# 変数の型を考慮したメソッド間の実行経路の検索

**Searching type-sensitive execution paths between method call pairs**

竹之内 啓太 * 石尾 隆 † 井上 克郎 ‡

**あらまし** プログラムの動作を理解するには，その実行経路を調査することが重要である．Java プログラムの場合，開発者は多態性によるメソッド呼び出しを解決しながらそれぞれのメソッドの内部の処理を追いかけ，その中から興味のある動作を拾い出していくことになる．本研究では，このような作業を計算機で支援するために，2 つのメソッド名が与えられると，プログラム中でそれらの 2 つのメソッドが順番に呼び出されるような実行経路を自動的に列挙する手法を提案する．動的束縛を伴うメソッド呼び出しが連続した場合に一貫した呼び出し先の解決を行うことでメソッド呼び出し経路の組み合わせを削減するとともに，ループは高々1 回しか実行されないといった経験的な探索経路の限定を設けることで，多くのメソッド呼び出しの組に対して現実的な時間での実行経路の列挙を可能とした．

## 1 はじめに

開発者にとって，作業対象のプログラムを効率的に理解することは重要である．Ko ら [1] は，研究室で働く 10 人を対象とした調査により，ソフトウェア開発の作業時間の 22%はコードの読解に，16%は依存関係の調査に，13%は検索に費やされており，コードの編集やテストといった開発作業にはそれぞれ 20%，13%しか費やされていないことを報告している．LaToza ら [2] によるマイクロソフト社の開発者を対象とした調査でも，コードの理解，新しいコードの追加，既存のコードの編集，コードに関係のない仕事のそれぞれに，ほぼ同じ時間が割かれていることが報告されている．

プログラム理解の際に開発者が遭遇する問題の多くは実行経路の集合から特定の条件を満たす実行経路を見つけ出す問題，すなわち Reachability Questions として解釈することができる [3]．たとえば，プログラムがある機能を実行したときにエラーメッセージを出力する条件を調査するという場合，その機能を実現しているメソッドの始点から終点までの実行経路の中から，そのエラーメッセージを出力する命令を含む実行経路を見つけ出す Reachability Question として解釈できる．

プログラム中の実行経路を見つけ出す作業では，あらゆる実行経路を考慮することが期待される．しかし，Java のようなオブジェクト指向プログラミング言語では，1 つの要求の実現が呼び出し関係でつながったメソッドにまたがっていることが多い [4]．そのため，動的束縛の解決などを考慮しながら複数のファイルにまたがって実行経路を探索していく必要があるが，このような作業はプログラムの動作を理解する途中の開発者にとっては困難であり，実行経路の見落としなどにつながる可能性もある．

この問題に対して，Kamiya [5] は，指定した 2 つのメソッドの呼び出し関係を開発者に提示する手法を提案している．2 つのメソッドは，注目する実行経路の開始点と終了点に対応しており，たとえば日付を表現する `Calendar` オブジェクトを取得する `getInstance` メソッドと，その日付データを実際に読み出す `get` メソッドを指定すると，日付データがどこで作成され，どこで使用されるかを，プログラムのコールグラフの断片として抽出する．指定されたメソッドが複数の場所で使用さ

*Keita Takenouchi, 大阪大学大学院情報科学研究科

†Takashi Ishio, 大阪大学大学院情報科学研究科

‡Katuro Inoue, 大阪大学大学院情報科学研究科

れていた場合も，それらの断片が順番に提示されるため，開発者はそれらを見落としなく調査することができる．この手法は，開発者の実行経路の理解を助けることが期待されるが，一方で，動的束縛の解決についてはメソッド呼び出し位置ごとに個別に解決しているため，全体としては実行不可能であるような経路を抽出してしまう可能性がある．また，この手法は指定されたメソッドに関連した呼び出し関係しか提示しないため，得られた結果からそれらの間の実行経路や，その途中で実行される他のメソッド呼び出しを探索する作業は開発者に残されている．

本研究では，Kamiya の手法 [5] と同様に 2 つのメソッドが指定されたとき，プログラム中でそれらのメソッドが呼び出される実行経路を抽出する手法を提案する．1 つの経路上で発生する複数のメソッド呼び出しに対して，一貫した動的束縛の解決を行うことで，実行不能な経路の提示を削減し，利用者が実行経路に対してより正確な理解を得ることを可能とする．

以降，2 章で研究の背景を，3 章で提案手法を説明し，4 章で評価実験の結果を示す．5 章で妥当性の脅威について述べ，6 章でまとめと今後の課題について述べる．

## 2 背景

### 2.1 Reachability Questions

LaToza ら [3] は，プログラム理解の際に開発者が遭遇する問題の多くは Reachability Questions として解釈できると述べている．Reachability Questions とは，実行経路の集合から特定の条件を満たす実行経路を見つけ出す問題であり，プログラム動作の因果関係を考えるにあたり有効なものであるとしている．

Reachability Questions の構成要素は，以下の 2 つである．

- 探索対象となる実行経路の集合 $traces$
- 見つけ出す経路の条件 $SC$

実行経路の集合 $traces$ は，対象プログラム $p$，経路の始点の集合 $O$，経路の終点の集合 $D$，束縛条件の集合 $C$ からなる 4 つ組 $traces(p, O, D, C)$ で表わされる．また，見つけ出す経路の条件 $SC$ の要素として，さまざまな述語が定義されており，たとえば $grep(str)$ という関数は $str$ に一致するテキストを含む文を検索することを表現する．この定義に従うと，あるプログラム $p$ において，メソッド $m$ の実行中にエラーメッセージ $errorText$ が出力される条件を調査するという問題は，メソッド $m$ の先頭の命令を $m_{start}$，メソッド $m$ の実行終了を $m_{end}$ とすると次のように表現できる．

$$find\ grep(errorText)\ in\ traces(p, m_{start}, m_{end}, ?)$$

ここで “?” は制約条件を特に指定しないことを意味する．

本研究や Kamiya の手法 [5] は，指定されたメソッド $m_1$，$m_2$ に対して，$m_1$ を呼び出しうるすべてのメソッド呼び出し命令を $O$，$m_2$ を呼び出しうるすべてのメソッド呼び出し命令を $D$ としたときの $traces(p, O, D, ?)$ を（ある一定の制限下で）求め，それを表現する実行系列を出力する手法に相当する．

### 2.2 動的束縛の解決によって生じる実行不可能な経路

Java プログラムにおいて，メソッド呼び出しの動的束縛を静的解析で分析する手法としては Class Hierarchy Analysis [6] や Variable Type Analysis [7] が存在する．これらの手法は，メソッド呼び出し命令ごとに実際に呼び出されうるメソッドを列挙する．たとえば，クラスの継承関係が図 2 であるとき，図 1 のコード例を考える．変数 `a` にクラス `B` あるいはクラス `C` のインスタンスを代入して 2 つのメソッド `f`，`m` を呼び出している．Variable Type Analysis [7] を用いた場合，メソッド呼び出し `a.f` と `a.m` によって実行されうるメソッドの集合は，それぞれ {`A.f`, `C.f`},

```
public static void main(String[] args)
{
  int i = 0;
  A a = null;
  if(i == 0) a = new B();
        else a = new C();
  a.f();
  a.m();
}
```

図 1　動的束縛を使ったコード例

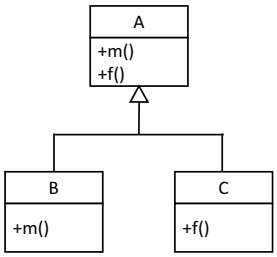


図 2　図 1 の例におけるクラスの継承関係

{B.m, A.m} となる.

個別のメソッド呼び出し解決の正確さとしてはこれで問題ないが，本研究のようにプログラム中での実行経路を考えた場合は実行不可能な経路を生じうる．a.f によって実行されうるメソッドと，a.m によって実行されうるメソッドを単純に連結すると，その組み合わせは次の 4 通りとなる:(i).$(A.f, B.m)$, (ii).$(A.f, A.m)$, (iii).$(C.f, B.m)$, (iv).$(C.f, A.m)$. これらのうち，実際に実行されるメソッドの組み合わせは，変数 a にクラス B のインスタンスが代入されたときの (i)，クラス C のインスタンスが代入されたときの (iv) の 2 通りであり，残る (ii) と (iii) の組み合わせは実行不能である．本研究では，オブジェクトを格納する各変数について，先に実行経路と変数に代入される型を確定することで，各呼び出しについて一貫した結果だけを抽出する．

## 3　提案手法

本研究では，メソッド間の実行経路をメソッドの実行系列として提示する手法を提案する．提案手法の入出力は以下の通りである．

**入力**　プログラム $p$ と 2 つのメソッド名 $m_1, m_2$.

**出力**　$traces(p, m_1, m_2, ?)$ に含まれる実行経路において実行されるメソッド名の列.

ただし，提示する実行経路の組み合わせを抑えるため，求める実行経路には以下の制限がある．

- 対象となるメソッド $m_1, m_2$ は，いずれも main メソッドからのメソッド呼び出しによって到達可能である（解析の対象に含まれないライブラリやフレームワーク等のコールバックによる実行の到達は考慮しない）.
- メソッド呼び出し文が実行されたとき，その直後にその中身のメソッドが実行されるものとする（static イニシャライザなどの割込みは考慮しない）.
- 再帰呼び出しとなるメソッド呼び出しの経路は考慮しない.
- for や while 文のような繰り返し文は，その内容を 1 回だけ実行する場合と 1 回も実行しない場合の 2 つの経路のみを考慮し，2 回以上の実行は考慮しない. この仮定は Kamiya の手法 [5] でも前提とされている.

提案手法は，これらの性質を満たす実行経路を抽出し，それらを提示する．上記の制約は解析を行う上での組み合わせを削減するためのであるが，解析結果を出力する段階でも，入力メソッド $m_1, m_2$ に関係のないメソッド呼び出しはメソッド名のみを提示し，メソッド内の実行経路の提示は省略するなどの情報量の削減を行っている.

提案手法は，以下の手順から構成される.

**手順 1.** 実行経路の探索起点となるメソッドの特定.

**手順 2.** 手続き間実行経路グラフの構築.

**手順 3.** 手続き間実行経路グラフからのメソッド列の抽出.

これらの手順の詳細を以下に述べるが，その具体例として図 3 で示されるサンプルプログラムに対して，メソッド “A.af” から “A.export” への実行経路を探索する

```
public static void main(String[] args)
{
  int i = 0; A a = null;
  if(i == 0) a = new B()
        else a = new C();
  a.init();
  a.f();
  a.m();
  a.export();
}
```

```
public class A {
  public void m(){ }
  public void f(){ af(); }
  private void af(){ }
  public void init(){ }
  public void export(){ }
}
```

```
public class B extends A{
  public void m(){int i = 0;
    if(i==0) bm(); }
  private void bm(){ }
}
public class C extends A{
  public void f(){cf();}
  private void cf(){ }
}
```

**図 3　提案手法の説明用サンプルプログラム**

場合を使用する．

### 3.1　手順 1．実行経路の探索起点となるメソッドの特定

まず最初に，実行経路の探索の起点となるメソッド群を抽出する．このメソッド群は指定されたメソッド $m_1, m_2$ の 2 つのメソッド呼び出しの経路を接続しうるメソッドの集合である．$m_1, m_2$ の両方を呼び出しうるメソッド $s$ があったとすると，その $s$ を呼び出すようなメソッド $t$ が存在したとしても，$t$ から $s$ への呼び出しは実行経路の情報としては有益ではないと考える．そのため，`main` メソッドからのメソッドの呼び出し関係を示したコールグラフを作成したとき，以下の条件を満たすメソッド集合 $M$ を探索の起点として抽出する．

$$M(m_1, m_2) = \{m \in R(m_1) \cap R(m_2) \mid \exists m_c \in calls(m) : (m_c \in R(m_1) \wedge m_c \notin R(m_2)) \vee (m_c \notin R(m_1) \wedge m_c \in R(m_2))\}$$

ただし $R(m)$ はコールグラフ上でメソッド $m$ に到達可能なメソッドの集合，$calls(m)$ はメソッド $m$ が呼び出すメソッドの集合である．

$M(m_1, m_2)$ の各メソッドからはメソッド $m_1, m_2$ に到達できるが，そこから呼び出した他のメソッドからは $m_1, m_2$ のどちらかにしか到達することができない．つまり，これらのメソッドは $m_1, m_2$ の間をつなぐ実行経路に貢献していると考えられる．この条件は Kamiya の手法 [5] における local maximum depth を持つノードの探索の概念に対応する．本研究での実装では，コールグラフの構築には Variable Type Analysis [7] を用いた．

これ以降の手順 2, 3 は，$M(m_1, m_2)$ に含まれる各メソッド $m$ に対して実行し，$m$ が接続する $m_1$ から $m_2$ までの経路を求める方法となっている．$m$ は経路探索の起点，部分的な呼び出し関係の根となることから，以降の手順ではルートメソッドと呼ぶ．ルートメソッドから先の経路にだけ着目することで，手順 2 で作成する手続き間実行経路グラフのサイズを削減している．サンプルプログラムでは，$M($`A.af`$,$ `A.export`$) = \{$`example.main`$\}$ であり，以降の手順の例では `main` メソッドがルートメソッドとなっている．

### 3.2　手順 2．手続き間実行経路グラフの構築

本ステップでは，探索基点として選択されたルートメソッド $m \in M(m_1, m_2)$ から開発者が指定したメソッド $m_1, m_2$ までの手続き間の実行経路を表現したグラフ（以降，手続き間実行経路グラフ）を構築する．サンプルプログラムを対象にルートメソッド `main` を基準として作成した手続き間実行経路グラフの例を図 4 に示す．手続き間実行経路グラフはメソッドの実行を表現するエントリーノード（図中の網掛けで示されているノード）と，メソッド内部での実行経路を表現するパスノード（図中の白色で示されているノード）を接続した有向グラフである．図中の各ノードのラベルの数字は機械的に割り当てられたノードの ID である．図中では紙面の

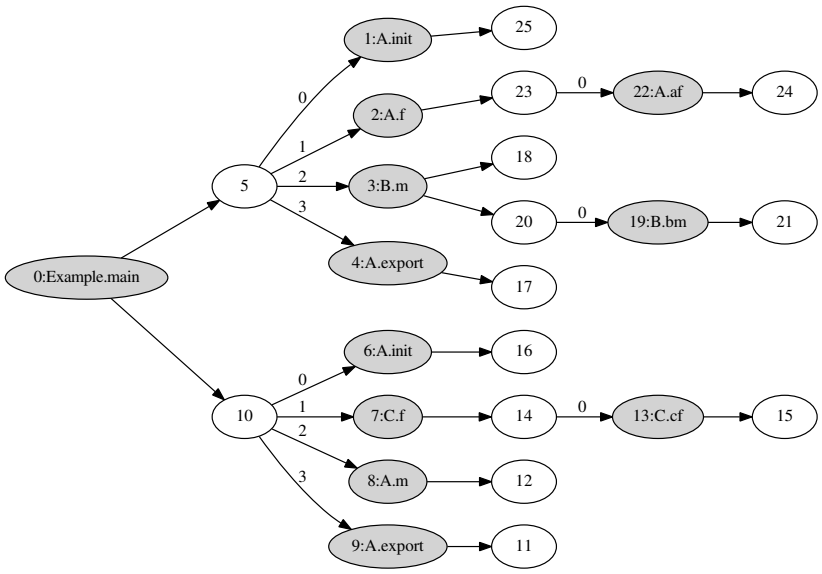


図 4　手続き間実行経路グラフ

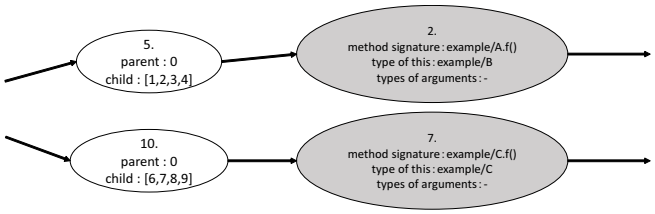


図 5　手続き間実行経路グラフの各ノードの属性

都合上，各ノードが保持する情報を一部省略しており，ID 2, 5, 7, 10 の 4 つのノードの属性情報を図 5 に示す．

エントリーノードはメソッドの実行に対応するノードであり，メソッドシグネチャと，`this` およびメソッド引数として渡されているオブジェクトの型情報を属性に持つ．たとえば図 5 の ID 2 のエントリーノードは，`example/B` クラスのインスタンスをレシーバ（`this`）としてメソッド `example/A.f()` を実行することを表現している．オブジェクトの型情報を属性に持たせることで，このメソッドの実行経路における動的束縛の解決時の制約を表現している．同一のメソッドの実行であっても，引数の型が異なれば，異なる頂点を作成する．各エントリーノードは，その内部での実行経路を表現するパスノードに対する有向辺を持つ．

パスノードは，メソッド内部で呼び出されるメソッドの系列を表現するノードである．実際に呼び出す対象のメソッドに対応するエントリーノードに対して順序付きの辺を持つ．たとえば，図 4 の `main` メソッドに対応する ID 0 のエントリーノードに接続された 2 つのパスノードは，`main` メソッドの実行中に呼び出されるメソッドの系列が

- $[A.init, A.f, B.m, A.export]$
- $[A.init, C.f, A.m, A.export]$

の 2 通りであることを示している．図 5 では，頂点の属性 child として，辺の順序関係を示している．

**Algorithm 1:** ConstructGraph

```
1  Input: rootNode ; ルートメソッドのエントリーノード
2  Output: V_e: エントリーノード集合, V_p:パスノード集合
3  V_e = {rootNode};
4  V_p = φ;
5  worklist = {rootNode};
6  while worklist ≠ φ do
7      v_e ⟵ worklist から取り出す;
8      SSAVars ⟵ v_e における SSA 形式の変数の集合;
9      for p ∈ Paths(v_e) do
10         entryListSet ⟵ AnalyzePath(v_e, p, SSAVars);
11         for entryList ∈ entryListSet do
12             for v'_e ∈ entryList do
13                 if v'_e ∉ V_e then
14                     V_E ⟵ V_e ∪ {v'_e};
15                     worklist ⟵ worklist ∪ {v'_e};
16                 end
17                 v_p から v'_e へのエッジ作成;
18             end
19             if v_p ∉ V_p then
20                 V_p ⟵ V_p ∪ {v_p};
21                 v_e から v_p へのエッジ作成;
22             end
23         end
24     end
25 end
```

手続き間実行経路グラフは，プログラム全体に対して 1 つ作成するのではなく，ルートメソッドごとに 1 つずつ構築する．このアルゴリズムの疑似コードを Algorithm 1 に示す．以下の説明では手順に対応する Algorithm 1 の行番号を括弧を用いて表記する．

まず，ルートメソッドのエントリーノードを作成する．ルートメソッドは外部からどのような型が引数として指定されるかを仮定せず，単一のノードを作成する．グラフの初期状態として，エントリーノードの集合 $V_e$ にルートメソッドに対応するエントリーノードを入れ，パスノードの集合 $V_p$ は空とする (3–4)．*worklist* はまだパスノードを求めていないエントリーノードの集合である．*worklist* にルートノードのエントリーノードを入れ，この集合が空になるまで以下 (7–24) の計算を繰り返す．最初に，*worklist* からエントリーノードをひとつ取り出し，そのエントリーノードを $v_e$ とする (7)．$v_e$ に対応するメソッドにおいて，メソッド内のローカル変数の単一代入（SSA）形式の変数の集合を求める (8)．また，$v_e$ に対応するメソッドの制御フローグラフから，メソッドの実行開始点から終了点まで，ループを高々1 回しか通過しないような実行経路をすべて求める．疑似コード中ではこの処理を 9 行目の $Paths(v_e)$ という手続きで表現している．求めた実行経路すべてについて，経路を $p$，それに対応するパスノードを $v_p$ として，実行経路の内部の解析を実行する．経路の内部の解析では，手続き *AnalyzePath* を実行して実行経路 $p$ 上で呼び出されうるメソッド系列の集合を，エントリーノードの列の集合として取得する．この実行されうるメソッドの系列を求める処理は別アルゴリズムとして後述する．

**Algorithm 2:** AnalyzePath

```
1  Input: v_e:エントリーノード
2         p:実行経路
3         SSAVars:SSA 形式の変数の集合
4  Output: entryListSet : エントリーノード列の集合
5  SSATypeMap ⟵ ResolveSSAVarTypes(SSAVars, p);
6  entryListSet ⟵ φ;
7  for SSATypeMap の変数の型の組み合わせごとに実行 do
8      entryList ⟵ ϵ;
9      for c ∈ MethodCall(p) do
10         receiverType ⟵ SSATypeMap(Receiver(c));
11         arguments ⟵ ArgmentsTypes(c);
12         methodSignature ⟵
           InvokedMethod(receiverType, MethodName(c));
13         c の呼び出し先メソッドを表現するエントリーノード v_c を作成;
14         v_c に属性 methodSignature, receiverType, arguments を登録;
15         entryList ⟵ concat(entryList, v_c);
16     end
17     entryListSet ⟵ entryListSet ∪ {entryList};
18 end
```

1 つの実行系列から得られたエントリーノード列それぞれに対し，その呼び出し列を表現するパスノードを構築するため，以下の手順 (12–22) を行う．実行されるエントリーノードの列の要素 $v'_e$ それぞれについて，$v'_e$ が $V_e$ に含まれないのであれば，$v'_e$ を $V_e$ と $worklist$ に追加する (13–16) ．そして，$v_p$ から $v'_e$ にエッジを作成するとともに $v_p$ の属性として $v'_e$ を登録していく (17)．すべての $v'_e$ に対し処理を終えたら，$v_p$ と同一属性を持つ（同一のメソッド呼び出し系列を表現する）パスノードが $V_p$ にすでに含まれているかどうかを調べる．もし含まれていなかった新たに追加し，$v_e$ から $v_p$ へのエッジを作成する．すでに含まれているのなら $v_p$ は追加せず破棄する (19–22)．

あるエントリーノード $v_e$ における実行経路 $p$ で呼び出されるメソッドの系列を求める手続き $AnalyzePath$ の疑似コードを Algorithm 2 に示す．この手続きの入力はエントリーノードと実行経路，SSA 形式の変数の集合であり，出力はメソッド呼び出しに対応するエントリーノードの列の集合である．$AnalyzePath$ の計算では，まず，実行経路 $p$ におけるデータフローに基づいて SSA 形式の各変数に代入されうるオブジェクトの型の範囲を決定する (5)．ローカル変数はその変数の定義部（型の代入文）から，メソッド引数はエントリーノードに与えられた属性から型の範囲を決定することができる．また，フィールドやメソッドの戻り値が表現する型の集合は Variable Type Analysis を用いて型を求めている．以下 (8–17) は，SSA 形式の各変数の型の組み合わせごとにひとつのエントリーノードの列を求める手順である．まず，実行経路 $p$ に並んでいる各メソッド呼び出しについて，SSA 形式の変数の型情報からそれぞれのメソッド呼び出しのレシーバオブジェクトの型，メソッド引数の型を決定する (10–11)．レシーバオブジェクトの型を使って，動的束縛を解決し，実際に呼び出されるメソッドのシグネチャを調べる (12)．実行されるメソッド，レシーバオブジェクトの型，メソッド引数の型の制約を持った $v_c$ をエントリーノードの列に追加する (12–15)．すべてのメソッド呼び出しをエントリーノードに変換したら，その系列を返す．手続き間実行経路グラフの構築後，メソッドの再帰呼び出しを回避するため，ルートメソッドのエントリーノードから深さ優先したと

きの後退辺となるエッジは取り除く．

### 3.3 手順 3. 手続き間実行経路グラフからのメソッド列の抽出

手続き間実行経路グラフは，ルートメソッド $m$ から $m_1$ および $m_2$ への経路を構成しうるメソッド呼び出しの列を表現している．このステップでは，得られた手続き間実行経路グラフから，$m_1$ の呼び出しから $m_2$ の呼び出しまでに実行されるメソッド系列を取得する．まず，入力の 2 つのメソッド $m_1, m_2$ に対応するエントリーノードを探索し，それぞれに対しノードの集合 $V(m_1), V(m_2)$ を求める．$v'_1 \in V(m_1), v'_2 \in V(m_2)$ となるすべての $(v'_1, v'_2)$ の組に対して以下の計算を行い，手続き間実行経路グラフの部分グラフを求める．

まず，それぞれの先祖ノードの集合 $Ancestor(m'_1), Ancestor(m'_2)$ を求め，$Ancestor(m'_1, m'_2) = Ancestor(m'_1) \cup Ancestor(m'_2)$ とすると，$Ancestor(m'_1, m'_2)$ に含まれるパスノードと，その子となるすべてのエントリーノードからのみ構成される部分グラフを考える．この部分グラフは，$m'_1, m'_2$ に関係する呼び出しについてのみ実行経路を考慮し，それ以外のメソッド呼び出しについては実行経路の情報を含まない．サンプルプログラムに対してこの手順を行うと，まず，入力の 2 つのメソッド $A.af$ と $A.export$ に対応するノード ID の組は (22, 4) または (22, 9) である．(22, 4) の組に対しては $Ancestor(22, 4) = \{0, 2, 4, 5, 22, 23\}$ となり，このノードに関係のあるエッジのみで構成される部分グラフは図 6 となる．

部分グラフに含まれるエントリーノードそれぞれについて，ただ 1 つのパスノードを選択していく，つまり実行経路を選択していくと，最終的に 1 つの実行経路を取得することができる．ただし，$m_1$ から $m_2$ の間に実行されるメソッド列のみを提示するため，パスノードにおける順序関係で $m_1$ に至るまでの経路上かそれより後に実行されるメソッド，$m_2$ に至るまでの経路上かそれより前に実行されるメソッドだけを探索する．

図 6 の場合，ノード ID 5 のパスノードから $m_1$ である `A.af` に到達するにはラベル 1 の辺を通過しなければならないため，ラベル 0 の辺は探索から除外し，1〜3 のラベルのみを探索する．この部分グラフは各エントリーノードが 1 つのパスノードしか持たないため，ただ 1 つの経路に対応する．結果として，図 7 のようなメソッド列が抽出され，これが提案手法の出力となる．図中の $*$ 印がついているメソッドが入力のメソッドであり，この結果から，`A.af` が実行されてから `A.export` が実行されるまでの間にソースコード中の `a.m()` によって実行されるメソッドが `B.m` であることが分かる．動的束縛をメソッド呼び出しごとに個別に扱っていると，ここで `A.m` も候補として表示されることになり，開発者にとっては提示される経路数の増加の原因となっていた．この手続き間実行経路グラフではノード ID が 3 であるエントリーノードにも内部の経路が存在していたが，これは $m_1, m_2$ とは直接関係がないため，提示するメソッド列では内部の経路が省略されている．これにより入力メソッドとは関係のない部分の情報を隠すとともに，提示するメソッド列の数の爆発を抑えている．なお，ノード ID が (22, 9) の組の部分グラフの場合は，パスノードのいかなる組み合わせの部分グラフであっても ID 22 のノードと ID 9 のノードを同時に含むものは存在しないため，提示されるメソッド列は存在しない．結果として図 7 の系列が提案手法の唯一の出力となる．

## 4 評価実験

本手法における中間的な評価として，手続き間実行経路グラフの構築に関する以下の 2 点を調査を行った．

1. 本手法にはどれくらいの時間を要するか．
2. 本手法での用いる手続き間実行経路グラフはどのくらいのサイズになるか．

メソッド内の実行経路数は爆発的に増える可能性があるため，これが原因で手続き間実行経路グラフのパスノードの数が爆発し，全体の実行時間に大きく影響を与え

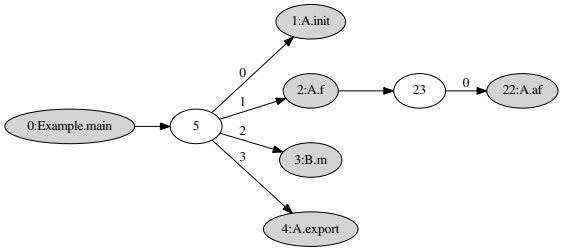


図 6　入力メソッドに関連する部分グラフの抽出結果

```
example/Example.main(){
 example/A.f(){
  *example/A.af()
 }
 example/B.m()
*example/A.export()
}
```

図 7　得られるメソッド列

表 1　ルートメソッドとしたときに 1 分以内にグラフ構築が完了したメソッド数

| プログラム | 全メソッド数 | 1 分以内にグラフ構築が完了した数 | 割合 |
|---|---|---|---|
| ant-1.8.4 | 9713 | 7327 | 0.754 |
| jedit-4.3.2 | 7786 | 5924 | 0.761 |
| junit-4.11 | 1243 | 1243 | 1.000 |

る可能性があると考えられる．そこで本実験ではルートメソッドが決まったときの手続き間実行経路グラフの構築時間を測定し，実用的な時間内にグラフが構築できるかを評価した．対象とするプログラムは ant-1.8.4, jedit-4.3.2, junit-4.11 のバイトコードである．使用した計算機の CPU は Intel Xeon 2.90GHz，メモリは 256GB である．

### 4.1　実験方法

対象の Java プログラム中の全メソッドについて，それぞれのメソッドをルートノードとしたときの手続き間実行経路グラフの構築時間を調べ，1 分以内で終わったものの割合を調べた．また，構築した手続き間実行経路グラフのエントリーノード数とパスノード数の関係を調べ，エントリーノード数が増加するにつれて，パスノード数がどのように増加する傾向があるのかを調査した．

### 4.2　結果と考察

対象の Java プログラム中の全メソッドについて，それぞれのメソッドをルートメソッドとしたとき，1 分以内に手続き間実行経路グラフの構築が完了したものの数を表 1 にまとめた．この表は左からプログラム名，全メソッド数，そのうち 1 分以内に手続き間実行経路グラフの構築が完了したメソッドの数，全メソッド数に対する完了したものの割合を示している．この結果より，1 分あれば 7 割以上のメソッドに対してグラフの構築が完了するといえる．よって，本手法は多くの場合に適用可能であると考えられる．

jedit, junit に対し構築した手続き間実行経路グラフのエントリーノードの数とパスノードの数を関係を図 8 に示す．この図は構築した全ての手続き間実行経路グラフについて，横軸をエントリーノードの数，縦軸をパスノードの数としてプロットしたものである．この図からはエントリーノードの数が増加するにつれパスノードが増加する傾向が読み取れるが，エントリーノードの数に対しパスノードの数は爆発的には増加していないことが分かる．紙面の都合上省略したが，ant にも同じ傾向が見られた．パスノードの数に依存して提示する実行経路数が増えるため，本手法は提示する経路数の爆発を抑えるのに有効であるといえる．

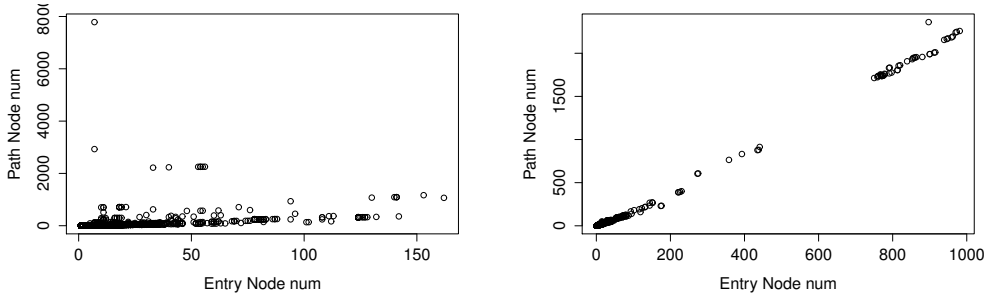


**図 8　jedit, junit を対象として構築した手続き間実行経路グラフの頂点数（左 : jedit, 右 : junit）**

## 5　妥当性への脅威

本研究ではローカル変数とメソッド引数の型を決定することで実行されるメソッドを特定し，手続き間実行経路グラフを構築した．評価実験では多くのメソッドに対して適用可能であることを示しているが，提案手法は `main` から到達可能なメソッドに対してのみの適用であり，すべてのメソッドが解析対象となっているわけではない．フィールドに代入される変数の型の解析には Variable Type Analysis を用いており，提案手法が出力する実行経路でも代入されることのない型も動的束縛の解決に含まれている可能性がある．そのため，あらゆる実行不能な実行経路を出力から排除しているわけではない．

## 6　まとめと今後の課題

本研究ではプログラムの実行経路への理解を支援するため，実行経路の列挙を実行するメソッド呼び出しの列という形で提示する手法を提案した．また，評価実験では本手法が多くの場合で現実的な時間で完了すること，列挙する実行経路の数が爆発的には増加しないことを確かめた．今後の課題としては，開発者にとって有益な情報の提示範囲と方法を検討し被験者実験による有効性の評価をすること，Kamiya の手法と比較したときの質的・量的な評価をすることがあげられる．

**謝辞**　本研究は JSPS 科研費 26280021 の助成を受けたものです．

### 参考文献

[1] Andrew J. Ko, Htet Aung, and Brad A. Myers. Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective maintenance tasks. In *Proceedings of the 27th International Conference on Software Engineering*, pages 126–135, 2005.

[2] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, pages 492–501, 2006.

[3] Thomas D. LaToza and Brad A. Myers. Developers ask reachability questions. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 185–194, 2010.

[4] Benedikt Burgstaller and Alexander Egyed. Understanding where requirements are implemented. In *Proceedings of the International Conference on Software Maintenance*, 2010.

[5] Toshihiro Kamiya. An algorithm for keyword search on an execution path. In *2014 International Conference on Software Maintenance, Reengineering and Reverse Engineering*, 2014.

[6] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, August 1995.

[7] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. *SIGPLAN Notice*, 35(10):264–280, October 2000.