動的データ依存関係解析を用いたJavaプログラムスライス手法

廣瀬 航也† 大畑 文明† 井上 克郎†‡

† 大阪大学大学院 基礎工学研究科 情報数理系専攻
〒 560-8531 大阪府豊中市待兼山町 1-3
Phone: 06-6850-6571 Fax: 06-6850-6574
‡ 奈良先端科学技術大学院大学 情報科学研究科
〒 630-0101 奈良県生駒市高山町 8916-5
E-mail: {k-hirose, oohata, inoue}@ics.es.osaka-u.ac.jp

あらまし プログラムデバッグを効率よく行うための手法として，プログラムスライスがある．プログラムスライスは一般的に，プログラム文間の制御依存関係およびデータ依存関係を解析することで得られる．また依存関係解析には，低コストではあるが精度の低い静的解析と，精度は高いが多大なコストを要する動的解析がある．

本発表では，近年，ソフトウェア開発環境で多く利用される，Java を対象としたプログラムスライス手法を提案する．Java は，オブジェクトへの参照・動的束縛など，実行時に決定される要素を多数含むため，静的解析では限界がある．提案手法では，静的解析と動的解析を組み合わせることにより，スライスの精度を高める．

キーワード プログラムスライス，静的解析，動的解析，Java

Program Slicing Method Using Dynamic Data Dependence Analysis for Java Programs

Koya Hirose†, Fumiaki Ohata† and Katsuro Inoue†‡

†Graduate School of Engineering Science, Osaka University
1-3 Machikaneyama-cho, Toyonaka, Osaka 560-8531, Japan
Phone: +81-6-6850-6571 Fax: +81-6-6850-6574
‡Graduate School of Inforamtion Science,Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0101, Japan
E-mail: {k-hirose, oohata, inoue}@ics.es.osaka-u.ac.jp

Abstract Program slice is proposed as a technique to do the program debugging efficiently. Program slice is obtained by generally analyzing the control dependence and the data dependence between program statements. Moreover, the dependence analysis is divided into a static analysis which requires a little cost but produces results with low accuracy, and a dynamic analysis which produces results correctly but requires a large cost.

In this paper, we propose a program slice technique intended for Java programs. Java is popularly used by the software development environment in recent years. There is a limitation in a static analysis since Java contains a lot of dynamically determined issues such as reference to an object and dynamic binding. Our technique improves accuracy of slice by combining static analyses with a dynamic analysis.

Key words program slice, static analysis, dynamic analysis, Java

1 まえがき

プログラムデバッグを効率よく行うための手法として，**プログラムスライス**(*Program Slice*)[6]が提案されている．プログラムスライスとは，直観的には，ある文 s のある変数 v の値に影響を与え得る文の集合である．一般に，プログラムスライスは，プログラム文間の依存関係を解析することで求められる．これまで，手続き型言語を対象とした様々な解析によるプログラムスライス計算手法の提案がなされてきた．主なものに，**静的スライス**(*Static Slice*)[6]，**動的スライス**(*Dynamic Slice*)[1]，**Dependence-Cache(DC) スライス**(*Dependence-Cache Slice*)[2] 等がある．

また，現在のソフトウェア開発環境において，C などの手続き型言語だけでなく，Java, C++等いわゆるオブジェクト指向言語の利用が高まっている．オブジェクト指向言語には，従来の手続き型言語にはないクラス，継承などの新しい概念が導入されており，既存のプログラムスライス計算手法をそのまま用いても正確なスライスが得られない．そこで，オブジェクト指向言語への対応を試みた静的スライス [3]，動的スライス [7] が提案されている．オブジェクト指向言語には，実行時に決定される要素が多く含まれるため，静的スライスでは正確性に限界がある．一方，動的スライスは実行系列を保存するため，多大な時間，空間コストが必要となる．

本稿では，オブジェクト指向言語である Java を対象に，静的スライスと動的スライスの中間に位置する DC スライスを適用する．DC スライスでは動的にデータ依存解析を行うことで，実行時に決定されるデータ依存関係を正確に解析できる．提案手法は，DC スライスの手法を Java に適応することで，実行時に決定される要素の解析が行え，静的スライスより正確なスライスの抽出が行える．また，制御依存関係解析を静的に行い，実行系列の保存を行わないため，解析コストが動的スライスより小さくなる．

以降，2 ではプログラムスライスについて述べる．3 では，DC スライスについて述べる．4 では，動的データ依存関係解析を用いた Java スライス手法について述べる．最後に 5 で，まとめと今後の課題について述べる．

2 プログラムスライス

プログラムスライシング(*Program Slicing*) 技術とは，プログラム中のある文 s におけるある変数 v(**スライス基準** (s, v) と呼ぶ) に対して v の値に影響を与える全ての文をプログラムから抽出する技術で，その結果取り出された文の集合をプログラムスライスまたは単に**スライス**(*Slice*) と呼ぶ．

スライスは，デバッグ，保守，プログラム理解等に利用される．

2.1 スライス計算手順

一般に，プログラムスライスは以下の手順で計算される．

Step 1 依存関係解析
プログラム文間の制御依存関係およびデータ依存関係を調べる．

Step 2 プログラム依存グラフ作成
依存関係解析を元に**プログラム依存グラフ**(*Program Dependence Graph, PDG*) を作成する．

Step 3 グラフ探索
スライス基準に対応する節点から PDG の辺を逆向きにたどる．到達された節点がスライスとして抽出される．

2.2 依存関係

次の条件をすべて満たすとき，文 s から文 t への**制御依存関係**(*Control Dependence*) があるという．

- 文 s が条件文である．
- 文 t が実行されるかどうかは，文 s の結果に依存する．

また，以下の 3 つの条件をすべて満たすとき，文 s から文 t へ変数 v に関する**データ依存関係**(*Data Dependence*) があるという．

- 文 s で変数 v を定義している．
- 文 t で変数 v を参照している．
- 文 s における変数 v の定義が文 t に到達する

2.3 PDG

PDG の各節点は，プログラムに含まれる代入文，入出力文，条件判定文，手続き呼出し文などの各文を表し，各辺は 2 つの文間の依存関係を表す．制御依存関係を表す辺を制御依存辺，データ依存関係を表す辺をデータ依存辺と呼び，それぞれ $\rightharpoonup, \overset{v}{\rightarrow}$ とあらわす (v はデータ依存関係が生じる原因となる変数の名前)．

一般にプログラムには複数の手続きが定義され

```
 1: #include <stdio.h>
 2: #define SIZE 5

 3: int cube(int x) {
 4:     return x*x*x;
 5: }

 6: void main(void)
 7: {
 8:     int a[SIZE];
 9:     int b[SIZE];
10:     int c, d, i;

11:     a[0] = 0;
12:     a[1] = -1;
13:     a[2] = 2;
14:     a[3] = -3;
15:     a[4] = 4;

16:     for (i=0; i<SIZE; i++) {
17:         b[i] = a[i];
18:     }

19:     scanf("%d", &c);
20:     d = cube(b[c]);
21:     if (d < 0)
22:         d = -1 * d;
23:     printf("%d", d);
24: }
```

図 1: ソースプログラム

ている．各手続き間には引数や大域変数を通じてデータ依存関係が生じる．また，手続き呼び出し文から呼ばれた手続きには制御の流れが生じる．そこで，PDG にプログラム中の文とは直接対応しない節点を用意し，その節点が手続き間の中継ぎをすることで手続き間に生じるデータ依存関係，制御依存関係を表す．

図 1 の C のソースプログラムに対応する PDG は図 2 のようになる．

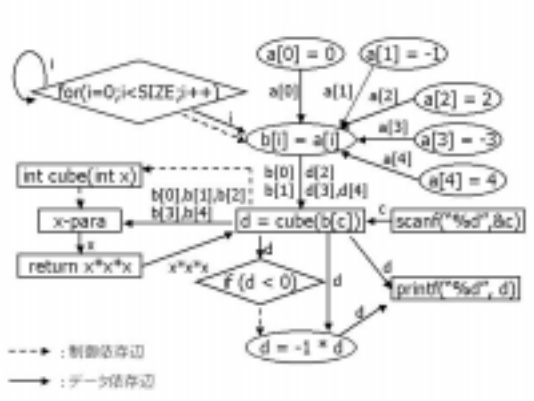

図 2: プログラム依存グラフ

```
s d
* *  1: #include <stdio.h>
* *  2: #define SIZE 5

* *  3: int cube(int x) {
* *  4:     return x*x*x;
* *  5: }

* *  6: void main(void)
* *  7: {
* *  8:     int a[SIZE];
* *  9:     int b[SIZE];
* * 10:     int c, d, i;

*   11:     a[0] = 0;
*   12:     a[1] = -1;
* * 13:     a[2] = 2;
*   14:     a[3] = -3;
*   15:     a[4] = 4;

* * 16:     for (i=0; i<SIZE; i++) {
* * 17:         b[i] = a[i];
* * 18:     }

* * 19:     scanf("%d", &c);
* * 20:     d = cube(b[c]);
* * 21:     if (d < 0)
*   22:         d = -1 * d;
* * 23:     printf("%d", d);
* * 24: }
```

図 3: 図 1 のプログラムの (23.d) に関するスライス

2.4 静的スライスと動的スライス

静的スライスは，制御依存関係とデータ依存関係を共に静的解析して得られる．起こりうる全ての入力を考慮するため，不必要な情報を含むことがあり，その結果，スライスサイズが大きくなるという問題がある．

動的スライスは，制御依存関係とデータ依存関係を共に動的解析して得られる．ある入力データに関する実行系列 (実際に実行された文の並び) を扱うため，スライスサイズを減らすことができるが，全ての実行系列を保存し，そこから依存関係を抽出するために，多大な時間，空間コストを必要とする．

図 1 のプログラムを対象に，スライス基準 (23, d) に関する各スライスを図 3 に示す．なお，動的スライスの抽出においては，入力に 2 を与えた．列 s, d の”*”で記された文がそれぞれ静的スライス，動的スライスに含まれる文を示している．

3 Dependence-Cache スライス

配列を含むプログラムを静的に解析する際，配列の添字の値を把握するのは困難で，不要に多くの依存関係を抽出してしまうことがある．また，ポ

インタを含むプログラムを静的解析する際においても，ポインタによるエイリアスによって，陽には現れないデータ依存関係が発生するため，その依存関係を知るために，各ポインタが何を指しているかを把握しておく必要がある．point-to グラフ [5] を使った静的な解析手法が提案されているが，正確性に限界がある．

DC スライスは，動的にデータ依存関係を抽出することで，配列の添字，ポインタの参照先を把握する．一方，制御依存関係は静的に解析し，実行系列を保存することはしないため動的スライス抽出に比べて実行時間の短縮が得られる．

3.1 計算手順

以下に DC スライスの計算手順を示す．

Step 1 実行前解析

静的に PDG を構築する．まず，プログラム中の各文や条件節に対応する節点を用意する．制御依存関係を解析し制御依存辺を引く．この段階ではデータ依存解析は行わず，データ依存辺は引かない．よって PDG は節点と制御依存辺のみからなる．

Step 2 実行時データ依存関係解析

スライス対象プログラムをある入力について実行しながら，3.2 で述べる方法でデータ依存関係を解析し，PDG にデータ依存辺を追加する．

Step 3 スライスの抽出

スライス基準に対応する節点から PDG を逆向きにたどり，到達された節点に対応する文，条件節がスライスとして抽出される．

3.2 動的データ依存関係解析

実行中に，ある文 s である変数 v が使用された時，v がどの文 (t) で定義されたかが分かれば，t から s への v に関するデータ依存関係があることが分かる．逆に言えば，v を定義してる文さえ分かればデータ依存関係を知ることができる．

そこで，全ての変数に対して，その値を定義したのはどの文か，という情報を持たせておき，その変数の使用があった場合には，その情報からデータ依存関係を把握する．

プログラムに現れる各変数 v に対してキャッシュ $C(v)$ を用意する．プログラムの各実行地点において，$C(v)$ には変数 v が最後に定義された文番号が格納されている．文 s が実行された際に変数 v が

- 使用された場合，$C(v)$ に対応する節点から s に対応する節点に v に関するデータ依存辺を引く．
- 定義された場合，$C(v)$ の値を s の文番号に更新する．

```
1  a[0]:=0;
2  a[1]:=1;
3  a[2]:=2;
4  readln(c);
5  b:=a[c]+5;
6  writeln(b);
```

図 4: 配列を含むプログラム

表 1: 図 4 のプログラムにおける，各実行時点でのキャッシュの推移

実行文	$a[0]$	$a[1]$	$a[2]$	b	c
1	1	-	-	-	-
2	1	2	-	-	-
3	1	2	3	-	-
4	1	2	3	-	4
5	1	2	3	5	4
6	1	2	3	5	4

3.3 適用例

3.3.1 配列を含むプログラム

図 4 のプログラムに対して，動的データ依存関係解析を行う．入力に 0 を与えた場合を考える．各実行時点における各変数 v のキャッシュ $C(v)$ の推移を表 1 に表す．

文 1, 2, 3 では，それぞれ変数 $a[0], a[1], a[2]$ が定義されているので，文 3 を実行した時点で $C(a[0]) = 1, C(a[1]) = 2, C([2]) = 3$ となる．文 4 では，c を定義しているので，$C(c) = 4$ となる．文 5 では，$a[0]$ を使用している．文 5 の実行直前には $C(a[0]) = 1$ であるので，文 1 から文 5 へ $a[0]$ に関するデータ依存辺を引く．同様に c を使用しており，文 4 から文 5 へ c に関するデータ依存辺を引く．また，b を定義しているので，$C(b) = 5$ となる．

3.3.2 ポインタを利用するプログラム

図 5 のプログラムに対して，動的データ依存関係解析を行う．各実行時点における各変数 v のキャッシュ $C(v)$ の推移を表 2 に表す．

文 1, 2, 3, 4 では，それぞれ変数 a, b, c, d が定義されているので，文 4 を実行した時点で $C(a) = 1, C(b) = 2, C(c) = 3, C(d) = 4$ となる．

文 5 では，c を使用している．文 5 の実行直前には $C(c) = 3$ であるので，文 3 から文 5 へ c に関するデータ依存辺を引く．また，$*c$ を定義しているので，$C(*c) = 5 (C(a) = 5)$ となる．

文 6 では，b，d，$*d$ を使用しているので，文 2, 4, 3 から文 6 へデータ依存辺を引く．さらに，文 6 では $**d$ を定義しているので，$C(**d) = 6 (C(a) = 6)$

```
1  a=2;
2  b=1;
3  c=&a;
4  d=&c;
5  *c=5;
6  **d=b;
7  printf("%d",a);
8  printf("%d",**d);
```

図 5: ポインタを利用するプログラム

表 2: 図 5 のプログラムにおける，各実行時点でのキャッシュの推移

実行文	a	b	c	d
1	1	-	-	-
2	1	2	-	-
3	1	2	3	-
4	1	2	3	4
5	5	2	3	4
6	6	2	3	4
7	6	2	3	4
8	6	2	3	4

となる．

文 7 では，a を使用している．文 7 の実行直前には $C(a) = 6$ であるので，文 6 から文 7 へ a に関するデータ依存辺を引く．

文 8 では，$d, *d, **d$ を使用しているので，文 $4, 3, 6$ から文 8 へデータ依存辺を引く．

3.4 他のスライス手法との比較

静的スライス，動的スライス，DC スライスの違いを表 3 に示す．

解析手法の違いにより，一般に以下の様な性質となる．これらの性質は [2] において，実験によって実証されている．

スライスサイズ: 静的 $\geq$ DC $\geq$ 動的
実行前解析時間: 静的 $>$ DC $>$ 動的
実行時間: 動的 $>$ DC $>$ 静的

また，図 3 との比較のため，図 1 のプログラムを対象に，スライス基準 $(23, d)$ に関する DC スライスを図 6 に示す．入力には，動的スライスと同じく 2 を与えた．

4 動的データ依存関係解析を用いた Java スライス手法

現在のソフトウェア開発環境において，C などの手続き型言語だけでなく，Java, C++等いわゆるオブジェクト指向言語の利用が高まっている．特に近年利用が高まっている Java を対象にしたスライス抽出を考える．

表 3: スライス手法の違い

	静的	動的	DC
制御依存関係解析	静的	動的	静的
データ依存関係解析	静的	動的	動的
PDG 節点	文	実行系列	文

```
 1: #include <stdio.h>
 2: #define SIZE 5

 3: int cube(int x) {
 4:     return x*x*x;
 5: }

 6: void main(void)
 7: {
 8:     int a[SIZE];
 9:     int b[SIZE];
10:     int c, d, i;

11:     a[0] = 0;
12:     a[1] = -1;
13:     a[2] = 2;
14:     a[3] = -3;
15:     a[4] = 4;

16:     for (i=0; i<SIZE; i++) {
17:         b[i] = a[i];
18:     }

19:     scanf("%d", &c);
20:     d = cube(b[c]);
21:     if (b[c] < 0)

23:     printf("%d", d);
24: }
```

図 6: ソースプログラム

4.1 オブジェクト指向言語を対象としたスライス

オブジェクト指向言語には，従来の手続き型言語にはないクラス，継承，ポリモーフィズムなどの新しい概念が導入されており，既存のプログラムスライス計算手法をそのまま用いても正確なスライスが得られない．そこで，オブジェクト指向言語への対応を試みた静的スライス，動的スライスが提案されている．

静的スライスでは，参照型の指すインスタンスの型推測を行う必要がある．point-to グラフを使った手法などにより，推測の範囲を限定できるが，実行時に参照されるインスタンスを一意に決定することは困難であり，静的な解析には正確性に限界がある．

一方，動的スライスは実行時に決定される参照型の指すインスタンスを容易に調べられるため，正確な解析が行える．また，Java においてオブジェクトは全て参照により操作されるため，オブジェクトへの参照を正確に解析することは重要である．

正確性の観点から見た場合，動的スライスの方が静的スライスより優れている．しかし，動的スライスは手続き型言語を対象とした動的スライス同様に実行系列を保存するため，解析に多大なコストがかかると考えられる．そこで，提案手法では，DC スライスの Java への適用を考える．DC スライスはデータ依存関係解析を動的に行うために，オブジェクトへの参照等の実行時に決定される要素を多く含む Java の解析に向いていると言える．また，実行系列を保存しないこと，及び制御依存関係解析を静的に行うことで解析コストを抑えることが期待できる．

4.2 提案手法

DC スライスは配列の添字やポインタの解析を正確に行うためにデータ依存関係解析を動的に，解析コスト削減のために制御依存関係解析を静的に行う．Java では，配列や参照が存在するために，これらの正確な解析には動的なデータ依存関係解析が有効である．

また，Java には以下の重要な特徴がある．

- オブジェクトとは，データとそれを操作するコード (メソッド) の組み合わせである．
- カプセル化によりデータへのアクセスはメソッドを通して行う．
- Java の動的束縛では，実行中のオブジェクトのクラスに基づいて適切なオーバーライドメソッドを選択する．

これより，実行時に決定されるメソッド呼び出しが頻繁に起こりうることがわかる．そこで，従来の DC スライスは，全ての制御依存関係解析を静的に行っているが，本手法では，データ依存関係解析に加えメソッド呼び出しに関する制御依存関係解析も動的に行うことで呼び出されるメソッドを一意に決定し，解析の正確性を高める．

4.2.1 アルゴリズム

図 7 に計算手順を示す．

(1), (2) で静的な制御依存関係解析を行い，PDG を構築する．(3) ではデータ依存関係解析とメソッド呼び出しに関する制御依存関係の動的な解析を行い，PDG を完成させる．動的データ依存関係解析の計算手法は 4.2.2 で示す．(4), (5), (6) では，スライス基準に対応する接点から PDG をたどり，スライスを抽出する．

入力
P: スライス対象 Java プログラム
I: P への入力
(s_c, v_c): スライス基準
一時記憶
PDG_D: P の I についての実行時の PDG
N: 節点の集合
C: 節点
出力
OUT: P の I についての実行の出力
S: P の I についての実行時, (s_c, v_c) に関するスライス
アルゴリズム
(1) P の全ての文または条件節 s について，PDG_D の節点 $V(s)$ を作成する
(2) P の全ての条件文 s について，その条件節を cnd，その中身の文を stm とすると，制御依存辺 $V(cnd) \rightharpoonup V(stm)$ を PDG_D に加える
(3) I についての P の実行が終了するまで以下を繰り返す (I について実行する次の文を s とする)
(a) s について，動的データ依存関係解析を行う
(b) s がメソッド宣言文ならば
(i) $V = C$
(ii) 制御依存辺 $V \rightharpoonup V(s)$ を PDG_D に加える
(c) s がメソッドを呼び出すならば
$C = V(s)$
(d) I について，s を実行する
(4) $S = \{V(s_c)\}, N = \phi$
(5) $N = \{$ 節点 $n | n \xrightarrow{v} s_c \} \cup \{$ 節点 $m | m \rightharpoonup s_c \}$
(6) $N \neq \phi$ の間，以下を繰り返す
(a) 節点 $n \in N$ を 1 つ選ぶ
(b) $S = S \cup n$
(c) $N = N \cup \{m | m \notin S \wedge (m \xrightarrow{w} n \vee m \rightharpoonup n)\}$
ただし，w は任意の変数名

図 7: Java スライシングアルゴリズム

4.2.2 動的データ依存関係解析アルゴリズム

制御依存関係，データ依存関係の動的な解析の計算手順を図 8 に示す．

各変数に対応するキャッシュはその変数が生成されるときに逐次生成する．同じクラスから生成された複数のインスタンスはそれぞれ独立にインスタンス変数を保持する．同様に，各インスタンス変数のキャッシュは各インスタンス変数が生成される際に生成され，それらは独立して保持される．つまり，インスタンスごとに独立した解析を行う．

4.3 適用例

図 9 のプログラムを入力 -1 について実行した場合の各実行時点におけるキャッシュ C の推移を表 4 に表す．ただし，クラス $Base$，クラス $Derived$ 内のローカル変数 i については省略した．

また，スライス基準 $(13, c)$ に関するスライスを図 10 に示す．

入力

s: スライス対象 Java プログラム P 中の文
I: P への入力
PDG_D: P の I についての実行時の PDG

一時記憶

$C(v)$: 変数 v のキャッシュ

出力

OUT: P の I についての実行の出力
PDG_D: P の I についての実行時の PDG

アルゴリズム

文 s において，変数 v が
(1) 宣言された場合
  (a) $C(v)$ を作成
  (b) $C(v) = V(s)$
(2) 定義された場合
  $C(v) = V(s)$
(3) 参照された場合
  v に関するデータ依存辺 $C(v) \xrightarrow{v} V(s)$ を PDG_D に加える

図 8: 動的データ依存関係解析アルゴリズム

表 4: 図 9 のプログラムにおける，各実行時点でのキャッシュの推移

実行文	$b1$	$b2$	c	i	$b1.a$	$b2.a$
1	-	-	-	-	-	-
2	2	-	-	-	-	-
3	2	-	-	-	-	-
4	2	-	4	-	-	-
5	2	5	4	-	-	-
31	2	5	4	-	-	-
21	2	5	4	-	-	-
22	2	5	4	-	-	22
6	2	5	4	6	-	22
7	2	5	4	6	-	22
8	8	5	4	6	-	22
21	8	5	4	6	-	22
22	8	5	4	6	22	22
11	8	5	11	6	22	22
23	8	5	11	6	22	22
24	8	5	11	6	22	22
12	8	5	11	6	22	22
26	8	5	11	6	22	22
27	8	5	11	6	22	27
13	8	5	11	6	22	27
14	8	5	11	6	22	27

4.4 実装

提案手法の実装方針としてインタプリタ方式とプリプロセッサ方式が考えられる．今回は，実現容易性，実行速度で有利なプリプロセッサ方式の実装を選択した．

4.4.1 プリプロセッサ方式

プリプロセッサ方式は，Java のソースプログラムをコンパイルする前に，そのプログラム自身の動的解析を行う命令を付加するプリプロセッサによる実装である．解析命令を付加したプログラムをコンパイル，実行することで，通常の実行を行いながら解析も自動的に行われる．実行に関しては，通常の実行と同様に行えるため，Just-in-Time コンパイラ等による高速実行が可能である．

```
 1  class Main {
 2    Base b1;
 3    void m() {
 4      int c;
 5      Base b2 = new Derived();
 6      int i = read();
 7      if (i < 0)
 8        b1 = new Base();
 9      else
10        b1 = new Derived();
11      c = b1.m(i);
12      b2.set(c)
13      System.out.println(c);
14      System.out.println(b1.a);
15    }
16  }

21  class Base {
22    int a = 10;
23    int m(int i) {
24      return (a-i);
25    }
26    void set(int i) {
27      a = i;
28    }
29  }

31  class Derived extends Base {
32    int m(int i) {
33      return (a+i);
34    }
35  }
```

図 9: プログラム

4.5 実装概要

現在，Java によるプリプロセッサとユーザインタフェースの作成を行っている．

実装概要を図 11 に示す．

4.5.1 キャッシュの実装

各変数のキャッシュを以下のように実装する．

- 各クラスに現れるインスタンス変数 v に対してキャッシュ $C(v)$ をそのクラスのインスタンス変数として用意する．
- 各クラスに現れる静的変数 v に対してキャッシュ $C(v)$ をそのクラスの静的変数として用意する．
- 各クラスに現れるローカル変数 v に対してキャッシュ $C(v)$ を v が存在するスコープ内のローカル変数として用意する．

これにより，あるクラスの複数のインスタンスが生成された際に，それぞれのインスタンスが各インスタンス変数 v を独立して保持するのと同様に，それぞれのインスタンスが各キャッシュ $C(v)$ を独立に保持する．つまり，それぞれのインスタンスについて独立した解析を行うことができる．また，変数 v が継承される場合，同様にキャッシュ $C(v)$

```
 1  class Main {
 2    Base b1;
 3    void m() {
 4      int c;
 5
 6      int i = read();
 7      if (i < 0)
 8        b1 = new Base();
 9
10
11      c = b1.m(i);
12
13      System.out.println(c);
14
15    }
16  }

21  class Base {
22    int a = 10;
23    int m(int i) {
24      return (a-i);
25    }
26
27
28
29  }

31
32
33
34
35
```

図 10: 図 9 のプログラムのスライス基準 $(14, c)$ に関するスライス

も継承される.

5 まとめ

本研究では，オブジェクト指向言語である Java のプログラムに対して，動的・静的解析を用いてスライスを抽出する手法を提案した．提案手法は，動的な解析により実行時に決定される要素の解析を行い，正確性の高いスライスを抽出する．

今後の課題として，以下が挙げられる．

- 実装の完成
- 他のスライス手法との比較実験による評価
- 並列処理への対応
- 例外への対応

Java では並列処理や例外も動的に決定される要素を含むため，これらに関する依存関係解析を動的に行うことで，正確な解析が行えると考えている．

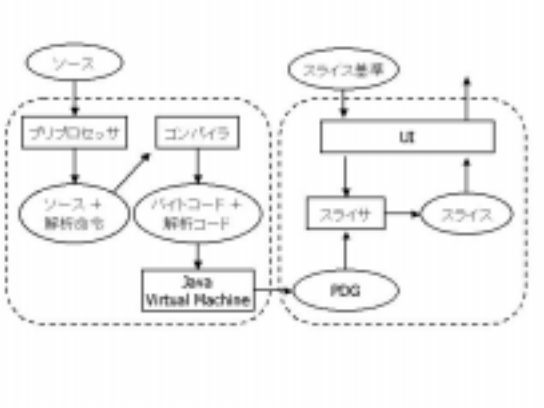

図 11: 実装概要

参考文献

[1] Agrawal, H., and Horgan, J. : "Dynamic Program Slicing", *SIGPLAN Notices*, Vol.25, No.6, pp. 246–256, 1990.

[2] Ashida Y., Ohata F., and Inoue K. : "Slicing Methods Using Static and Dynamic Information", *Proceedings of the 6th Asia Pacific Software Engineering Conference (APSEC'99)*, pp. 344–350, December 1999.

[3] Larsen L. D. and Harrold M. J. : "Slicing Object-Oriented Software", *Proceedings of the 18th International Conference on Software Engineering*, pp. 495–505, Berlin, March 1996.

[4] Liang, D., and Harrold, M. J. : "Slicing Objects Using System Dependence Graphs", *Proceedings of the IEEE International Conference on Software Maintenance*, Washington, D.C., November 1998.

[5] Steensgaard, B. : "Points-to analysis in almost linear time", *Technical Report MSR-TR-95-08*, Microsoft Research, 1995

[6] Weiser, M. : "Program Slicing", *Proceedings of the Fifth International Conference on Software Engineering*, pp. 439–449, 1981.

[7] Zhao, J. : "Dynamic Slicing of Object-Oriented Programs", *Technical Report SE-98-119, Information Processing Society of Japan (IPSJ)*, pp. 11–23, May 1998.