Javaバーチャルマシンを利用した 動的依存関係解析手法の提案

誉田 謙二　梅森 文彰　大畑 文明　井上 克郎

大阪大学大学院基礎工学研究科
〒 560-8531 大阪府豊中市待兼山町 1-3
Phone: 06-6850-6571　Fax: 06-6850-6574
E-mail: {konda, umemori, 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手法の一つに，スライスがある．スライスは，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，それぞれ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に分類できる．
本研究では，DC スライスを Java へ適用する手法を提案する．Java は実行時決定要素を多数含むため，静的制御依存関係解析と動的データ依存関係解析を組み合わせた，DC スライスが有効である．
提案手法では，Java の実行時決定要素を JVM を用いて考慮し，バイトコードに対して DC スライスを計算する．そして，計算されたスライスをソースコードに対応付ける．

キーワード プログラムスライス，静的解析，動的解析，Java Virtual Machine

Dynamic Dependence Analysis Method using Java Virtual Machine

Kenji Konda, Fumiaki Umemori, Fumiaki Ohata and Katsuro Inoue

Graduate School of Engineering Science, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
Phone: 06-6850-6571　Fax: 06-6850-6574
E-mail: {konda, umemori, oohata, inoue}@ics.es.osaka-u.ac.jp

Abstract
Program slicing has been used for efficient program debugging activities. Program slice is computed by analyzing data dependency and control dependency relations between program statements. We would categorize dependency analyses in two, static and dynamic; the former requires less analysis cost, but it produces weaker result. The latter has opposite characteristics.
In this paper, we propose a method for applying DC Slice to Java programs. Since Java has many dynamically determined elements, DC Slice, which is based on static control dependence analysis and dynamic data dependence analysis, would be more practical than other slicing methods.
Our method computes DC Slice for programs in Java bytecode using JVM, on which dynamically determined elements are considered, and then, translates the resulting slice on Java bytecode into the slice on Java sourecode.

Key words Program Slice, Static Analysis, Dynamic Analysis, Java Virtual Machine

1 まえがき

プログラム理解やデバッグ（保守）を効率よく行う手法の一つに，**プログラムスライス**（*Program Slice*，以下，スライス）がある．**プログラムスライシング**（*Program Slicing*）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**（*Slicing Criterion*）<s, v> と呼ぶ）に対して v の値に影響を与えるすべての文をプログラムから抽出する技法で，その結果取り出された文の集合がスライスとなる．一般に，スライスはプログラム文間の**依存関係**（*Dependence Relation*）の解析により得られる．今まで多くのスライス計算技法が提案されてきた．主なものには，**静的スライス**（*Static Slice*）[10]，**動的スライス**（*Dynamic Slice*）[4]，**DC スライス**（*DC Slice*）[12][7] がある．

近年のソフトウェア開発環境において，オブジェクト指向言語の利用が高まっており，クラスや継承など，オブジェクト指向独特の概念を考慮した静的スライス [9]，動的スライス [6]，DC スライス [3] が提案されてきた．オブジェクト指向言語には実行時決定要素が多く含まれており，起こり得るすべての可能性を考慮する静的スライスは精度の点で問題がある．一方，動的スライスは実行時決定要素を容易に扱うことができ，精度の高いスライスを計算できる．しかし，動的スライスは，全ての**実行系列**（*Execution Trace*）を保存する必要があるため，多大な時間，空間コストを要する．DC スライスは，一部の実行時決定要素を扱うため静的スライスより精度の高いスライスを計算できる．さらに，全実行系列を保存する必要がないため，解析コストは動的スライスより大幅に小さくなる．そのため，オブジェクト指向プログラムにおけるスライス計算に有効であると考えている．

本研究では，オブジェクト指向言語 Java で記述されたプログラムに対する，**Java バーチャルマシン**（*Java Virtual Machine*，以下，JVM）に基づく DC スライス計算手法を提案する．提案手法では，バイトコードに対して DC スライスを計算し，コンパイラによって生成されたバイトコードとソースコードと対応表を利用し，スライス抽出結果をソースコードに対応付ける．JVM によるバイトコード単位での依存関係の解析に基づいていることから，細粒度の DC スライス計算が可能となる．

以降，2. でプログラムスライスについて説明する．3. で DC スライスについて述べ，4. で Java に対する JVM を用いた DC スライスの計算手法について述べる．さらに 5. で提案手法の有効性を関連研究と比較しながら述べ，最後に 6. 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

2.1 プログラムスライス計算手順

スライス計算にはさまざまな手法が存在するが，本稿ではプログラム依存グラフによるスライス計算手法を用いる [8]．はじめにプログラム文間に存在する 2 つの依存関係について説明したのち，手法 [8] によるスライス計算手順について簡単に述べる．

制御依存関係

プログラム中の 2 文 s，t に関して，以下の条件を満たすとき，s から t の間に**制御依存関係**（*Control Dependence*，CD）が存在するという．

(1) s は条件文である
(2) t の実行は s の判定結果に依存する

データ依存関係

プログラム中の 2 文 s，t に関して，以下の条件を満たすとき，s から t の間に変数 v に関する**データ依存関係**（*Data Dependence*，DD）が存在するという．

(1) s は v を定義する
(2) t は v を参照する
(3) s における変数 v の定義が文 t に到達する

スライス計算法

Phase 1: 依存関係解析

各プログラム文に対し，
(a) **制御依存関係解析**
(b) **データ依存関係解析**
を行う．

Phase 2: プログラム依存グラフ構築

Phase 1 で求めた依存関係を利用し，**プログラム依存グラフ**（*Program Dependence Graph*，以下，PDG）を構築する．PDG の節点は各プログラム文，各辺は文間の依存関係（制御依存関係，データ依存関係）を表す．図 1 にサンプルプログラムおよびその PDG を示す．

Phase 3: スライス計算

スライス基準 <s, v> に対するスライスを計算する．s に対応した PDG の節点 V_s から，逆方向に制御依存辺およびデータ依存辺を経て推移的に到達可能な節点集合に対応する文の集合を計算する．図 1 において，スライス基準 < 文 5, b> に対するスライスは { 文 1, 文 2, 文 3, 文 5} となる．

上述の Phase 1 の依存関係解析の方針により，静的スライスと動的スライスの 2 つに大別することができる．以降，それぞれについて述べる．

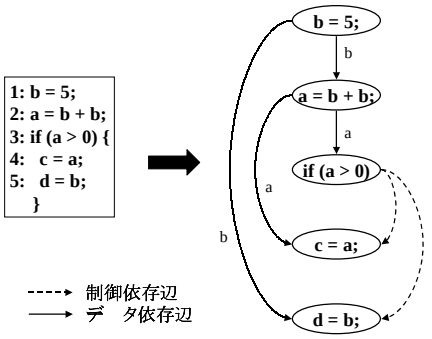

図 1: サンプルプログラム および その PDG

2.2 静的スライス

静的スライスは，Phase 1 において (a) 制御依存関係解析，(b) データ依存関係解析をともに静的に行うスライス計算手法である．与えられたソースプログラムの各文を節点とし，起こり得るすべての実行経路に対して依存関係解析を行い，図 1 のような PDG を構築し，スライスを計算する．

静的スライスは，現実には短い時間で計算される．静的スライスは，プログラムに起こり得るすべての実行経路を考慮して PDG を構築するため，プログラムに存在する特定の機能を抽出したい場合には有効である．しかし，プログラム実行においてすべての実行経路が利用されることは少なく，実行時エラーの原因を把握するためのフォールト位置特定に対しては効果的とはいえない．

2.3 動的スライス

動的スライスは，Phase 1 において (a) 制御依存関係解析，(b) データ依存関係解析をともに動的に行うスライス計算手法である．まず，特定の入力を与えてプログラムを実行する．そして，得られた実行系列の各**実行時点**（*Execution Point*）を節点とし，特定の実行経路に対する依存関係解析による PDG を構築し，スライスを計算する．

動的スライスでは，解析対象を特定の実行経路に限定し，その実行の際に発生する依存関係に基づくものであるため，一般に計算されるスライスは静的スライスに比べて小さくなる．また，実際に実行された部分の中からのみスライスが計算されるため，フォールト位置特定を効率よく行うことができる．しかし，動的スライスの計算には，実行系列および，実行中に発生するすべての制御依存関係とデータ依存関係を記憶しなければならないため，多大な空間コストと時間コストを要する．とりわけ，実行系列の大きさはプログラム実行文の数に比例することから，入力データによっては非常に大きなものとなり，それに伴いスライス計算に要する時間も増大する．

3 DC（Dependence-Cache）スライス

例えば，配列を含むプログラムに対して静的スライスを計算する場合，配列の添字の値を静的に把握することは難しく，可能性のある添字値をすべて考慮しなければならないため不要に多くのデータ依存関係を生成してしまうことがある．また，ポインタを介した**エイリアス**（*Alias*）などによる陽に現れないデータ依存関係を考慮しなければならないため，静的なデータ依存関係解析では限界がある．

DC スライスは，Phase 1 における (b) データ依存関係解析を動的に行う．これにより，配列の添字やポインタの参照先などの実行時決定要素を正確に把握することができる．一方，(a) 制御依存関係解析については静的に行うため，実行系列を保存する必要がなく，動的スライスに比べ解析コストを抑えることができる．

3.1 DC スライス計算手順

DC スライスの計算手順は，以下の 3 つの段階から構成される．

Phase 1: 依存関係解析

各プログラム文に対し，
(a) 静的制御依存関係解析
(b) 動的データ依存関係解析
を行う．(b) は動的スライスのものとは異なり，実行系列の保存は行わない．詳細は 3.2. で述べる．

Phase 2: PDG 構築

Phase 1 で求めた依存関係を利用し，PDG を構築する．ここで構築される PDG の各節点は，動的スライスとは異なり，プログラム文に対応する．

Phase 3: スライス計算

指定されたスライス基準に対応する節点からグラフ探索を行い，スライスを計算する．

3.2 動的データ依存関係解析

プログラムの実行時に軽量な手間でデータ依存関係を抽出することを考える．プログラム中のある文 s においてある変数 v が参照されるとき，v を定義した文 t が分かれば，s から t の間に，v に関するデータ依存関係が存在することが把握できる．つまり，各変数 v について，その変数がどこで定義されたかを保存しながらプログラムを実行すれば，動的なデータ依存関係解析を実現することができる．

そこで，プログラム中で用いられるすべての変数 v に対し**キャッシュ**（*Cache*）$C(v)$ を用意する．$C(v)$ には変数 v が最後に定義された文番号が格納されており，文 t の実行時に変数 v に対するアクセスがあった場合，次のような処理が行われる．

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

図 2: 配列を含むプログラム

表 1: 図 2 におけるキャッシュの推移

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

文 t で v が定義された場合

$C(v)$ の値を t の文番号に更新する．

文 t で v が参照された場合

v に関する，$C(v)$ に対応する節点から t に対応する節点に発生するデータ依存関係を抽出する．

例として，図 2 のような配列を含むプログラムに対して動的データ依存関係解析を行う場合を考える．入力として変数 c に 0 を与えて実行させたときの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 に示す．

文 1 から文 4 では，それぞれ変数 $a[0]$，$a[1]$，$a[2]$，c が定義されているため，文 4 の実行が終了した時点で $C(a[0]) = 1$，$C(a[1]) = 2$，$C(a[2]) = 3$，$C(c) = 4$ となる．文 5 で変数 $a[0]$ が参照されるため，文 5 の実行時に文 $C(a[0])$，つまり文 1 から文 5 に $a[0]$ に関するデータ依存関係が発生することになる．

3.3 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 スライスの計算手法の違いを表 2 に示す．

また，計算手法の違いにより，解析精度（スライスサイズ），解析コスト（依存関係解析時間）に関し以下のような特性を持つ [12][11]．

表 2: 各スライス手法の違い

	静的スライス	DC スライス	動的スライス
CD 解析	静的	静的	動的
DD 解析	静的	動的	動的
PDG 節点	プログラム文	プログラム文	実行時点

解析精度（スライスサイズ）

静的スライス $\geq$ DC スライス $\geq$ 動的スライス

解析コスト（依存関係解析時間）

動的スライス $\gg$ DC スライス $>$ 静的スライス

具体的なスライス結果の例を，図 3 に示す．これは，解析対象プログラムのスライス基準 <9 , b> に対するスライスである．なお，DC スライスおよび動的スライスにおいては，入力として変数 c に 2 を与えた実行を想定している．

これらのことから，DC スライスは，動的スライスより小さい解析コストで，静的スライスより高い精度のスライスを計算できる手法といえる．

4 JVM に基づく DC スライスの Java への適用

オブジェクト指向言語である Java において，データ格納の基本単位であるオブジェクト（または，クラスのインスタンス）は，データおよびそれを扱うメソッドの組み合わせである．そのため，Java プログラムに対してスライス計算を行う場合，実行メソッドを特定するためにインスタンスの型を把握しておかねばならない．

しかし，静的スライスはプログラム実行を伴わないため，インスタンスの型を静的に推測する必要がある．既存の手法 [2] を用いることでいくつかの型に限定することはできるが一意に決定することは難しく，実行メソッドの特定は困難である．そのため，静的スライスでは精度の面で限界がある．

一方，動的スライスはプログラム実行を伴うため，実行時にインスタンスの型を容易に把握することが

```
プログラム                静的スライス              DC スライス               動的スライス
1:  a[1] = 1;             1:  a[1] = 1;             1:  a[1] = 1;             1:  a[1] = 1;
2:  a[2] = 2;             2:  a[2] = 2;             2:  a[2] = 2;             2:  a[2] = 2;
3:  a[3] = 3;             3:  a[3] = 3;             3:  a[3] = 3;             3:  a[3] = 3;
4:  a[4] = 4;             4:  a[4] = 4;             4:  a[4] = 4;             4:  a[4] = 4;
5:  read(c);              5:  read(c);              5:  read(c);              5:  read(c);
6:  while (c > 0) {       6:  while (c > 0) {       6:  while (c > 0) {       6:  while (c > 0) {
7:    b = a[c] + 1;       7:    b = a[c] + 1;       7:    b = a[c] + 1;       7:    b = a[c] + 1;
8:    c = c - 1;          8:    c = c - 1;          8:    c = c - 1;          8:    c = c - 1;
 :  }                      :  }                      :  }                      :  }
9:  write(b);             9:  write(b);             9:  write(b);             9:  write(b);
```

図 3: スライス手法の比較

できる．そのため，実行メソッドも特定できることから，精度の高いスライス計算が可能である．

このように，Java に対するスライス計算を考える際には，精度の点で動的スライスは静的スライスより優れている．しかし，動的スライスは実行系列の保存を必要とし，解析コストは膨大なものとなる．そこで本研究では，Java に対し DC スライスを適用する．DC スライスでは，インスタンスの型などの実行時決定要素を正確に把握でき，高い精度のスライス計算が可能となる．さらに，実行系列の保存を必要とせず，動的スライスに比べ解析コストを大幅に抑えることができる．

4.1 方針

Java における実行時決定要素としては，以下のようなものがある．

- 配列の添字
- オブジェクトの参照変数
- メソッドの動的束縛
- 例外処理
- 並列処理

DC スライスの計算に必要となる動的データ依存関係解析の実現には，これらを正確に把握するための環境が求められる．そこで，Java プログラムのコンパイル結果として生成される**バイトコード**を解釈，実行する機構である JVM を利用した動的データ依存関係解析の実現を考える．JVM 上でバイトコードが実行される際には，上記の実行時決定要素はすべて把握可能で，バイトコードに対する動的データ依存関係解析を行うことができる．

また一般に，Java で記述されたソースコードの各文は，コンパイラにより複数のバイトコード命令に変換される．そのため，ソースコードの文単位で依存関係解析を行うのではなく，バイトコードの命令単位で依存関係解析を行うことで，細粒度の解析の実現が期待できる．

本研究では，本来ソースコード対して提案されてきた DC スライスをバイトコードに適用する．具体的には，バイトコードの各命令を節点とする PDG を構築し，バイトコード上での DC スライスの計算を行う．そして，得られた DC スライスをソースコードへ反映させる．

ソースコードとバイトコードの対応付けは，コンパイラの機能拡張により実現する．通常，Java コンパイラは Java で記述されたソースコードをバイトコードに変換することが主な役割であるが，本手法ではさらに，図 4 に示す，バイトコードの各命令とソースコードのトークン集合との対応表の生成を Java コンパイラにより行う．この対応表を用いることで，

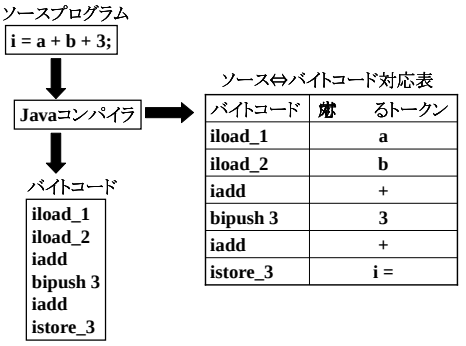

図 4: バイトコードへの変換例

- ユーザにより指定されたソースコード上でのスライス基準と，それに対応するバイトコードとの対応付け
- バイトコード上で計算された DC スライスと，それに対応するソースコードとの対応付け

を行うことができる．

以降，DC スライスのバイトコードへの適用について述べたあと，提案手法の実現について述べる．

4.2 DC スライスのバイトコードへの適用

ここでは，バイトコードに対する DC スライスの計算手順について述べる．

Phase 1: 依存関係解析
各バイトコード命令に対し，
(a) 静的制御依存解析
(b) 動的データ依存解析
を行う．(a) については 4.2.1. で，(b) については 4.2.2. で詳しく述べる．

Phase 2: PDG 構築
Phase 1 で求めた依存関係を利用し，バイトコードの各命令を節点とした PDG を構築する．

Phase 3: スライス抽出
指定されたスライス基準に対応する節点からグラフ探索を行い，バイトコードでのスライスを計算する．

以降，各 Phase での処理について詳細を述べる．

4.2.1 静的制御依存関係解析

Phase 1(a) では，与えられたバイトコードに対し，制御依存関係解析を静的に行う．その際，ソースコードにおける条件節とその述部という概念に基づく制御依存関係の定義をそのままバイトコードに適用することは困難であるため，本研究では，バイトコードでの制御依存関係を次のように定義する．

バイトコードの 2 命令 s，t に関して，以下の条件を満たすとき，s から t の間に制御依存関係が存在するという．

入力 バイトコード
出力 命令間に存在する制御依存関係
処理 バイトコードにおける静的制御依存関係を抽出する

(1) バイトコードの命令列を基本ブロックに分割する
($\mathcal{N}$: 基本ブロックの集合)
(2) **foreach** n **in** $\mathcal{N}$ **begin**
(3) **if** n の最後の命令が分岐命令なら **then**
(4) n から制御が移動する基本ブロック集合 $\mathcal{N}'$ を導出
(5) **foreach** n' **in** $\mathcal{N}'$
(6) n の最後の命令と n' の各命令の対を
制御依存関係として抽出する
(7) **end**

図 5: 静的制御依存関係解析アルゴリズム

(1) s は分岐命令である
(2) t は s から直接到達する**基本ブロック**（*Basic Block*）[1] 内の命令である

このように定義した制御依存関係は，図 5 に示すアルゴリズムにより抽出できる．なお．このアルゴリズムはメソッド単位に適用する．

4.2.2 動的データ依存関係解析

Phase 1(b) では，JVM 上でバイトコードを実行し，それと並行して動的データ依存関係解析を行う．ソースコードに対する DC スライスの計算においては，各変数に対してキャッシュを用意した．バイトコードにおいてもそれと同様に，メソッド内のローカル変数やインスタンスのメンバ変数などのデータ領域それぞれに対してキャッシュを用意する．ただし，JVM にはスタックを介した演算が存在するため，スタックをデータ領域とみなし，それに対するキャッシュも用意する．

そして，各データに関して，値が参照された場合にはキャッシュに保存されている命令と実行中の命令間に発生したデータ依存関係を抽出し，値が定義された場合にはキャッシュの内容を更新する．なお，あるデータの値が定義されたとき，それに対応するキャッシュが存在しない場合には新たにキャッシュを生成する．

以上の方針に基づき定義された，動的データ依存関係解析アルゴリズムを図 6 に示す．このアルゴリズムは，JVM 上でバイトコードの一命令が実行されるたびに適用される．本手法では，同一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キャッシュを保持しており，各インスタンス独立にデータ依存関係解析が行われることになる．

4.2.3 PDG の構築

静的制御依存関係解析，動的データ依存関係解析により抽出された依存関係を用いて PDG を構築する．構築される PDG の例を図 7 に示す．PDG の節点はバイトコードの各命令であるため，実行系列を

入力 バイトコードの命令 s
出力 s の実行により発生するデータ依存関係
処理 バイトコードにおける動的データ依存関係を抽出する

(1) **foreach** n **in** s で参照される変数
(2) n のキャッシュに保持されている命令と s の対を
データ依存関係として抽出する
(3) **foreach** n **in** s で定義される変数 **begin**
(4) **if** n のキャッシュがなければ **then**
(5) n のキャッシュを生成する
(6) n のキャッシュを s に更新
(7) **end**

図 6: 動的データ依存関係解析アルゴリズム

保存する必要はなく，動的スライスに比べ解析コストは十分に小さい．

4.2.4 スライス計算

PDG を用いてスライス計算を行う．バイトコードに対するスライス計算も従来手法と同じで，スライス基準に対応する PDG 節点から PDG 辺を逆に辿り，到達可能な節点集合を求めることによる．

4.3 実装

提案手法の実装システムの構成を図 8 に示す．

ソースコードのコンパイル時にバイトコードとソースコードの対応表を生成する．そして，静的に制御依存関係解析を行ったのち，JVM 上でバイトコードの実行を行いながら動的にデータ依存関係解析を行う．これらにより抽出された依存関係を元にバイトコードの各命令が節点となる PDG を構築する．ユーザによりソースコードにおけるスライス基準が指定されると，対応表を用いてそれをバイトコードにおけるスライス基準に変換し，PDG 探索よるスライス計算を行う．最後に，対応表を参照しながらスライス結果をソースコードに対応付ける．

現在，ソースコードとバイトコードの対応表を出力するためのコンパイラおよび，動的データ依存関係解析を行う JVM の実装を行っている．

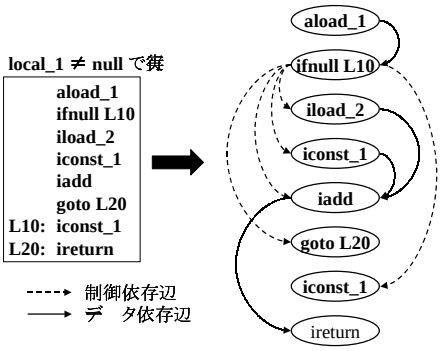

図 7: バイトコードでのプログラム依存グラフ

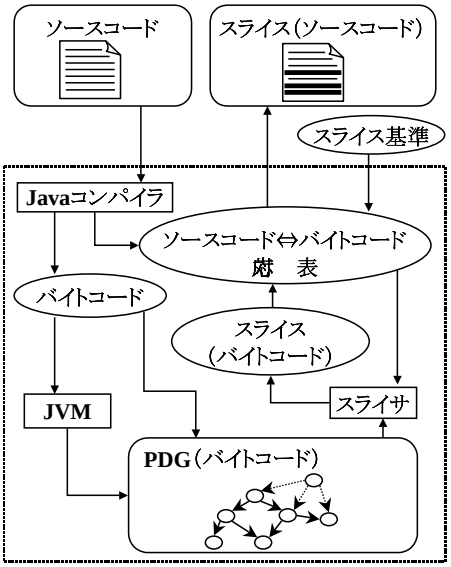

図 8: システム構成

4.3.1 コンパイラ

コンパイラは，JDK（Java Development Kit）付属の Java コンパイラ `javac` に対する機能拡張の形で実装を行っている．

ソースコードからバイトコードへの変換を行いながら，このコード変換規則に基づくバイトコード一命令とソースコード数トークンとの対応表を生成する．図 4 がその出力例である．

4.3.2 JVM

JVM は，コンパイラと同様，JDK 付属の Java バーチャルマシン `java` に対する機能拡張の形で実装を行っている．

スタック，ローカル変数など，データを格納する各領域に対応するキャッシュを確保する．また，インスタンスが新しく生成されるときには，インスタンス本体だけでなく，そのインスタンスが保持する各メンバ変数に対応するキャッシュも確保する．このように，インスタンス本体にキャッシュが付随する形式での実装を行うことで，インスタンスが独立にキャッシュを保持できるため，インスタンス独立にデータ依存関係解析が行われることになる．

5 考察

5.1 関連研究

Java プログラムに対する動的解析に関して，本研究以外にもいくつか研究がなされている．ここでは，その中の 2 つの関連研究を挙げる．

5.1.1 バイトコード埋め込み方式による動的解析

BIT（Bytecode Instrumenting Tool）[5] は，バイトコードに，それ自身を解析するためのバイトコード命令を挿入するライブラリ群である．ユーザは BIT を用いてクラスファイルの変更を行い，通常の JVM を利用することでそのバイトコードの実行時の情報を取得することができる．

しかし，挿入するバイトコードをユーザが記述する必要があり，ユーザは特別な知識を要求される．また，結果として得られる動的情報は，実行命令のトレースや分岐の成功比率など，統計データの計測を目的とするものが多く，依存関係のような複雑な計算を必要とする情報の抽出は困難である．

5.1.2 ソースコード埋め込み方式による動的解析

[3] は，本研究と同様，Java に対して DC スライスを計算する．[3] は，ソースコードに，それ自身を解析するためのソースコードを挿入するプリプロセッサである．プリプロセッサを経たのち，通常のコンパイルおよび実行を行うことで，実行時に発生する動的データ依存関係に基づく PDG が構築される．

ソースコードの埋め込みによる動的依存関係解析の実現を行っているため，Java の構文制約による埋め込み位置の制限が厳しく，結果として十分な解析の精度が得られない問題がある．また，ソースコードの変更を前提とするため，ソースコードの存在しないクラスを利用するプログラムに対しスライス計算を行う場合，極端に解析程度が低下してしまう．

5.2 提案手法の有効性

本手法では，Java プログラムにおける依存関係解析をバイトコードに対して行う．そして，バイトコードに対して計算された DC スライスを，対応表をもとにソースコードに反映する．構文制約の影響を受けないため，解析精度の低下を防ぐことができる．また，クラスファイルのみ存在するようなクラスを利用したプログラムにおけるスライス計算も可能となる．さらに，PDG 節点をソースコードの文単位ではなく，バイトコードの命令単位としたことにより，細粒度の解析結果を得ることもできる．

また，ユーザは DC スライスの計算にあたり，本システムについて特別な知識を必要とせず，通常のコンパイル，実行を行うだけで DC スライス計算に必要な情報を取得することができる．

6 まとめと今後の課題

本研究では，Java プログラムに対し，JVM を用いた静的制御依存関係解析および動的データ依存関係

解析によりDCスライスを計算する手法を提案した．提案手法では，JVM上でのプログラム実行時にバイトコードに対する動的データ依存関係解析を行うことで，インスタンスの型特定や配列の添字など，実行時決定要素を考慮することができる．これにより，精度の高いスライスを少ないコストで計算できる．

今後の課題としては，実装の完成，システムの評価（他のスライス手法との比較，システム性能）を挙げることができる．

参考文献

[1] A.V.Aho, R.Sethi and J.D.Ullman: "Compilers Principles, Techniques, and Tools", Addison-Wesley, (1986).

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

[3] F.Ohata, K.Hirose and K.Inoue: "A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic Information", Proceedings of Eighth Asia-Pacific Software Engineering Conference (APSEC2001), pp.273–280, Macau, China, December (2001).

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

[5] H.B.Lee and B.G.Zorn: "BIT:A Tool for Instrumenting Java Bytecodes", Proceedings of the USENIX Symposium on Internet Technologies and Systems, pp.73–83, Montray, California, December (1997).

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

[7] K.Inoue, F.Ohata and Y.Ashida: "Lightweight Semi-Dynamic Methods for Efficient and Effective Program Slicing", Technical Report of Osaka University, Department of Information and Computer Scineces, Inoue Laboratory, June (2000).

[8] K.J.Ottenstein and L.M.Ottenstein: "The program dependence graph in a software development environment", Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp.177–184, Pittsburgh, Pennsylvania, April (1984).

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

[10] M.Weiser: "Program Slicing", IEEE Transaction on Software Engineering, 10(4), pp.352–357 (1984).

[11] 高田 智規, 井上 克郎, 大畑 文明, 芦田 佳行: "制限された動的情報を用いたプログラムスライシング手法の提案", 電子情報通信学会論文誌 D-I(採録決定), September (2001).

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