できるようにする. ブロック単位のソフトウェア偵察を行うことによってこれ以外にも 2 つのメソッドを抽出することができ, またそれらはすべて正解集合の中に含まれている. `FilteredListModel#getTrueRow()` が新規に作成されたか一部を変更されたかの判断についてはユーザに委ねることになるが, 一部を変更されたと仮定すると変更前のメソッド本体は 223 行目だけという意味を成さない構成になるため, 新規作成されたものであるとみなすことが可能である.

以上の推定の中で, メソッドもしくはブロック単位のソフトウェア偵察の結果とユーザに判断を委ねた部分を除けば, すべてがオブジェクト指向プログラムの一般的な性質を用いていることに注意して欲しい. このようにソフトウェア偵察の結果から始めて, ユーザの判断とオブジェクト指向プログラムの性質を組み合わせて利用していくことによって, 次々と各メソッドを, 新規に作成されたか, 一部を変更されたか, 全く変更されていないか推定していくことができる. 最終的に, 新規作成もしくは変更ありと推定されたメソッドを対象機能の実装箇所として抽出することができる.

4. 提案手法

前節で説明したように, ソフトウェア偵察を用いて対象機能に関連するソースコードの一部分を抽出し, オブジェクト指向プログラムの性質を用いて関連するソースコード全体を網羅するように対話的に抽出範囲を拡大する手法を提案する. 本手法は入力として, 対象機能を含む複数の機能に関連する複数の実行トレースを受け取り, 対象機能を実装していると考えられるメソッドの集合を出力する. 本手法は, 受け取った複数の実行トレースをもとに, 4.3 節で示す命題および仮定を満たす解を極小解から順番に対話的に探索する. これらの命題と仮定は, 抽出範囲が満たすべき制約条件を表し, 抽出範囲がソフトウェア偵察による抽出結果を含むという制約とオブジェクト指向プログラムの静的および動的性質による制約によって構成される.

4.1 諸定義

以下では, 提案手法の定義に必要ないくつかの概念および記法を定義する. 対象ソフトウェアを S とおく. S を構成するすべての機能からなる集合を F とおく. 2 つの機能 $f, f' \in F$ について f がその振る舞いの一部として f' を含むとき $f \xrightarrow{\text{includes}} f'$ と書く. 例えば, 3 節で説明した jEdit のアクティビティログ画面の検索機能は, アクティビティログ画面の中で実行されるためアクティビティログの表示機能の中に含まれることになる. S のソースコードを構成するすべてのクラスおよびインターフェースからなる集合を C, それらを構成するすべてのメソッドからなる集合を M とおく. ここでクラスもしくはインターフェース $c \in C$ に定義されているメソッド全体からなる集合を $\mathrm{methodsOf}(c)$ と書く. C に含まれるクラスおよびインターフェースのうち, 機能 $f \in F$ に対する最新追加規則 (2 節参照) にしたがって新規作成されたものを $C_{\mathrm{new}}(f)$, 変更されたものを $C_{\mathrm{mod}}(f)$, 変更されていないものを $C_{\mathrm{unmod}}(f)$, 追加時に存在していないものを $C_{\mathrm{nonexist}}(f)$ とおく. ただし, f に対して $f \xrightarrow{\text{includes}} f'$ を満たす機能 f' が存在するとき, 最新追加規則を “最後に f および f' が追加されたと仮定して, そのうちの f のみを追加したときに追加されるソースコード” と解釈しなおす. このとき $C_{\mathrm{new}}(f)$, $C_{\mathrm{mod}}(f)$, $C_{\mathrm{unmod}}(f)$, $C_{\mathrm{nonexist}}(f)$ は互いに素で $C = C_{\mathrm{new}}(f) \cup C_{\mathrm{mod}}(f) \cup C_{\mathrm{unmod}}(f) \cup C_{\mathrm{nonexist}}(f)$ が成り立つ. また, $f \xrightarrow{\text{includes}} f'$ を満たす機能 f' が存在しないとき $C_{\mathrm{nonexist}}(f) = \emptyset$ である. 同様に M に含まれるメソッドに対しても $M_{\mathrm{new}}(f)$, $M_{\mathrm{mod}}(f)$, $M_{\mathrm{unmod}}(f)$, $M_{\mathrm{nonexist}}(f)$ を定義すると, $M_{\mathrm{new}}(f)$, $M_{\mathrm{mod}}(f)$, $M_{\mathrm{unmod}}(f)$, $M_{\mathrm{nonexist}}(f)$ は互いに素で $M = M_{\mathrm{new}}(f) \cup M_{\mathrm{mod}}(f) \cup M_{\mathrm{unmod}}(f) \cup M_{\mathrm{nonexist}}(f)$ が成り立つ. メソッド m の本体に含まれるブロック全体からなる集合を $\mathrm{blocksOf}(m)$ と書く.

4.2 実行トレース

1 つの機能に対して 1 つ以上の実行方法を考え, そのそれぞれに対して実際に機能を実行して取得した実行トレースと実行せずに取得した実行トレースを用意する. ここで

機能の 1 つの実行方法を実行パターンとよぶ. 用意する実行トレースは機能の実行パターンを n 個とした場合, $2n$ 個になる. これらを対象機能を含むいくつかの機能に対して用意する. 対象機能以外の機能として例えば対象機能を含む機能がある場合, そのような機能に対しても実行トレースを用意するのが望ましいと考えられるが, ユーザが抽出精度の向上に資すると考えるならば用意するのはそれだけにかぎらない. ある機能 f のある実行パターン p を考える. このとき p にしたがって実際に f を実行した場合の実行トレースを T_p^+, しなかった場合の実行トレースを T_p^- と書く.

4.3 抽出範囲が満たすべき制約条件

機能 $f \in F$ について, $M_{\mathrm{new}}(f)$ または $M_{\mathrm{mod}}(f)$ に属するメソッドが f に対する抽出範囲となる. 抽出範囲を拡大していく過程において, $M_{\mathrm{new}}(f)$, $M_{\mathrm{mod}}(f)$, $M_{\mathrm{unmod}}(f)$, $M_{\mathrm{nonexist}}(f)$ が常に満たすべき制約条件を命題および仮定の形式で表現する. これらの制約条件は, 抽出範囲がソフトウェア偵察による抽出結果を含むという制約と, オブジェクト指向プログラムの静的および動的性質による制約によって構成される. 以下では, 任意のオブジェクト指向プログラムについて成り立つ性質を命題 (ただし未証明), 多くのオブジェクト指向プログラムにおいて成り立つと予想される性質を仮定とする. ブロック単位のソフトウェア偵察を定義するためにいくつかの記法を導入する. M を構成するすべてのブロックからなる集合を B とおく. B に含まれるブロックのうち, ある機能 $f \in F$ のある実行パターン p を実行した実行トレース T_p^+ のみで実行されたものを $B_{\mathrm{spec}}(f)$, T_p^+ および T_p^- を含めた 2 つ以上の実行トレースで実行されたものを $B_{\mathrm{comm}}(f)$ とおく. このとき $B_{\mathrm{spec}}(f)$, $B_{\mathrm{comm}}(f)$ は互いに素である.

4.3.1 オブジェクト指向プログラムの静的性質

オブジェクト指向プログラムの静的性質によって課される抽出範囲への制約を以下に示す.

命題 1 任意の機能 $f, f' \in F$ について, $f \neq f'$ のとき, $M_{\mathrm{new}}(f') \cap M_{\mathrm{new}}(f) = \emptyset$, 特に $f \xrightarrow{\text{includes}} f'$ の場合, $M_{\mathrm{nonexist}}(f') \cap M_{\mathrm{new}}(f') \subseteq M_{\mathrm{nonexist}}(f)$ である.

命題 2 標準クラスまたは共通ライブラリ, フレームワーク内のクラス c は $c \in C_{\mathrm{unmod}}(f)$ である.

命題 3 任意の機能 $f \in F$ および f における新規クラス $c \in C_{\mathrm{new}}(f)$ について, c に属するすべてのメソッド $m \in \mathrm{methodsOf}(c)$ は $m \in M_{\mathrm{new}}(f)$ である.

命題 4 任意の機能 $f \in F$ および f における変更されていないクラス $c \in C_{\mathrm{unmod}}(f)$ について, c に属するすべてのメソッド $m \in \mathrm{methodsOf}(c)$ は $m \in M_{\mathrm{unmod}}(f)$ である.

仮定 5 任意の機能 $f \in F$ およびクラス $c \in C$ について, c に属するあるメソッド $m \in \mathrm{methodsOf}(c)$ が $m \in M_{\mathrm{new}}(f)$ かつ m がオーバーライドしているメソッドが抽象メソッドであるとき $c \in C_{\mathrm{new}}(f)$ である.

命題 6 メソッド $m \in M_{\mathrm{new}}(f)$ の内部に定義された無名クラス c は $c \in C_{\mathrm{new}}(f)$ である.

4.3.2 ソフトウェア偵察

抽出範囲がソフトウェア偵察による抽出結果を必ず含むことを表す制約条件を以下に示す. これらは本手法における仮定である. 仮定 7 はメソッド単位のソフトウェア偵察に, 仮定 8 はそのブロック単位への拡張に相当する.

仮定 7 任意の機能 $f \in F$ について, f のある実行パターン p において実行トレース T_p^+ で実行され, 実行トレース T_p^- で実行されなかったメソッド m は $m \in M_{\mathrm{new}}(f)$ である.

仮定 8 任意の機能 $f \in F$ について, ブロック $b \in B_{\mathrm{spec}}(f)$ とブロック $b' \in B_{\mathrm{comm}}(f)$ の両方を含むメソッド m は $m \in M_{\mathrm{new}}(f) \cup M_{\mathrm{mod}}(f)$ である.

4.3.3 オブジェクト指向プログラムの動的性質

オブジェクト指向プログラムの動的性質によって課される抽出範囲への制約を以下に示す.

命題 9 任意の機能 $f \in F$ のある実行パターン p における実行トレース T_p^+ において, メソッド $m \in M_{\mathrm{new}}(f)$ がメソッド $m' \in M_{\mathrm{unmod}}(f)$ に呼び出されている場合, m はコンストラクタではなく, かつ m はあるクラス c のメソッドをオーバーライドしており, かつ m' が所属するクラス c' は c もしくはその子孫クラスを静的に参照しているか, 外部クラスに持つ.

命題 10 任意の機能 $f \in F$, f の実行パターン p および f の新規クラス $c \in C_{\mathrm{new}}(f)$ について, T_p^+ 中であるメソッド $m \in \mathrm{methodsOf}(c)$ がメソッド $m' \in M_{\mathrm{unmod}}(f)$ に呼び出されている場合, T_p^+ 中のそれ以前の時点で $m'' \in M_{\mathrm{new}}(f) \cup M_{\mathrm{mod}}(f)$ が少なくとも 1 つ実行されている.

仮定 11 任意の機能 $f \in F$ および f の実行パターン p について, T_p^+ 中でメソッド $m \in M_{\mathrm{unmod}}(f)$ が $m' \in M_{\mathrm{new}}(f) \cup M_{\mathrm{mod}}(f)$ に呼び出されている場合, m の呼び出し先に $m'' \in M_{\mathrm{new}}(f) \cup M_{\mathrm{mod}}(f)$ が存在しないか, m' が命題 10 の m'' に相当するかのいずれかである.

仮定 12 任意の機能 $f \in F$ および f の実行パターン p について, T_p^+ 中の f 固有の実行期間外に実行されたメソッド $m \in M_{\mathrm{new}}(f) \cup M_{\mathrm{mod}}(f)$ の実行に対して, その呼び出し先にメソッド $m' \in M_{\mathrm{new}}(f)$ が存在しないならば, m は命題 10 の m'' に一致する.

仮定 13 任意の機能 $f \in F$ について $f' \xrightarrow{\text{includes}} f$ を満たす機能 $f' \in F$ が存在するとき, f' のある実行パターン p' の実行トレース $T_{p'}^+$ 中で, あるメソッド $m \notin M_{\mathrm{nonexist}}(f')$ の呼び出し元にあるすべてのメソッド m' は $m' \notin M_{\mathrm{nonexist}}(f')$ である.

仮定 14 任意の機能 $f \in F$ について $f' \xrightarrow{\text{includes}} f$ を

満たす機能 $f' \in F$ が存在するとき, f' 固有の実行期間内に実行されたメソッド m は $m \notin M_{\mathrm{nonexist}}(f')$ である.

仮定 15 任意の機能 $f \in F$ について, あるメソッド $m \in M_{\mathrm{mod}}(f)$ の中のブロック $b \in B_{\mathrm{spec}}(f)$ の外側のブロック b' が $b' \in B_{\mathrm{comm}}(f)$ のとき, T_p^+ で b が実行され, T_p^- で b が実行されなかった f の実行パターン p に対して, T_p^+ 中で b に分岐するための条件式の値に影響を与えているメソッド $m' \in M_{\mathrm{new}}(f) \cup M_{\mathrm{mod}}(f)$ が存在する.

仮定 16 任意の機能 $f \in F$ について, あるメソッド $m \in M_{\mathrm{mod}}(f)$ の変更された箇所がブロック $b \in B_{\mathrm{comm}}$ に含まれるとき, T_p^+ および T_p^- で b が実行された f の実行パターン p に対して, T_p^+ 中で b に分岐するための条件式の値に影響を与えているメソッド $m' \in M_{\mathrm{new}}(f) \cup M_{\mathrm{mod}}(f)$ が存在する.

4.4 対話的機能抽出手法

前節で紹介した制約条件の解は, 対象機能を f としたとき C の $C_{\mathrm{new}}(f)$, $C_{\mathrm{mod}}(f)$, $C_{\mathrm{unmod}}(f)$, $C_{\mathrm{nonexist}}(f)$ への分割, および M の $M_{\mathrm{new}}(f)$, $M_{\mathrm{mod}}(f)$, $M_{\mathrm{unmod}}(f)$, $M_{\mathrm{nonexist}}(f)$ への分割の形をとる. それぞれの解が示す抽出範囲は $M_{\mathrm{new}}(f) \cup M_{\mathrm{mod}}(f)$ となる. 一般に制約条件を満たす可能な解の集合およびそれらが示す抽出範囲の候補の集合は膨大な数となるため, たとえその中に正解集合が含まれていたとしても, ユーザにとっては有益な情報とはならない. また, 無数の実行パターンを同時に考慮に入れれば, より有益な情報を導き出すことができるかもしれないが, 現実的には限られた実行パターンしか扱うことができない. そこで, 限られた実行パターンが与えられたときに抽出範囲の候補を極小のものから順番に対話的に絞りながら機能抽出を行う手法を考える. 具体的には以下のように対話的手法を構成する.

1) 与えられた実行パターンに対して, すべての命題と仮定を満たす解の集合とそれらが示す抽出範囲の候補の集合を考え, それらの抽出範囲の候補の中で極小となるものの集合に着目する. (ここで極小の抽出範囲は仮定 7 と仮定 8 から求められ, ソフトウェア偵察による抽出結果を含むことに注意.)
2) 極小となる抽出範囲の候補の数に応じて, 3) または 7) に分岐する.
3) 極小となる抽出範囲の候補がただ 1 つの場合 (最小となる抽出範囲が存在する場合), その候補に対して, その候補を除いたときに極小となる候補の集合を加えたものを抽出範囲の次の候補の集合とする.
4) 抽出範囲の次の候補の集合の中からどの候補を選ぶかによって分類が変わるメソッドを抽出し, それらの各メソッド m に対してユーザに m が $M_{\mathrm{new}}(f)$, $M_{\mathrm{mod}}(f)$, $M_{\mathrm{unmod}}(f)$ のいずれに属するかを問い合わせる.
5) 問い合わせ結果を仮定して解の集合を絞り込む.
6) 絞り込んだ結果得られた解の集合が示す抽出範囲の候補のうち極小となるものが 3) の最小となる抽出範囲のみであった場合, それを解として出力し, そうでない場合 2) に戻る.
7) 極小となる抽出範囲の候補が複数存在する場合, それらの候補の中からどの候補を選ぶかによって分類が変わるメソッドを抽出し, それらの各メソッド m に対してユーザに m が $M_{\mathrm{new}}(f)$, $M_{\mathrm{mod}}(f)$, $M_{\mathrm{unmod}}(f)$ のいずれに属するかを問い合わせる.
8) 問い合わせ結果を仮定して解の集合を絞り込み, その結果得られた抽出範囲の候補のうち極小となるものの集合に着目して 2) に戻る.

抽出範囲の極小の候補を構成するにはできるだけ多くのメソッドを $M_{\mathrm{unmod}}(f)$ に分類する必要があるが, 仮定 7 により $M_{\mathrm{new}}(f)$ に分類されるメソッド, 仮定 8 により $M_{\mathrm{new}}(f)$ または $M_{\mathrm{mod}}(f)$ に分類されるメソッドが存在する場合がある. また $M_{\mathrm{new}}(f)$ に分類されるメソッドが存在する場合, 命題 9 によってその呼び出し元のメソッドが $M_{\mathrm{unmod}}(f)$ に分類される可能性がなくなる場合がある. このように極小の候補の構成には多くの命題および仮定が関係する.

5. 事例研究

提案手法によってどこまで正解集合に近い解を得ることができるかを調べるために, jEdit[†]を用いた事例研究を行った. jEdit はオープンソースのテキストエディタで, そのバージョン 4.3 で追加された機能のうち規模の大きいもの 3 つに対して, 本手法にしたがって手作業で機能抽出を試みた. 対象となる機能の正解集合は文献 [9] のデータ[‡]をもとに定める. しかし, 本研究における実装箇所の定義である最新追加規則と文献 [9] の実装箇所の定義の相違, 機能追加の目的と照らして明らかに不適切と考えられる結果などに対応するため, 正解集合を一部修正した. ユーザが最適な選択を行うという仮定の下で, 4 節で述べた制約条件を満たす解の集合のうち最も正解集合に近いものを提案手法にしたがって手作業で求める. ここで, 提案手法は限られた実行パターンに基づいて機能抽出を行うため, たとえユーザが最適な選択を行ったとしても正解集合が得られるとは限らないことに注意が必要である. なお, 対象プログラムの動的な性質はデバッガを用いて調べる. 本事例研究では提案手法の精度を**適合率**と**再現率**を用いて評価する. 対象機能を f とするとき, f の正解集合を M_f, 本手法によって得られたメソッドの集合を M_f^* とおく. このとき, 適合率は $|M_f \cap M_f^*|/|M_f^*|$ で求められ, 再現率は $|M_f \cap M_f^*|/|M_f|$ で求められる. 対象となる 3 つの機能に対して本手法および既存手法であるソフトウェア偵察とソフトウェア偵察をブロック単位で行うように改良したものを手作業で適用し

† http://www.jedit.org/
‡ http://www.cs.wm.edu/semeru/data/benchmarks/

た. その結果, メソッド単位のソフトウェア偵察では全く抽出できなかった機能に対しても, ブロック単位に改良することによりメソッドを抽出できるようになり再現率が向上することがわかった. さらに, 本手法の適用により適合率を 100%に維持したまま再現率をより向上させられることがわかった. 本事例研究の結果を表 1 に示す. ただし, 文献 [9] の正解集合では無名クラス c 内のメソッドはすべて c が定義されているメソッド m の一部であるとみなされているのに対し, 本手法では m と c 内のメソッドを別々の対象として抽出している. 文献 [9] の正解集合と比較するため, 本手法によって抽出された結果に対しても c 内のメソッドが m の一部であるとみなして, 適合率と再現率を計算している. 表 1 からわかるように, ユーザが最適な選択を行うとするならば適合率が 100%, 再現率が 75%以上の解を得られることがわかった.

6. 考察

前節で紹介した jEdit を用いた事例研究において, 本手法を手動で適用することによって適合率が 100%, 再現率が 75%以上の解を得られることがわかった. 本事例研究では, 正解集合を参照した状態でユーザが最適な選択を行うという仮定の下で手法を適用したため, 適合率が 100%になるのは当然である. しかしながら, 正解集合を網羅できるようユーザが最適な選択を行うと仮定したとしても再現率は 100%に到達しなかった. この原因として以下が挙げられる.

1) 対象機能の特定の振る舞いに至るシステムの操作方法がわからない.
2) 対象機能の中にログ出力などユーザの見えないところで実行され, かつその実行をユーザの操作によって制御できない部分がある.
3) 標準クラスがすでに持っている機能を有効にすることで実現される機能がある.

これらの原因はすべて, 必要なメソッドがソフトウェア偵察によって抽出できないという結果となって現れる. 2) の要因で抽出できない理由は, 対象機能のその部分を実行しない実行トレースを取ることができないためである. 3) の要因で抽出できない理由は, 対象機能を実行したときとしなかったときの差分が標準クラス内で閉じてしまうためである. 1) の要因については, 不具合の検出にどのようなテストケースを用意すればよいかという問題と同等のものであり完全な解決は難しい. いっぽう, 2), 3) の要因についてはなんらかの対処をできる可能性がある. 今後の課題として検討していく予定である.

7. 関連研究

機能抽出技術は大きく動的解析に基づくもの, 静的解析に基づくもの, 情報検索に基づくもの, それらの複合によるものに分けられる. 動的解析に基づく機能抽出ではプログラム実行時の情報を記録した実行トレースが用いられる. 代表的な技術としてソフトウェア偵察 [1] が挙げられる. ソフトウェア偵察では対象機能を実行したときの実行トレースと実行しなかったときのトレースの差分情報に基づいて抽出範囲が定められる. ソフトウェア偵察によって対象機能の実装箇所の一部分を精度良く取り出すことができるが, 実装箇所の広い範囲を網羅的に抽出することはできない. そこで, 使用する実行トレースの数を増やすことによってより広い範囲を抽出できるようソフトウェア偵察を拡張するアプローチが試みられてきた [2][3]. 本研究では, 3 節で説明したように単純に実行トレースの数を増やすだけでは抽出できないような部分まで抽出できるようにすることを目指している.

静的解析に基づく機能抽出では, ソースコードの静的依存関係を利用して抽出範囲の拡張が行われる [4]. 多くの場合動的解析技術と組み合わせて用いられ, 本研究でも静的解析技術は動的解析技術と組み合わせて用いられている. 本研究は, オブジェクト指向プログラムの静的な性質に基づいて静的解析技術を用いている点に特徴がある. また文献 [4] では, 多くのソフトウェアにおいて複数の機能を独立に実行することができないことがあることから, 形式的概念解析を用いて, 個々の機能ではなく機能の集合に対して実装個所を求める手法が提案されている. これに対し本手法では, 特に機能の振る舞いの間の包含関係に留意しつつ, 個々の機能に対して実装個所の抽出を行っている.

情報検索に基づく機能抽出では, クラス名, メソッド名, 変数名などを対象にキーワード検索を行うことによって抽出が行われる [5][6]. そのため抽出範囲を比較的容易に拡張することができるが, 対象機能に関係しないコードまで抽出してしまう可能性がある. また情報検索技術では, なぜ抽出されたコードがその機能に関係しているのかを説明することができない.

Cerberus[7] では, 動的解析技術, 静的解析技術, 情報検索技術の 3 つを組み合わせることによって抽出精度と網羅性の向上が図られている. 本研究では, 上記理由からこれらの中の情報検索技術は用いず, かつ最小限の実行トレースを利用するだけで抽出精度と網羅性を向上させる手法の構築を目指している.

なお, 提案されている手法の多くは自動抽出手法であるが, 本研究と同様, 対話的抽出手法を提案している文献も存在する. たとえば文献 [10] では, 機能の実装個所をソースコードより高い抽象度で探索するための手法として関心事グラフが提案されている. 関心事グラフはプログラム要素間の静的な依存関係を表現したもので, ユーザの操作によって対話的に構築される. いっぽう本研究では, 主に動的依存関係に基づいて抽出範囲の拡張が行われる.

表 1 ユーザが最適な選択を行うという仮定の下での jEdit に対する提案手法の適用結果

文献 [9] における ID	機能の概要	対象機能のトレースの数	対象機能以外のトレースの数		適合率	再現率
1638642	分割画面の仕切り線を移動したときに連続して再描画できるようにする	10	3	本手法	100% (9/9)	75.0% (9/12)
				ソフトウェア偵察 (ブロック単位)	–	8.3% (1/12)
				ソフトウェア偵察 (メソッド単位)	–	0% (0/12)
1593375	アクティビティログを文字列検索できるようにする	2	1	本手法	100% (13/13)	76.5% (13/17)
				ソフトウェア偵察 (ブロック単位)	–	41.2% (7/17)
				ソフトウェア偵察 (メソッド単位)	–	23.5% (4/17)
1578785	外部プログラムよりバッファが変更されたときにプロンプトなしでリロード可能にする	8	2	本手法	100% (9/9)	75.0% (9/12)
				ソフトウェア偵察 (ブロック単位)	–	25.0% (3/12)
				ソフトウェア偵察 (メソッド単位)	–	0% (0/12)

8. おわりに

オブジェクト指向プログラムの性質を利用して, ソフトウェア偵察によって抽出された範囲を対話的に拡張する手法の提案を行った. ユーザが最適な選択を行うという仮定の下で, jEdit のいくつかの機能を対象に提案手法による抽出範囲の調査を行ったところ, 少数の実行トレースを用いるだけで正解集合の 75%以上を網羅できることがわかった. 今後, これらの結果に基づいて手法の詳細な設計を行うと共に, より広い範囲を網羅できるよう手法を拡張していく予定である.

参考文献

[1] Wilde, N. and Scully, M. C.: Software reconnaissance: Mapping program features to code, *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 7, No. 1, pp.49–62 (1995).
[2] Antoniol, G. and Gueheneuc, Y.-G.: Feature identification: A novel approach and a case study, *Proc. International Conference on Software Maintenance (ICSM 2005)*, pp. 357–366 (2005).
[3] Eisenberg, A. D. and Volder, K. D.: Dynamic feature traces: finding features in unfamiliar code, *Proc. International Conference on Software Maintenance (ICSM 2005)*, pp. 337–346 (2005).
[4] Eisenbarth, T., Koschke, R. and Simon, D.: Locating features in source code, *IEEE Transactions on Software Engineering*, Vol. 29, No. 3, pp. 210–224 (2003).
[5] Poshyvanyk, D., Guéhéneuc, Y.-G., Marcus, A., Antoniol, G. and Rajlich, V.: Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval, *IEEE Transactions on Software Engineering*, Vol. 33, No. 6, pp. 420–432 (2007).
[6] Liu, D., Marcus, A., Poshyvanyk, D. and Rajlich, V.: Feature location via information retrieval based filtering of a single scenario execution trace, *Proc. International Conference on Automated Software Engineering (ASE 2007)*, pp. 234–243 (2007).
[7] Eaddy, M., Aho, A. V., Antoniol, G. and Guéhéneuc, Y.-G.: CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis, *Proc. International Conference on Program Comprehension (ICPC 2008)*, pp. 53–62 (2008).
[8] Eaddy, M., Aho, A. and Murphy, G. C.: Identifying, assigning, and quantifying crosscutting concerns, *Proc. Workshop on Assessment of Contemporary Modularization Techniques (ACoM 2007)*, (2007).
[9] Dit, B., Revelle, M., Gethers, M. and Poshyvanyk, D.: Feature location in source code: A taxonomy and survey, *Journal of Software: Evolution and Process*, Vol. 25, No. 1, pp. 53–95 (2013).
[10] Robillard, M. P. and Murphy, G. C.: Concern graphs: Finding and describing concerns using structural program dependencies, *Proc. Internal Conference Software Engineering (ICSE 2002)*, pp. 406–416 (2002).