# 特別研究報告

題目

# インスタンスの型を考慮した Java プログラムの実行経路の列挙手法の提案

指導教員
井上 克郎 教授

報告者
竹之内 啓太

平成 27 年 2 月 13 日

大阪大学 基礎工学部 情報科学科

平成 26 年度 特別研究報告

# インスタンスの型を考慮した Java プログラムの実行経路の列挙手法の提案

竹之内 啓太

**内容梗概**

開発者がプログラム理解するにあたり遭遇する問題の多くは，Reachability Questions として解釈することができることが知られている．Reachability Questions とは，プログラムの実行経路の中から特定の実行経路を見つけ出す問題である．この問題の解決には実行経路の列挙が必要となる．

そこで，本研究では指定したメソッドの始点から終点までの実行経路の列挙を出力するツールを作成した．この実行経路はメソッド間の呼び出しを考慮したものであり，呼び出し元と呼び出し先を関連付けるために既存の静的解析の手法を用いた．しかし，これらの手法をそのまま用いると，各呼び出しを独立に考慮してしまうことになり，実行不能経路を生み出してしまうという問題点がある．この問題に対し，本研究ではメソッド内のメソッド呼び出しをインスタンスの型を考慮した系列として捉えることで解決を試みた．評価実験では，インスタンスの型を考慮することにより実行不能経路を削減する提案手法の有効性を，それを組み込んだ場合と組み込まない場合の実行経路数を比較することで評価した．その結果，約 2.2%のメソッドに対して提案手法は有効であり，実行経路数が削減できたメソッドでは，中央値で 60~80%にまで列挙する実行経路数を削減することができた．

**主な用語**

静的解析
制御フロー
動的束縛の解決

# 目 次

## 1　まえがき

Ko ら [1] は，開発者は設計やテスト以外の作業に多くの時間を割いていることを指摘している．この研究では，研究室で働く 10 人を対象とした調査により，その作業時間の 22%がコードを読むのに使われ，20%がコードの編集，16%が依存関係の調査，13%が検索，13%がテストに使われていることを明らかにした．また，[2] のマイクロソフト社の開発者を対象とした調査では，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，それぞれにほぼ同じ時間が割かれていることが分かった．よって，知見のないプログラムを効率的に理解することは重要なことであるといえる．

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 として解釈できることを明らかにした．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．たとえば，プログラム実行時にあるエラー文が出力されたとき，そのエラー文がメソッドの実行中のどこで発生しているのかを調査したい，という問題はメソッドの始点から終点までの実行経路のうち，そのエラー文を含む実行経路を見つけ出す Reachability Question として解釈できる．

Reachability Questions の解決には，実行経路の列挙が必要である．ここで，実行経路の列挙は，動的に行う手法と静的に行う手法が考えられる．動的な手法では，実行時情報を用いることで，正確な実行経路を得ることができる．ただし，列挙された実行経路が可能性のあるものを全て網羅しているとは限らない．また，準備として，入力データの用意や，実行時情報の取得が必要となるため，準備に必要なコストは大きくなる．静的な手法では，ソースコードの解析を行い，実行される可能性のある経路を全て列挙する．準備には，ソースコードを用意するだけでよいため，動的な手法に比べると準備に必要なコストは低い．

本研究では，準備として必要なコストが低く，実行可能な経路を網羅できる，静的な実行経路の列挙を行う．ただし，静的に実行経路の列挙を行う場合には，実行できる可能性のある経路を全て列挙することになるため，実行不能な経路が出力に含まれてしまうという問題がある．実行不能な経路は，列挙された実行経路を開発者が読解するコストを増加させてしまうため，できる限り減らしておくことが望ましい．

現在広く使われているオブジェクト指向プログラミング言語では，実行時のオブジェクトの型により実際に呼ばれるメソッドが変化する，動的束縛が行われる．そのため，静的に実行経路を得るには，動的束縛で呼び出される可能性のあるメソッドを特定しておく必要がある．これには多くの既存手法が提案されており，CHA[4],RTA[5], VTA[4], k-CFA[6] あるいは，ポインタ解析 [7] が適用できる．ただし，これらの手法はメソッド呼び出し箇所ごとに個別に呼び出されるメソッドを取得するため，実行不能な経路が含まれてしまう場合があった．

そこで本研究では，メソッド呼び出しのレシーバーオブジェクトの型を，メソッド全体で考慮することにより，実行不能経路を削除する手法を提案する．提案手法は制御フローグラフの作成，メソッド呼び出し系列の集合の作成，系列に基づいた制御フローグラフの探索，経路の列挙という 4 つの段階から構成される．制御フローグラフの作成とその探索で実行経路を列挙することができるが，探索時にメソッド呼び出し系列の集合の作成を行うことにより，上記で説明した実行不能経路を列挙の中から除外している．

評価実験ではこの実行不能経路を削減する手法の有効性を確かめた．Java プログラムを対象に解析を行い，手法を用いる場合と用いない場合に列挙されるそれぞれの実行経路数を比較した．その結果，約 2.2%のメソッドに対して提案手法は有効であり，実行経路数が削減できたメソッドでは，中央値で 60~80%にまで列挙する実行経路数を削減することができた．また，一部のメソッドは 1 割以下にまで実行経路数を削減することができた．

以降，第 2 節では本研究の背景を述べる．3 章では，手法に用いる用語の説明を述べる．4 章では，提案手法を説明する．5 章では，提案手法の評価実験を述べる．6 章では関連研究を紹介する．最後に，7 章で本研究のまとめと今後の課題を述べる．

## 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として解釈できることを述べている．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は以下の2つから構成される．

- 探索対象となる実行経路
- 見つけ出す経路の条件

論文中ではこれらを定式化しており，探索対象となる実行経路の集合を$TR$，見つけ出す経路の条件の集合を$SC$としている．実行経路の集合$TR$の要素は，$p$:対象プログラム，$O$:始点の集合，$D$:終点の集合，$C$:束縛条件の集合，の4つの組$trace(p, O, D, C)$と表わされる．また，見つけ出す経路の条件$SC$の要素として，さまざまな関数が定義されている．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の集合を表している．

プログラム実行時にあるエラー文$errorText$が出力されたとき，そのエラー文はメソッドの実行中のどこで発生しているのかを調査したい，という問題はメソッドの始点から終点までの実行経路のうちで，エラー文$errorText$を含む命令を通過する実行経路を見つけ出すReachability Questionとして解釈できる．これを式として表すと，$p$:対象プログラム，$m_{start}$:メソッドの始点，$m_{end}$:メソッドの終点として

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

となる．ただし，”?” は特に指定しないという意味であり，$traces(p, m_{start}, m_{end}, ?)$は$trace(p, m_{start}, m_{end}, ?)$を要素に持つような$TR$である．このように，開発者が遭遇する問題の多くはReachability Questionsとして式で表すことができる．

### 2.2 メソッドの動的束縛に対する静的解析

実行経路を列挙するためには，メソッドの呼び出し元と呼び出し先を関連付ける必要がある．オブジェクト指向型言語のソースコード中にレシーバaに対するメソッド$m$の呼び出し$a.m$があるとき，この呼び出しによってどのメソッドが呼ばれるのかを静的に解析する手法を，本節では説明する．説明のため，サンプルプログラムを作成した．サンプルコードとクラスの継承関係を図1に示す．

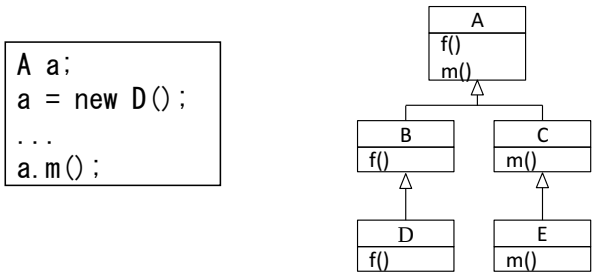



図 1: サンプルコードとクラスの継承関係

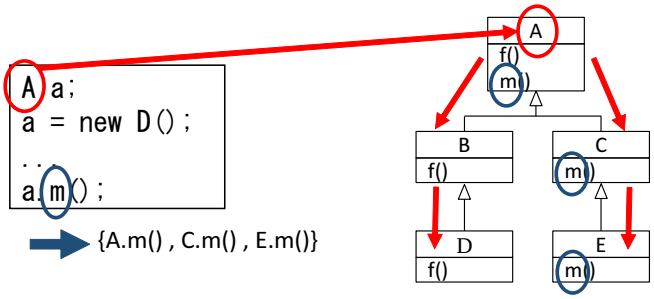



図 2: Class Hierarchy Analysis

### 2.2.1 Class Hierarchy Analysis

Class Hierarchy Analysis(CHA)[4] はクラスの継承関係を利用し，メソッド呼び出しを解析する手法のひとつである．メソッド呼び出し $a.m$ によって呼ばれるメソッドを解析する手順は次のとおりである．まず，メソッド呼び出しのレシーバ $a$ のインスタンスが宣言されている型を調べる．次に，宣言されているクラスのサブクラスの集合（自身を含む）を求め，それぞれのクラスから親クラスをたどっていき，最も近い親クラスが持つメソッド $m$ を $a.m$ によって呼び出される可能性のある候補として挙げる．

サンプルプログラム中のメソッド呼び出し $a.m$ を対象に CHA を適用したときの流れを図 2 に示す．レシーバ $a$ が宣言されている型は $A$ であり，クラスの継承関係より $A$ のサブクラスの集合は $\{A, B, C, D, E\}$ である．それぞれのクラスから親クラスをたどっていき，メソッド $m$ をもつクラスを候補として追加していく．$\{A, B, D\}$ はクラス $A$ へ，C は C へ，E は E へたどり着く．よって，$a.m$ が呼び出し可能性のあるメソッドの候補は $\{A.m, C.m, E.m\}$ となる．

CHA は実装が比較的容易であるが，宣言型のみを考慮するため，候補の中に呼び出される可能性のないものを多く含んでしまう可能性がある．

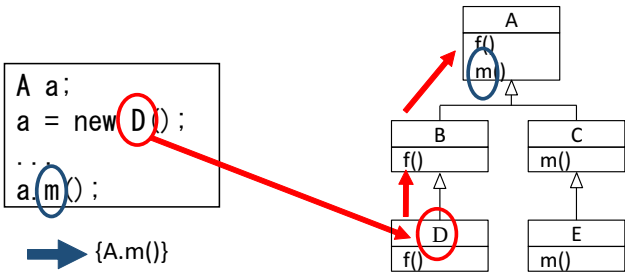



図 3: Variable Type Analysis

### 2.2.2 Variable Type Analysis

Variable Type Analysis(VTA)[4] はメソッド呼び出しのレシーバに代入される型を考慮し，メソッド呼び出しを解析する手法のひとつである．この手法の手順を次に示す．まず，レシーバに代入される可能性のある型を調べる．そして，その型のクラスの親クラスをたどっていき，最も近い親クラスが持つメソッド $m$ を $a.m$ によって呼び出される可能性のある候補として挙げる．

サンプルプログラム中のメソッド呼び出し $a.m$ を対象に VTA 解析したときの流れを図 3 に示す．代入文より，レシーバ a は D クラスのインスタンスであるので，クラスの継承関係にもとづいて $D$ の親クラスをたどっていく．最初にたどり着く，メソッド $m$ をもつクラスは $A$ であるので，このメソッド呼び出しの候補として $\{A.m\}$ を挙げる．

VTA はインスタンスの代入されている型を考慮するため，CHA より精度が高い．そのかわり，ひとつの Java プログラムを対象としたとき，プログラムの各メソッドに対するデータフロー解析を必要とする．

## 2.3 経路列挙における問題点

オブジェクト指向型言語において，CHA や VTA を用いてプログラムの実行経路の列挙をする際，メソッド呼び出し関係が原因となって実行不能経路を含んでしまうという問題点がある．

たとえば，図 4 に示すソースコードを考える．ソースコード中のメソッド呼び出し $a.f$ と $a.m$ によって呼び出されるメソッドをそれぞれ VTA によって解析したとき，$a.f$ に対しては $\{D.f, A.f\}$，$a.m$ に対しては $\{A.m, E.m\}$ という候補を挙げる．それぞれの呼び出しによって 2 つの候補が挙げられるのは，レシーバのインスタンスの型が $D$ または $E$ の 2 通りであるためである．これらの結果を独立に用いて実行経路を列挙するとき，$a.f$ と $a.m$ によって呼ばれるメソッドの組み合わせは，(($a.f$ に呼ばれるメソッド) , ($a.m$ に呼ばれるメソッド))

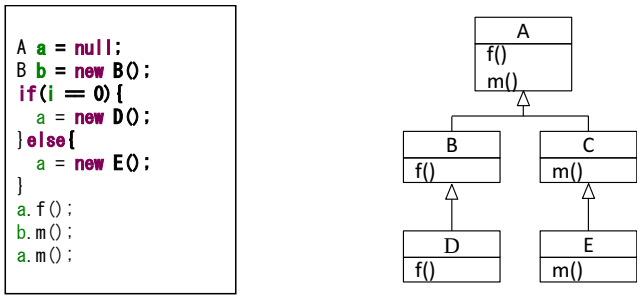



図 4: サンプルコードとクラスの継承関係

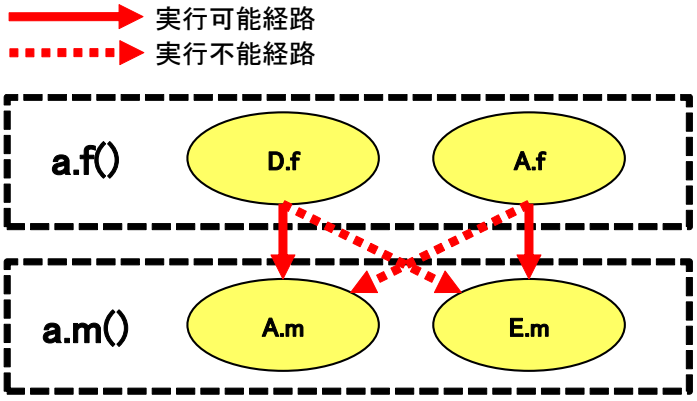



図 5: 実行可能経路と実行不能経路

として，

(i). $(D.f, A.m)$

(ii). $(D.f, E.m)$

(iii). $(A.f, A.m)$

(iv). $(A.f, E.m)$

の 4 通りとなる．しかし，実際のメソッド呼び出しの組み合わせは $a$ のインスタンスの型が $D$ のとき (i)，$E$ のとき (iv)，の 2 通りとなる．つまり，(ii) と (iii) の組み合わせの実行経路は実行不能である．

このように，プログラムの実行経路を列挙したとき，メソッド呼び出しを独立に考慮することが原因で実行不能経路が含まれる．この問題を解決するため，本研究ではメソッド呼び出しを系列とみなし，同じオブジェクトが格納されている変数の型をまとめて考慮した．

# 3　用語の説明

この章では次章の説明で用いる用語の説明を行う．

## 3.1　制御フローグラフ

プログラムの実行経路を考えるには，プログラムの制御の流れを捉える必要がある．プログラムの制御の流れをグラフ形式で表現したものが制御フローグラフ [8] であり，コンパイラの最適化や静的解析において一般的に用いられる．本研究でも，制御フローグラフの探索によって実行経路の列挙を行っている．制御フローグラフにおいて，頂点はプログラムの基本ブロック，辺は制御の流れを表している．以下では，基本ブロック，辺の説明とその作成方法を説明する．また，制御フローグラフの具体例は次章の図 9 に示している．

### 3.1.1　基本ブロック

基本ブロックはプログラムを制御にもとづいて分割したときの最低限の分割単位であり，基本ブロック内ではその間に分岐や合流がない．そのため，基本ブロック内の命令は上から下へ順に実行されるものとして考えることができる．

Java プログラムの 1 つのメソッドを基本ブロックに分割するため，以下のような Java バイトコード命令を基本ブロックの先頭と定義する．

1. そのメソッドの先頭．
2. 飛び越し文の行き先の命令．
3. 条件付き飛び越し文の直後の命令．

これらの先頭文のひとつから次の先頭文までが基本ブロックとなる．

### 3.1.2　制御フローグラフの辺

制御フローグラフの辺は基本ブロックどうしをつなぐ制御の流れを表す．辺は有効辺であり，基本ブロック A から基本ブロック B への辺があるとき，基本ブロック A の実行の後に基本ブロック B の実行が行われる可能性があることを示している．辺は以下のいずれかの場合に基本ブロック A から基本ブロック B への辺が作成される．

1. 基本ブロック A に飛び越し命令があり，その行き先が基本ブロック B の先頭である．
2. 基本ブロック A の直後が基本ブロック B の先頭である．

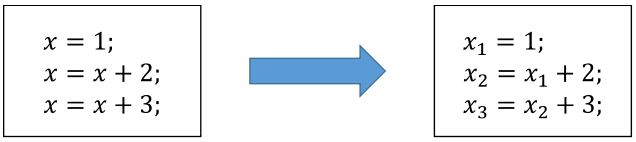



図 6: 静的単一代入形式への変換例

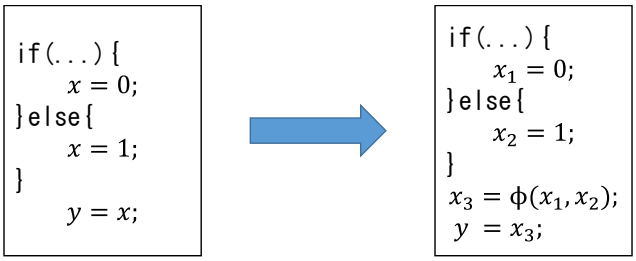



図 7: $\phi$ 関数の使用

### 3.2 静的単一代入形式

静的単一代入形式 [8] とはプログラム上のすべての変数の使用に対し，対応する定義が 1 か所しかないように表現したものである．静的単一代入形式に変換するには，一般にプログラム中の変数名を変更することが必要である．簡単な例を図 6 に示す．この例では，もとのプログラムでの変数 $x$ を $x_1$,$x_2$,$x_3$ へ変換し，各変数が一度しか値が代入されない形式となっている．

図 6 では分岐構造が存在しないため，変数名を変更するだけで静的単一代入形式への変換ができた．しかし，図 7 のような例では，$x$ の使用に対する定義が 2 か所で行われているため，変数名の変更だけでは対応ができない．このような場合は $\phi$ 関数と呼ばれる関数を使用する．$\phi$ 関数は引数のうちいずれかの値を返す関数であり，$x_3 = \phi(x_1, x_2)$ という代入文は変数 $x_3$ へ，$x_1$ または $x_2$ の値を代入するという意味である．

静的単一代入形式では使用する変数がひとつの定義に対応しているため，変数名が同じであればその値が必ず同じであるという性質がある．本研究では，同じインスタンスが格納された変数の集合を求めるために静的単一代入形式を用いた．

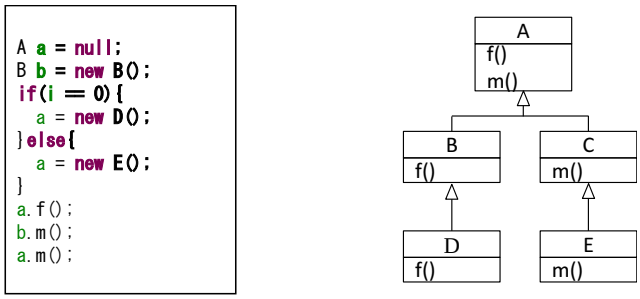



図 8: サンプルコードとクラスの継承関係

## 4 提案手法

本章では提案手法を説明する．本手法では，解析対象となるプログラム $p$ とメソッド $m$ という入力に対し，

$$traces(p, m_{start}, m_{end}, ?)$$

という $TR$ を返す．対象とするプログラムのバイトコードのみを用いる静的解析手法である．

提案手法は，以下の 4 つのステップから構成される．

(i). 制御フローグラフの作成

(ii). メソッド呼び出し系列の集合の取得

(iii). 系列に基づいた制御フローグラフの探索

(iv). 経路の列挙

(i) で解析対象のメソッドの制御フローグラフを作成し，(ii) で各命令がどのメソッドを呼び出すのかを示した系列の集合を取得する．このとき，インスタンスの型を考慮することで，問題点として挙げた実行不能経路を列挙から削除することにつながる．これらをもとに，(iii) で制御フローグラフの探索を実施する．そして，(iv) でこの探索により見つかった，指定したメソッドの始点から終点までの実行経路の列挙を行う．

本節ではこれらの処理を詳しく述べる．手法の説明のため 2.3 節で用いたサンプルプログラムを例として用いたため．図 8 に再掲する．

### 4.1 制御フローグラフの作成

まず，解析対象となるプログラムに含まれるメソッドごとの制御フローグラフを作成する．メソッド中のバイトコードを基本ブロックに分割し，それぞれをグラフの頂点とする．

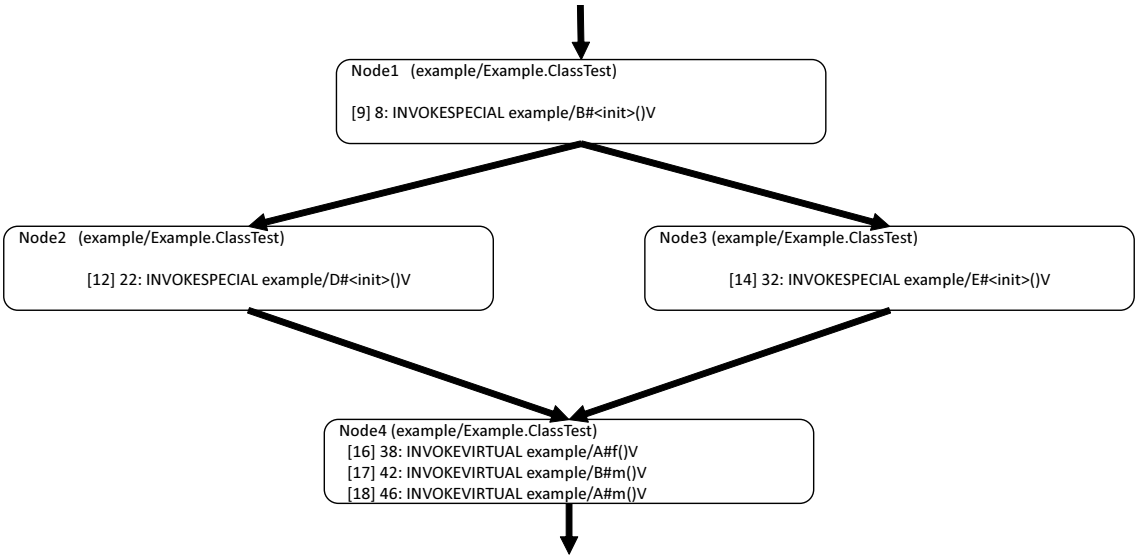



図 9: 制御フローグラフ

このグラフの各頂点は基本ブロック内の情報である以下の情報を保持する．

- メソッド呼び出し命令
- 基本ブロックの含むメソッドのメソッドシグネチャ
- ソースコードの行番号

制御フローグラフの辺はメソッド内の基本ブロックの飛越し命令に基づいて作成する．

サンプルプログラムに対して作成した制御フローグラフは図 9 となる．頂点が基本ブロックとなっており，その要素としてバイトコードのメソッド呼び出し命令を登録している．各命令は左からソースコードの行番号，バイトコードの命令番号，命令内容となっている．

### 4.2 メソッド呼び出し系列の集合の作成

このステップではレシーバのインスタンスの型を考慮したときのメソッド呼び出し系列の集合を作成する．作成されるメソッド呼び出し系列は表 1 のような系列であり，動的束縛の影響を受ける各メソッド呼び出し命令 (INVOKEVIRTUAL 命令) で呼ばれるメソッドを表したものとなる．表の 1 行目は変数 $a$ にクラス $D$ のインスタンスが代入された場合の呼び出しを示しており，2 行目は変数 $a$ にクラス $E$ のインスタンスが代入された場合の呼び出しを示している．このように，あらかじめ動的束縛の結果を系列として保持することで，制御フローグラフの探索時に実際に実行されるメソッドの情報を系列から取得するだけで，レシーバのインスタンスの型を考慮した実行経路の探索が可能になる．

表 1: メソッド呼び出し系列の集合

| |
|---|
| $(38 : D, f), (42 : A.m), (46 : A.m)$ |
| $(38 : A, f), (42 : A.m), (46 : E.m)$ |

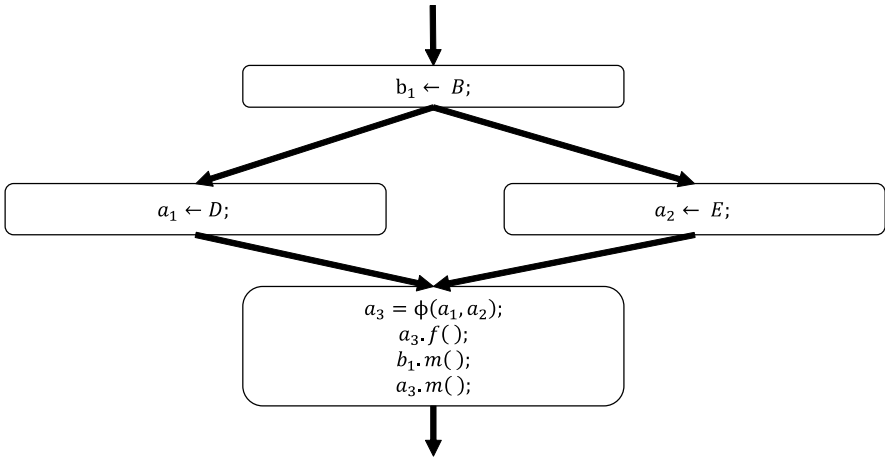



図 10: サンプルプログラムの静的単一代入形式

まず，対象メソッド $m$ を静的単一代入形式形式で表す．メソッド $m$ を単一代入形式で表したときの，オブジェクトを格納するすべての変数の集合を $V(m)$ と定義する．

サンプルプログラムを静的単一代入形式で表した結果を図 10 に示す．サンプルプログラムのメソッド名をここでは $s$ とすると，

$$V(s) \quad = \quad \{a_1, a_2, a_3, b_1\}$$

が得られる．

次に，$V(m)$ の要素の変数 $v$ に対して VTA を適用することで，$v$ に入る可能性があるインスタンスの型の集合 $T(m, v)$ を求める．サンプルプログラムについて，この集合を求めると以下のようになる．

$$\begin{aligned} T(s, a_1) &= \{D\} \\ T(s, a_2) &= \{E\} \\ T(s, a_3) &= \{D, E\} \\ T(s, b_1) &= \{B\} \end{aligned}$$

ここで，メソッド $m$ における変数への型の代入の組の集合 $B(m)$ を以下のように定義する．これはメソッド $m$ を静的単一代入形式で表した時の，オブジェクトを格納するすべての変数とその代入される可能性がある型の組み合わせの集合となる．

$$B(m) = \prod_{v \in V(m)} \{(v,t) | t \in T(m,v)\}$$

サンプルプログラムのメソッド $s$ に対してこの集合を求めると，

$$\begin{aligned} B(s) &= \prod_{v \in V(s)} \{(v,t) | t \in T(s,v)\} \\ &= \{(a_1, D)\} \times \{(a_2, E)\} \times \{(a_3, D), (a_3, E)\} \times \{(b_1, B)\} \\ &= \{((a_1, D), (a_2, E), (a_3, D), (a_1, B)) \\ &\quad , ((a_1, D), (a_2, E), (a_3, E), (a_1, B))\} \end{aligned}$$

となる．これは，メソッド $s$ において，$a_3$ はクラス $D$ あるいは $E$ のオブジェクトを格納するという 2 通りの代入だけが存在することを示している．

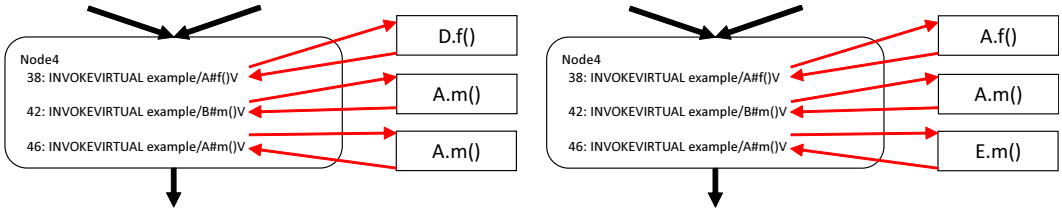
メソッド $m$ において，変数 $v \in V(m)$ とその型が決まれば，実行されるメソッドは一意に定まるため，$B(m)$ からメソッド呼び出し命令の命令番号と実行されるメソッドのタプルの系列の集合 $L(m)$ を作成することが可能である．たとえば，$a_3$ にクラス $D$ のオブジェクトが代入されている場合，命令 38(INVOKEVIRTUAL example/A#f()V) は $D.f$ を実行する．

サンプルプログラムのメソッド $s$ について $B(s)$ より $L(s)$ を作成すると，メソッド呼び出し命令の命令番号の集合は {38, 42 ,46} であるため，以下のとおりとなる．

$$\begin{aligned} L(s) &= \{\{(38 : D.f), (42 : A.m), (46 : A.m)\}, \\ &\quad \{(38 : A.f), (42 : A.m), (46 : E.m)\}\} \end{aligned}$$

### 4.3 系列に基づいた制御フローグラフの探索

探索は指定したメソッドの始点をスタート地点として，深さ優先探索で制御フローグラフを探索していく．探索するメソッドについて前節で述べた表 1 のメソッド呼び出し系列の集合 $L(m)$ を取得しておき，その各系列について探索を行う．到達した頂点のバイトコード命令をひとつずつ読んでいき，タプルに含まれるバイトコードの命令を読んだときは，対応するメソッドの先頭の頂点から探索していく．呼び出し元のメソッドの情報はスタックに詰んでおき，メソッドの終点に到達したら，スタックから値を取り出し，呼び出し元のメソッドに戻る．また，通過した頂点はリストとして保持しておき，通過経路を記録していく．ループによって探索が終わらないことを回避するため，通過経路に含まれている頂点に再び到達したとき，その経路の探索は打ち切る．探索を開始したメソッドの終点に到達したとき，そ




(a) $\{(38 : D.f), (46 : A.m), (46 : A.m)\}$ のとき (b) $\{(38 : A.f), (46 : A.m), (46 : E.m)\}$ のとき

図 11: 制御フローグラフの探索

のときの通過経路を実行経路の集合へ加えていく．すべての経路の探索が終わったとき，この集合の要素がすべての実行経路となる．

サンプルプログラムについての探索の例を図 11 に示す．(a) はメソッド呼び出し系列 $\{(38 : D.f), (46 : A.m), (46 : A.m)\}$ のときの探索であり，このとき列挙される実行経路には，呼び出すメソッドの列として $D.f \to A.m \to A.m$ が得られる．また，(b) はメソッド呼び出し系列 $\{(38 : A.f), (46 : A.m), (46 : E.m)\}$ のときの探索であり，同様に $A.f \to A.m \to E.m$ が得られる．

## 5　評価実験

本研究では提案手法の量的評価を行った．提案手法を組み込んだ場合と組み込まない場合に列挙する実行経路数を比較し，手法を用いない場合に列挙する実行経路の中からどれだけの実行不能経路を削減できるかを評価した．

### 5.1　実験方法

対象プログラムに含まれるメソッドひとつひとつに対し，メソッドの始点から終点までの実行経路数を求めた．このとき，手法の有効性を確かめるため，手法を組み込んだ場合の実行経路数と手法を組み込んでいない場合の実行経路数を求めた．

ただし，探索時間は非常に長い時間掛かってしまう場合もあるため，一定の時間で探索を打ち切ることにした． 本実験では，5 分で打ち切りとした．

探索したメソッドを以下の 3 つに分類し，それぞれのメソッドの個数を調べた．また，(iii) のメソッドの実行経路数の比較，元の経路数に対する減少後の経路数の割合の調査を行った．

(i). 5 分以内に探索が探索が終わらなかったもの

(ii). 5 分以内に探索が探索が終わり，手法が有効でなかったもの

(iii). 5 分以内に探索が探索が終わり，手法が有効であったもの

実験の対象は Java 言語で書かれた以下のプログラムのバイトコードを用いた．

- SVNKit 1.8.8
- Apache Xerces 2.10
- Quartz 1.8.3

### 5.2　結果

探索した全メソッド数 (上記の (i)+(ii)+(iii))，5 分以内に探索が完了したメソッド数 (上記の (ii)+(iii))，そのうち手法により経路数が減ったメソッド数 (上記の (iii)) を表 2 にまとめた．また，手法により経路数が減ったメソッドの実行経路数を表 3 に示す．この表は各システムについて実行経路数の最小値，第一四分位数，中央値，平均値，第三四分位数，最大値をまとめたものであり，上の行は手法を用いたときの経路数の数値，下の行は手法を用いなかったときの経路数の数値である．また，手法により経路数が減ったメソッドに対し，

$$Ratio(m) = \frac{(\text{手法を用いたときの実行経路数})}{(\text{手法を用いなかった実行経路数})}$$

表 2: 各メソッドの個数

| システム名 | 全メソッド数 | 探索が完了した数 | 手法により経路数が減った数 |
|---|---|---|---|
| SVNKit | 3123 | 2467 | 75 |
| Apache Xerces | 1355 | 1195 | 11 |
| Quartz | 1049 | 808 | 37 |

表 3: 手法の効果があったメソッドの実行経路数の比較

| システム名 | 手法の有無 | Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|---|---|---|---|---|---|---|---|
| SVNKit | あり | 12 | 60 | 121 | 6398 | 2682 | 278667 |
| | なし | 30 | 72 | 147 | 19223 | 12618 | 334395 |
| Apache Xerces | あり | 8 | 53 | 133 | 3707 | 2622 | 28476 |
| | なし | 18 | 196 | 201 | 6453 | 3864 | 48960 |
| Quartz | あり | 6 | 6 | 53 | 15120 | 4840 | 279900 |
| | なし | 10 | 16 | 408 | 34240 | 39170 | 466600 |

をまとめたものが表 4 である．この値は，メソッドについての手法を用いなかったときの実行経路数に対する，手法を用いたときの実行経路数の割合を示す．この値の各プログラムについてのヒストグラムを図 12 に示す．(a) が SVNKit 1.8.8，(b) が Apache Xerces 2.10, (c) が Quartz 1.8.3 のヒストグラムである．

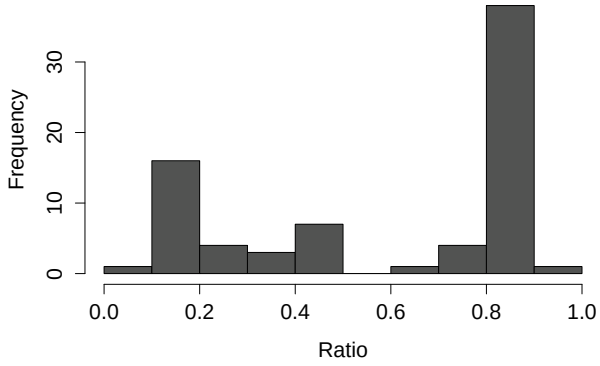
### 5.3 考察

探索を実施した全メソッドのうち，探索が 5 分以内に完了し，手法の効果があったものは約 2.2%であった．本研究の手法は，

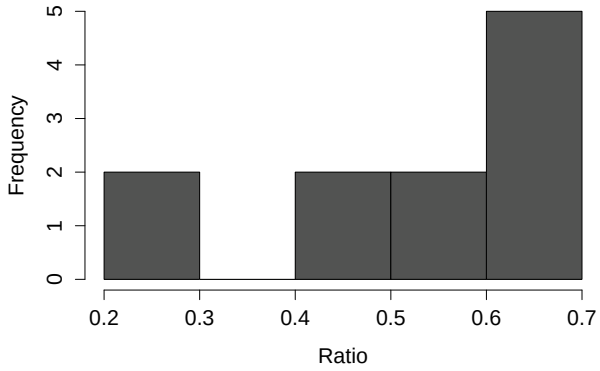
- メソッド呼び出しのレシーバに複数の型が入り，
- 同じインスタンスに対するメソッド呼び出しが連続している

という場合に実行経路を削減できるが，この構造が予想していたよりも少なかったため，期待していたほどの結果は得られなかったと考えられる．

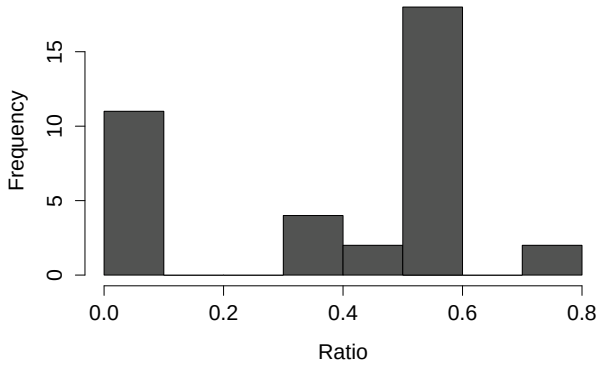
ただし，探索が完了したメソッドには，ゲッターやセッターといった制御構造が単純で，実行経路の削減を考える必要のないものが多く含まれている．そのため，Reachability Questions




(a) SVNKit 1.8.8 のヒストグラム




(b) Apache Xerces 2.10 のヒストグラム




(c) Quartz 1.8.3 のヒストグラム

図 12: 手法を用いなかった実行経路数に対する，手法を用いたときの実行経路数の割合

表 4: 手法を用いなかった実行経路数に対する，手法を用いたときの実行経路数の割合

| システム名 | Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|---|---|---|---|---|---|---|
| SVNKit | 0.01619 | 0.22010 | 0.82090 | 0.57860 | 0.83430 | 0.99980 |
| Apache Xerces | 0.26370 | 0.4680 | 0.5909 | 0.5481 | 0.6786 | 0.6786 |
| Quartz | 0.04412 | 0.06298 | 0.60000 | 0.41310 | 0.60000 | 0.75710 |

の対象となるような，ある程度複雑なメソッドに対しては，この数値以上の有効性が期待できると考えられる．

手法の効果があったメソッドに対しては，表 4 より列挙する実行経路数を中央値で 6〜8 割にまで削減できたことが分かる．列挙する実行経路数が 10 分の 1 以下になったメソッドも見られたため，手法の有効性がかなり大きい場合もあるといえる．

## 6　関連研究

Reachability Questions の解決へのアプローチとして，実行経路を列挙する方法だけでなく，メソッド呼び出し関係を可視化したコールグラフの提案もされている．

Reachability Questions に対する調査を支援するツールとして，LaToza らは REACHER と呼ばれるメソッドのコールグラフの可視化ツールを提案した [9]．REACHER は Eclipse 等に付属している通常のコールグラフと異なり，基準となるメソッドから推移的な呼び出し元，呼び出し先を含めた広い範囲の呼び出し関係をグラフ形式で表示する．さらに検索機能を有しており，あるメソッド m の実行後に検索対象のメソッド m' が呼び出されるまでのメソッド関係を表示したり，m の実行前に m' が呼び出されるときの呼び出し関係を表示することができる．これらは Reachability Questions における実行経路の列挙の支援となり，被験者実験でも調査時間や正確性の向上が認められた．他に，Reachability Questions の支援となるツールとして，神谷ら [10] のツールがある．このツールは，任意の 2 つのメソッド呼び出しのコールグラフでの位置関係を調査することができる．これを利用するとプログラムが，あるメソッドを実行した後にどこを実行するのかという理解がしやすくなり，プログラムの実行経路の把握につながる．

## 7　まとめと今後の課題

本研究では，プログラム理解につながる Reachability Questions に答えるため，プログラムの実行経路の列挙を出力するツールを作成した．また，メソッド呼び出しが原因となり実行不能経路が列挙に含まれてしまう問題に対処するため，メソッド内におけるメソッド呼び出しをレシーバの型を考慮したメソッド呼び出しの系列として捉えることで，実行不能経路を削減した．この手法の評価実験を行い，提案手法が一部のメソッドで有効にはたらくことを確かめた．

今後の課題としては，メソッド間の情報の伝播と探索方法の改善が挙げられる．

本論文の手法では，インスタンスの型をメソッド内にのみ伝播させているが，メソッド間にもこの情報を伝播させることができれば，現時点での列挙に含まれる実行不能経路をさらに削減できると考えられる．メソッド間で情報を伝播させるには，メソッドの呼び出し元のレシーバの型を呼び出し先の this の型として伝播させる方法や，メソッド呼び出しの引数となっているインスタンスの型を呼び出し先で受け取る引数の型として扱う方法などが考えられる．

探索方法の改善が必要であると考えられるのは，現状の深さ優先探索ではプログラムの規模が大きくなると，グラフの探索に莫大な時間がかかる可能性があるためである．規模の大きなグラフの全経路を探索する方法として，Zero suppressed BDD(ZDD)[11] が有効であることが知られているので，これを本手法に組み込むことができれば，規模の大きなプログラムにも対応することが可能であると考えられる．

## 謝辞

本研究において, 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.

本研究において, 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深く感謝いたします.

本研究において, 逐次適切な御指導および御助言を頂き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深く感謝いたします.

本研究において, 様々な御指導および御助言を頂き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鹿島悠氏に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] 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 ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 185–194, 2010.

[4] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. *SIGPLAN Not.*, 35(10):264–280, October 2000.

[5] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 324–341, 1996.

[6] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages -or- Taming Lambda*. PhD thesis, Carnegie Mellon University, 1991.

[7] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Department of Computer Science, University of Copenhagen, May 1994.

[8] 中田育男. **コンパイラの構成と最適化** *(***第** *2* **版***)*. 朝倉書店, 2009.

[9] Thomas D. LaToza and Brad A. Myers. Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2011, Pittsburgh, PA, USA, September 18-22, 2011*, 2011.

[10] T. Kamiya. An algorithm for keyword search on an execution path. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, 2014.

[11] Donald E. Knuth. *The Art of Computer Programming Vol 4, Fascicles 1.* Addison-Wesley Professional, 2009.