制御フローを考慮しない データ依存関係解析の実験的評価

石尾 隆[†1] 井上克郎[†1]

データ依存関係解析は，様々なプログラム解析手法の基盤として用いられている．通常のデータ依存関係解析では，変数に代入された値がどこで参照されるかを制御フローグラフ上で計算するが，本研究では，制御フローを考慮せず，すべての代入結果がすべての参照に到達すると仮定してデータ依存関係を計算し，通常のデータ依存関係解析の結果と比較した．その結果，10 個の Java プログラムに含まれた約 58 万個のメソッドのうち，90.9%のメソッドについては，2 つの計算方法で同一の結果が得られることを確認した．また，正しい解析結果を得るために制御フローの考慮が必要なメソッドは，代入文が 2 つ以上ある変数を持つメソッドに限られており，その条件に該当するメソッドが 15.8%だけであることを示した．

An Empirical Evaluation of a Flow-Insensitive Data Dependence Analysis

Takashi Ishio[†1] and Katsuro Inoue[†1]

Data dependence analysis is employed for various program analysis techniques. Traditional data dependence analysis computes reaching definitions on a control-flow graph. In this research, we have compared the traditional analysis and our flow-insensitive analysis that assumes all assignments to a variable have control-flow paths to all statements that refer to the variable. As a result, we have found that flow-insensitive analysis could compute the correct result for 90.9% of 580 thousand methods in 10 Java programs. We have also shown only 15.8% of methods required flow-sensitive analysis since the methods have at least two assignment statements to a single variable.

†1 大阪大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

1. はじめに

システム依存グラフ (SDG)[5] の辺の 1 つであるデータ依存関係は，プログラムスライシング[18] や，ある特定の機能を実装したソースコード位置の特定[2]，手続き抽出リファクタリング[8]，類似ソースコードの検索[17] など，様々なプログラム解析手法の基盤として用いられている．しかし，システム依存グラフを構築する計算には時間がかかることも知られている．Jász ら[7] によると，CPU に AMD Opteron 2.2GHz，メモリを 4GB 搭載した計算機で，C 言語に対するプログラムスライシングツールである CodeSurfer を実行した場合，14KLOC の valgrind に対する SDG の構築には 21 分が必要であり，30KLOC の gdb に対する SDG の構築には 132 分必要であった．一方で，開発者が 1 日に行うプログラミングなどの作業は，30 分の短いタスクが数回，2 時間の長いタスクが 1 回程度であると報告されており[14]，開発者の日常的な作業にデータ依存関係解析を活用することは困難な状況にある．

著者らの研究グループでは，データ依存関係を高速に抽出するために，制御フロー解析を行わず，変数のすべての代入からすべての参照に，直接，データ依存関係を接続する解析手法を提案した[4),15]．この近似計算は，10 万行程度のプログラムに対して約 5 分で解析が完了する[15] 一方で，従来のデータ依存関係の解析では出現しない不適切なパス（infeasible paths）を抽出することになる．これに対して，悦田の実験[4] では，制御フロー解析なしで近似計算したデータ依存関係あっても，開発者が行うプログラムの調査を支援可能であることを示した．しかし，この実験は，JEdit という 1 つのソフトウェアを対象としたプログラム理解の実施を行ったものであり，他のソフトウェアにも同様の手法が効果的であるかどうかは示していない．

本研究では，この観測結果が一般化できるかどうかを評価するために，従来から使用されているデータ依存関係と，近似計算したデータ依存関係を，10 個の Java プログラムに対して比較する実験を行った．その結果，全プログラムの 90.9%のメソッドについては，制御フローを考慮せず，すべての代入からすべての参照にデータ依存関係があると考えても，正しいデータ依存関係が得られることを示した．正しい結果が得られるメソッドの割合はプログラムごとに大きな違いはなく，制御フローを考慮しないデータ依存関係が，一般的な多くの Java プログラムに適用可能であることが期待できる．また，データ依存関係を正しく得られないメソッドの集合は，1 つの変数に対する 2 回以上の代入を持つメソッドの一部であり，そのようなメソッドは平均で 1 プログラムの 15.8%，最大でも 20.9%であった．

```
    public File getFile() {
1:    String filename = getFileName();
2:    if (filename == null) {
3:      filename = getDefaultFileName();
4:    }
5:    File f = new File(filename);
6:    return f;
    }
```

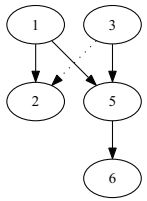

図 1　データ依存関係の解析例．左のコード片におけるデータ依存関係が右のグラフの実線に対応する．
Fig. 1　An example of data dependence. The solid edges in the right graph shows data dependence of the left code.

本研究の貢献は，開発者が日常的な開発作業に使用できるプログラム解析技法の 1 つとして，制御フローを考慮しない，変数の代入および参照関係を用いた簡便なデータ依存関係が活用できる可能性を示したことである．

以降，2 節では，研究の背景であるデータ依存関係の定義について述べる．3 節ではデータ依存関係の調査方法とその結果について述べる．4 節は結果の妥当性および影響に関する議論を行う．5 節では関連研究を紹介し，6 節でまとめを述べる．

2. 背　　景

2.1　データ依存関係

システム依存グラフ[5] を用いたプログラムスライシングでは，次のようにデータ依存関係を定義している．

プログラム文 s_1 と s_2 の間に，以下の条件が成り立ったとき，s_2 は s_1 にデータ依存するといい，s_1 から s_2 にデータ依存辺を引く．

(1)　s_1 が変数 v に値を代入する．
(2)　s_2 が変数 v の値を参照する．
(3)　s_1 から s_2 に，v の値を書き換えない実行経路が少なくとも 1 つ存在する．

条件 3 に示されている通り，データ依存関係を求めるには，制御フロー情報が必要である．しかし，Java のようなプログラミング言語では，例外処理や `finally` ブロックのように複雑な制御フローをもたらすプログラム要素が存在しており，計算に時間がかかる．

2.2　制御フローを考慮しないデータ依存関係

我々の研究グループが提案したデータフローの可視化手法[4] では，文の順序に関わらず，ある文 s_1 が変数 v に値を代入しており，別の文 s_2 が変数 v の値を参照している場合を，近似計算したデータ依存関係であるとみなす．これは，データ依存関係の条件 3 を取り除いたものに相当し，解析速度を向上するかわりに不適切なパス (infeasible path) を抽出しうる．たとえば**図 1** 左のコード片では，1 行目で変数 `filename` に代入された値が 2 行目と 5 行目で参照され，3 行目で代入された値が 5 行目で参照される．また，5 行目で代入された変数 `f` は 6 行目で参照される．これらのデータ依存関係を図で表現すると，図 1 右のグラフにおける実線で示された辺となる．制御フローを考慮しない場合は，すべての代入と参照の組を取るので，変数 `filename` については，3 行目の代入から 2 行目の参照へと，不適切なパスを抽出する．図 1 ではこの辺を点線で示している．変数 `f` については，代入と参照がそれぞれただ 1 つしか存在しないので，制御フローを考慮しなくとも，結果に変化はない．

過去の実験[4] では，JEdit の複数のメソッドに渡るデータフローを調査する 2 つのタスクで，不適切なパスが発見されたのは 1 度だけであった．そして，不適切なパスの出現は開発者の作業に影響を与えず，データ依存関係解析の有効性を損なわなかった．不適切なパスが少なかった要因として，ローカル変数は使用する直前に宣言すべきであること，1 つのローカル変数は 1 つの目的に使用すべきであること（他の値を格納するために再利用しない）[1] が知られているためと考えられる．変数が再利用されず，ただ 1 度しか値を代入されなければ，その 1 つの代入からすべての参照へデータ依存辺を接続することで，制御フロー解析を用いずとも，正確なデータ依存関係を抽出することが可能である．このような状況が，JEdit や他の Java プログラムにおいて一般に成り立つことが確認できれば，悦田の手法[4] が一般の Java プログラムで有効に働く可能性が高いと予測できるとともに，データ依存関係解析に基づく様々な手法[2),8),17)] の計算の高速化にもつながる．

なお，解析の簡略化によって短縮できる時間の見積もりであるが，著者らが過去に実装したスライシングツール[6] で制御フロー解析，データ依存解析，ポインタ解析を完全に実行したところ，10 万行程度のプログラムに対するシステム依存グラフの構築に約 30 分を要していた．制御フローを考慮しなければ，同じ規模のプログラムでも約 5 分で処理を完了することが可能であった[15]．これらのツールは内部で使用しているデータ構造が違うので，厳密な比較ではないが，大幅な時間短縮を見込める可能性がある．

2.3　用　　語

本論文において，不適切なパスとは，制御フローを考慮したデータ依存関係解析では抽出されないが，制御フローを考慮しないデータ依存関係解析では抽出されるデータ依存関係のみを示すものとする．制御フローを考慮したデータ依存関係解析であっても，たとえば条件

式が必ず偽になる if 文に含まれた代入文などが原因で，プログラムの実行時に発生しえないデータ依存関係を持つことはある[12]が，それらは不適切なパスとはみなさない．

3. データ依存関係の調査

制御フローの考慮がデータ依存関係の正確さに与えている影響を明らかにするために，制御フローを考慮せず近似したデータ依存関係を，従来のデータ依存関係解析の結果と比較した．プログラムに含まれるデータ依存関係は，ローカル変数，メソッド呼び出しの引数，戻り値あるいは例外，フィールドによって生じるが，本研究では，このうち，ローカル変数によって生じる手続き内データ依存関係の調査を行っている．メソッド呼び出しの引数や戻り値によって生じるデータの流れは動的束縛の解決の正確さによって定まるが，広く使用されている Class Hierarchy Analysis[3] はクラス階層情報のみを使用して動的束縛の解決を行うため，制御フロー解析の有無は，メソッド呼び出しによって生じるデータ依存関係に影響を与えない．

フィールドや配列によって生じる手続き内・手続き間データ依存関係については，同一のオブジェクトを読み書きする可能性のある文間で，データ依存関係を接続することになる．このデータ依存関係の解析は，同一のオブジェクトを参照する可能性の有無をポインタ解析の結果に基づいて判定することになる．本研究のように制御フローグラフの作成を行わないのであれば，制御フローを考慮しないポインタ解析アルゴリズム[11]が適用可能である一方で，制御フローを考慮したポインタ解析手法は適用不可能である．制御フロー解析の有無による，フィールドや配列によって生じるデータ依存関係の変化の比較は，ポインタ解析のアルゴリズムの違いにも影響を受けるため，本論文ではメソッド間データフローに関しては，議論の対象から外している．

3.1 データ依存関係の抽出方法

本研究では，Java プログラムの各ローカル変数について，制御フローを考慮したデータ依存関係解析を実行する．データ依存関係の解析には，以下のような方式，条件を採用した．

バイトコード解析． データ依存関係の解析は，Java のバイトコード解析を用いて実装した．バイトコード上では，`finally` ブロックや `break`，`continue` などの実行制御文が，いわゆる GOTO 命令の形式に変換されており，容易に制御フローグラフを取得することができる．バイトコード上では，ローカル変数の値を計算用のスタックに読み出す `LOAD` 命令群（int 型変数に対する `ILOAD` など変数型ごとの命令[10]）と，ローカル変数の値をインクリメントする `IINC` 命令が，ローカル変数に対する参照となる．また，ローカル変数への代入となるのは，`STORE` 命令群と `IINC` 命令である．代入命令の数と参照命令の数の積が，制御フローを考慮しない場合のデータ依存辺の数となり，制御フローグラフから計算されたデータ依存辺の数との差が，不適切なパスの数となる．

取り扱うローカル変数． 本研究で扱うローカル変数とは，メソッドの中で宣言されたすべてのローカル変数であり，メソッドの仮引数や，`catch` 節で宣言された変数を含む．メソッドを実行しているオブジェクトを表現する予約語 `this` については，仮引数の一種と考えることもできるが，メソッドの実行中には書き換えが不可能な値であり，データ依存関係を必ず正しく抽出できるため，今回の解析対象からは除外している．

仮引数に関するデータ依存関係． メソッドの最初の命令の直前に，すべての仮引数に値を代入する仮想的な命令があるとみなすことで，データ依存関係解析を行う．

例外の発生に関する制御フローの扱い． Java では，命令の実行中に例外が発生すると，対応する `catch` ブロックに制御が移動する．本研究では，`try` ブロックのすべての命令について，その実行後に `catch` に遷移する可能性があると仮定した．なお，このような仮定を置かないと，`catch` ブロック内の命令が制御フローグラフ上で到達不能になってしまい，制御フローを考慮した場合にデータ依存関係の見逃しが生じる可能性がある．本研究では，制御フローを考慮した場合に正しい結果が出ることが望ましいと考え，例外に関する実行経路はすべて接続するものとした．

バイトコード上でのローカル変数の区別方法． ローカル変数は，バイトコード上では名前ではなく，変数に割り当てられた記憶域を示すインデクスで区別されており，スコープが互いに重ならない変数に関しては，異なる変数であっても同一のインデクスを割り当てられる可能性がある．本研究では，変数が同一のインデクスを持ち，同名であり，かつスコープの有効範囲が制御フローグラフ上で連結成分となっている場合にのみ同じ変数とみなすことで，異なるスコープに出現するローカル変数を区別する．デバッグ情報が付与されていないメソッドについては，ローカル変数の区別はインデクスのみで行った．これによって複数の変数を同一視してしまった場合は，不適切なパスの出現を多く見積もる可能性はあるが，見逃しは生じない．

3.2 調 査 対 象

調査対象とした 10 個のプログラムを**表 1** に示す．このうち，和歌山大学教務システムは，教育プロジェクト IT Spiral[19] で提供されているソフトウェア開発データである．JDK と和歌山大学教務システムを除き，バイナリ配布のパッケージ（ZIP ファイル）に含まれるすべてのクラスファイルを解析対象とした．バイナリ配布パッケージには，サンプルコードな

表 1　解析対象の Java プログラム
Table 1　Target Java Programs

Program	#Classes	#Methods	#Variables
ANTLR 3.3	1063	10930	26129
Apache Ant 1.8.2	1177	11033	20282
Apache Tomcat 7.0.8	2368	25632	60422
Batik 1.7	5877	46326	100485
Eclipse SDK 3.6.2	33270	258073	610099
JDK 1.6.0	15951	134497	308239
JEdit 4.3.2	1045	7521	18417
和歌山大学教務システム[19]	3788	36484	70842
Torque 3.3	2106	21468	40340
Vuze 4.6.0.2	6739	41822	101912
Total	72575	584353	1339110

```
1: int x = a_value;          1: int x1 = a_value;
2: use(x);                   2: use(x1);
3: x = another;              3: int x2 = another;
4: anotherUse(x);            4: anotherUse(x2);
```

図 2　変数の名前変更によって不適切なパスを除去できるコードの例.
Fig. 2　A variable renaming that prevents infeasible data-flow paths

ども含まれていることがあるが，コンパイルされているファイルはすべて解析対象とした．JDK に関しては，`jre/lib` サブディレクトリ下にある `rt.jar` をはじめとするファイルを解析対象とした．和歌山大学教務システムは，著者の環境で Eclipse 3.4 を用いてコンパイルしたもので，ある企業によって書かれた 170 個のクラスと，これらのクラスが使用しているオープンソースのフレームワークやライブラリをすべて含んだ数値である．

Total 行には，10 個のシステムの合計を示しているが，2 つ以上のシステムに共通したクラス 809 個を取り除いているため，各システムの個別の値の合計とは一致しない値になっている．クラスの重複は，同一パッケージ名，同一クラス名によって判定している．

3.3　データ依存関係に基づく変数の分類

変数ごとに計算されたデータ依存関係の情報から，制御フローを考慮せずともデータ依存関係を抽出できるかどうかで，変数を次の 3 種類に分類する．

Correct:　すべての代入からすべての参照にデータ依存関係を持っており，制御フロー解析を行わなくとも，従来のデータ依存関係解析と同一の結果を得ることができる変数.

Split:　変数の名前を付け替えることで，複数の Correct 変数に分割することができる変数．具体例を示すと，**図 2** に示されている 2 つのコード片のうち，左側のコードに登場する変数 `x` が該当する．このコードでは，1 行目で `x` に代入された値は 2 行目のみで使われており，3 行目で `x` に代入された値は 4 行目で使われている．そのため，1 行目から 2 行目までと，3 行目から 4 行目までのデータ依存関係が完全に独立したものとなっており，図 2 右側のコードのように変数の名前を付け替えると，不適切なパスを除去することができる．このような変数は，本来は Correct 変数のみで記述できる処理であることから，次の Infeasible とは区別して扱う．

Infeasible:　不適切なパスを 1 つ以上持ち，かつ，Split の条件を満たさない変数.

次に，これら 3 つの分類の形式的な定義を示す．まず，ローカル変数 v について，代入命令集合を D_v，参照命令集合を U_v とすると，データ依存辺の集合 E_v は，次のように表現できる．

$$E_v = \{(d,u) | d \in D_v \text{ から } u \in U_v \text{ へと，代入命令 } k \in D_v (k \neq d) \text{ を通過せずに到達可能な制御フローグラフ上の経路が存在する．}\}$$

また，制御フローを考慮しないデータ依存辺 I_v は，次のように定義できる．

$$I_v = \{(d,u) | d \in D_v, u \in U_v\}$$

これらを用いて，各変数 v の分類 Correct, Split, Infeasible を，次のように定義する．

Correct:　$E_v = I_v$ が成立する変数.

Split:　$E_v = \cup_{i=1}^{k} I_{vi}$ となるように，D_v と U_v をそれぞれ $D_{v1}, \cdots, D_{vk}$ および $U_{v1}, \cdots, U_{vk}$ と分割[⋆1]することが可能な変数.
図 2 の左側のコードにおける変数 `x` の場合，1 行目で代入された値は 2 行目のみで使われており，3 行目で代入された値は 4 行目で使われているから，$D_x = \{1,3\}, U_x = \{2,4\}$ であり，$E_x = \{(1,2),(3,4)\}$ となる．これは明らかに $I_v = \{(1,2),(1,4),(3,2),(3,4)\}$ とは異なるため，`x` は Correct の条件を満たさない．D_x, U_x を $D_{x1} = \{1\}, D_{x2} = \{3\}, U_{x1} = \{2\}, U_{x2} = \{4\}$ へと分割すると，$I_{x1} = \{(1,2)\}, I_{x2} = \{(3,4)\}$ となり，$E_x = I_{x1} \cup I_{x2}$ が成り立つことから，`x` は Split の条件を満たす.

⋆1 集合 S の分割とは，$S = \cup_{i=1}^{k} S_i$ を満たし，$1 \leq i,j \leq k, i \neq j$ であるようなすべての i,j について $S_i \cap S_j = \phi$ を満たすような S の部分集合 $S_1, \cdots, S_k$ を求めることである.

表 2　データ依存関係に基づく変数の分類結果
Table 2　Variables categorized by data dependence

Program	Variables	Correct		Split		Infeasible	
ANTLR 3.3	26129	22754	(87.1%)	2275	(8.7%)	1100	(4.2%)
Apache Ant 1.8.2	20282	18976	(93.6%)	529	(2.6%)	777	(3.8%)
Apache Tomcat 7.0.8	60422	56136	(92.9%)	1760	(2.9%)	2526	(4.2%)
Batik 1.7	100485	90415	(90.0%)	5298	(5.3%)	4772	(4.7%)
Eclipse SDK 3.6.2	610099	577535	(94.7%)	13667	(2.2%)	18897	(3.1%)
JDK 1.6.0	308239	273458	(88.7%)	22971	(7.5%)	11810	(3.8%)
JEdit 4.3.2	18417	17383	(94.4%)	379	(2.1%)	655	(3.6%)
和歌山大学教務システム	70842	66048	(93.2%)	2474	(3.5%)	2320	(3.3%)
Torque 3.3	40340	37238	(92.3%)	1782	(4.4%)	1320	(3.3%)
Vuze 4.6.0.2	101912	97566	(95.7%)	1817	(1.8%)	2529	(2.5%)
Total	1339110	1240879	(92.7%)	52279	(3.9%)	45952	(3.4%)

表 3　データ依存関係に基づくメソッドの分類結果
Table 3　Methods categorized by data dependence

Program	Methods	Correct		Split		Infeasible	
ANTLR 3.3	10930	9480	(86.7%)	482	(4.4%)	968	(8.9%)
Apache Ant 1.8.2	11033	10218	(92.6%)	262	(2.4%)	553	(5.0%)
Apache Tomcat 7.0.8	25632	23163	(90.4%)	722	(2.8%)	1747	(6.8%)
Batik 1.7	46326	41641	(89.9%)	1752	(3.8%)	2933	(6.3%)
Eclipse SDK 3.6.2	258073	237420	(92.0%)	7064	(2.7%)	13589	(5.3%)
JDK 1.6.0	147962	130518	(88.2%)	9373	(6.3%)	8071	(5.5%)
JEdit 4.3.2	7521	6817	(90.6%)	219	(2.9%)	485	(6.4%)
和歌山大学教務システム	36484	33620	(92.1%)	1018	(2.8%)	1846	(5.1%)
Torque 3.3	21468	19690	(91.7%)	819	(3.8%)	959	(4.5%)
Vuze 4.6.0.2	41822	39260	(93.9%)	876	(2.1%)	1686	(4.0%)
Total	597818	543260	(90.9%)	22324	(3.7%)	32234	(5.4%)

なお，変数 v が Split の条件を満たすとき，D_v および U_v の分割方法 $v_1, \cdots, v_k$ は一意に定まる．なぜなら，他に分割の候補 $v'_1, \cdots, v'_l$ を作るには，少なくとも 1 つの v_i について，$d \in D_{vi}, u \in U_{vi}$ であるが，ある $m, n (m \neq n)$ について $d \in D_{v'm}, u \in U_{v'n}$ となるような組 (d, u) を作らなければならない．ここで，$(d, u) \in E_v$ であるが，$(d, u) \notin I_{v'm}$ かつ $(d, u) \notin I_{v'n}$ となり，$E_v = \cup_{i=1}^{l} I_{v'i}$ を満たすことができない．よって，条件を満たす分割は $v_1, \cdots, v_k$ の他には存在しない．

Infeasible:　$E_v \neq I_v$ であり，Split の条件を満たさない変数．

この形式化に基づいて，調査対象のすべてのローカル変数を分類した結果を**表 2** に示す．この表から，変数の 90%に関しては，制御フロー解析なしでも，従来のデータ依存関係と同一の計算結果を得ることができ，変数名の簡単な変更を行うだけで，95%の変数については正しい結果を得ることができると読み取れる．プログラムごとに大きな差は見られず，Java プログラムにおいて一般的な傾向であると考えることができる．

このような結果が出た理由の 1 つとして，1 つの変数を 1 つの目的にのみ使うというコーディング方法が考えられる．各変数について，その代入命令の個数を数えたところ，代入命令がただ 1 つであるような変数，すなわち不適切なパスが抽出されることのない変数は 85.3%を占めていた．代入命令が 2 つであるような変数が 10.7%であり，代入命令が 3 つ以上あるような変数は約 4%となっている．これら 14.7%の変数のうち，3 分の 1 だけが，制御フローの考慮が必要な変数に該当している．

3.4　データ依存関係に基づくメソッドの分類

変数に対する分類の結果は，各変数が不適切なパスの原因になっているかどうかを示すものである．各メソッドについて考えると，1 つでも，不適切なパスを生じる変数が存在すれば，そのメソッドについては不適切なパスを生じることになる．そこで，変数の分類結果を用いて，メソッドに対しても同様の分類を行う．

Correct:　メソッドに出現するすべてのローカル変数が Correct に分類されているメソッド．つまり，制御フロー解析を行わなくとも，従来のデータ依存関係解析と同一の結果を得ることができるメソッド．

Split:　メソッドに出現する 1 つ以上のローカル変数が Split に分類されており，残るすべてのローカル変数が Correct に分類されているメソッド．つまり，開発者が変数の名前を付け替えさえすれば，プログラムの意味を変えることなく従来のデータ依存関係解析と同一の結果を得ることができるようになるメソッド．

Infeasible:　メソッドに出現する 1 つ以上のローカル変数が Infeasible に分類されているメソッド．制御フローの考慮なしには不適切なパスを生じるメソッドがここに分類される．

表 3 に，メソッド単位での分類結果を示す．メソッド単位で考えた場合でも，90.9%のメソッドについては制御フロー解析なしに，従来のデータ依存関係解析と同一の結果を得ることが可能である．また，変数の場合と同様に，プログラムごとの差異は大きくない．

正確な結果を必要とする解析では，メソッドが不適切なパスを持ちうる場合にのみ制御

表 4　制御フロー解析が必要な変数，メソッドの割合
Table 4　Variables and Methods that requires flow-sensitive analysis

	Variables			Methods		
Program	Incorrect	代入 2 回以上		Incorrect	代入 2 回以上	
ANTLR 3.3	3375	5482	(21.0%)	1450	2279	(20.9%)
Apache Ant 1.8.2	1306	2626	(12.9%)	815	1340	(12.1%)
Apache Tomcat 7.0.8	4286	9938	(16.4%)	2469	4291	(16.7%)
Batik 1.7	10070	17752	(17.7%)	4685	7232	(15.6%)
Eclipse SDK 3.6.2	32564	80494	(13.2%)	20653	41481	(16.1%)
JDK 1.6.0	34781	53015	(17.2%)	17444	24840	(16.8%)
JEdit 4.3.2	1034	2482	(13.5%)	704	1347	(17.9%)
和歌山大学教務システム	4794	9271	(13.1%)	2864	4849	(13.3%)
Torque 3.3	3102	5659	(14.0%)	1778	2976	(13.9%)
Vuze 4.6.0.2	4346	11984	(11.8%)	2562	5297	(12.7%)
Total	98231	195935	(14.6%)	54558	94440	(15.8%)

フローを考慮した解析を実施し，そうでない場合には制御フローを考慮しない解析を行うという方式が可能である．この判定を軽量に，かつ安全行うことができると考えられる基準は,「変数への代入が 2 つ以上存在すること」である．変数への代入が 1 つしかなければ，参照がどれだけあっても，必ずその代入が到達するはずなので，安全に解析を行うことができる．**表 4** に，代入が 2 つ以上ある変数の割合と，そのような変数を少なくとも 1 つ持つようなメソッドの割合を示す．この表において，Incorrect 列は，分類が Correct ではない（Split または Infeasible である）変数およびメソッドの数を示しており，代入が 2 回以上ある変数は，これらの変数を包含し，その数はおよそ 2 倍から 3 倍程度となっている．また，代入が 2 回以上ある変数を少なくとも 1 つ持つようなメソッドの数は，Correct に分類されたメソッドの 2 倍程度であった．この表から，すべてのメソッドのうち 15.8%だけに制御フロー解析を行えば，正確なデータ依存関係を計算することができることが分かる．プログラムによって値は異なり，今回の調査対象では最大で 20.9%となっているが，それでも全体に対して解析する場合に比べて，時間コストを大幅に削減できる可能性を示している．

3.5　フィールド，配列の手続き内データ依存関係への関与

ここまで，ローカル変数によって生じる手続き内データ依存関係について制御フロー解析なしでも，制御フロー解析ありの場合と同じ計算結果が得られるメソッドの割合を示した．しかし，手続き内でのデータ依存関係に，フィールドや配列が大きく貢献しているのであれば（開発者が配列やフィールドを手続き内でのデータ受け渡しに使用していれば），ローカル変数のデータ依存関係を知ることの価値が低いことになってしまう．そこで，オブジェクトを考慮しない解析（object-insensitive analysis）によって，フィールドや配列によって生じている手続き内データ依存関係の個数を調べた．

フィールドや配列の値は，オブジェクト参照や添え字の値によって，読み書きする記憶領域が変化する．ローカル変数であれば，値の書き込みによってそれまで格納されていた値を必ず上書きすると考えるが，フィールドの場合は別オブジェクトへの書き込み，配列の場合は異なる添字を用いての書き込みの可能性を考慮して，各書き込み命令で書かれた値は，それまでに書き込まれた値を上書きするとは限らないと考える．つまり，一度書き込まれた値は，制御フローによって到達しうるすべての読み込み命令によって参照される可能性があるとみなす．フィールドに関しては，フィールドの名前，型，所有クラス名によって区別し，あるフィールドに値を代入する命令から，同一フィールドの値を参照する命令に到達可能であれば，そのフィールドが手続き内でのデータ依存関係に関係していると考える．配列に関しては，解析の簡略化のため，プリミティブ型の配列は型名によって区別し，オブジェクトを格納する配列に関しては型を無視してすべて同一種の配列とみなす．ある配列への書き込み命令から，同じ要素型の配列の読み込み命令へと制御フローによって到達可能であれば，その配列が手続き内のデータ依存関係に関係していると考える．

解析の結果，10 個のプログラムに定義されたフィールドは全部で 245,684 個であり，1 つのフィールドは，平均で 1.47 個のメソッドによって値を書き込まれ，2.69 個のメソッドによって値を参照される．1 つの手続き内で，フィールドへの代入命令から，同一フィールドの参照命令に到達可能であったメソッドは，36,732 個であった．これは，全メソッドの 6.1%である．また，配列に対する書き込み命令から配列の読み込み命令に到達可能なメソッドは 9,290 個であった．これは全メソッドの 1.6%である．これらの結果から，メソッドの多くはローカル変数を手続き内でのデータの受け渡しに用いており，制御フローを無視した解析によって手続き内のデータ依存関係を正しく計算できることを示している．

4. 議　　論

4.1　妥当性への脅威

本研究の結果は，10 個の Java プログラムに対して求められたものである．対象となったプログラムは，開発環境，サーバ，テキストエディタなど様々な目的のために作られたソフトウェアであり，アプリケーションソフトウェアやライブラリを含んでいることから，多様な Java プログラムの書き方が反映されていると考えられる．一方で，企業の開発者が記述

したコードは和歌山大学教務システムの一部に限られており，企業の開発者が従うコーディング規約による影響はあまり反映されていない．そのため，企業のソフトウェアを収集して実験を行うと，異なる結果となる可能性がある．

本研究での調査は，プログラミング言語を Java に限定したものとなっている．近年では企業のソフトウェア開発でも Java が多く採用されていることから，この結果自体は有益であるが，たとえば C 言語や COBOL など，他の言語に適用した場合には，言語特有のコーディング方法による違いが出る可能性がある．

本研究では，データ依存関係解析をバイトコード上で定義し，例外処理に関しては保守的な制御フロー解析を採用した．例外処理を含む制御フローグラフをより正確に構築することができれば，制御フローを考慮したデータ依存関係解析がより正確になり，制御フローを考慮しない解析の正確さが相対的に低下する可能性がある．

バイトコード上でデバッグ情報から変数の識別を行ったため，デバッグ情報の不足や，コンパイラによって自動生成された変数が結果に反映されている．デバッグ情報が不足した変数については，スコープの異なる 2 つ以上の変数を同一視している．このとき，1 変数あたりの代入命令の数を多く算定するので，本来なら Correct であった 2 つの変数が，1 つの Split に変わっている可能性がある．また，自動生成された変数，たとえば拡張 `for` 文によって作られる `Iterator` 型の変数によって，正しく解析可能な変数の数を増加させている．ただし，自動生成された変数に関しては，メソッド単位での計算には影響を与えていない．

4.2　プログラムスライシングへの影響

本研究では，データ依存関係の近似計算のみを議論している．システム依存グラフの定義[5] には制御依存関係も含まれており，本研究の成果が，ただちにプログラムスライシングを置き換えるわけではない．

プログラムスライシングにおいて，正確な計算を要求する場合は，データ依存関係を求める計算のみを省略する形で，計算時間の短縮に貢献する．Thin Slicing[16] のように，データ依存関係のみを用いるスライシング技法には，本研究の成果をただちに適用可能である．

プログラムスライシングの簡略化を行い，制御フローグラフの構築を完全に省略することを目指す場合は，制御依存関係を近似計算する手法が必要である．悦田の手法[4] の場合は構文情報のみを用いた近似計算を導入しているが，この方法に基づく近似計算の性能に関しては，実験による評価が今後の課題となっている．

4.3　Ripple Effect

本研究におけるデータ依存関係の正確性の評価は，すべて手続き内の個別の依存辺に対して行ったものである．制御フローを考慮しないことによる正確さの変化が，悦田の実験[4] では開発者の作業に悪影響を与えなかったが，プログラムスライシングなどデータ依存関係を使用した手法に与える影響は，まだ評価していない．この評価にあたっては，データ依存辺の有無だけでなく，グラフ上での到達関係の変化に与える影響（Ripple Effect）が問題となる．たとえば，図 1 のコード片に対して，6 行目の変数 `f` に対するプログラムスライスを計算すると，制御フローを考慮した場合でも，しなかった場合でも，1 行目から 5 行目までがスライスに含まれるので，データ依存関係の差異はスライス結果に影響を与えない．一方，2 行目の変数 `filename` に対するプログラムスライスは，制御フローを考慮すると 1 行目だけであるが，制御フローを考慮しないと 1 行目と 3 行目がスライスに含まれることになり，結果に違いが生じる．このような解析結果への影響の評価は，今後の課題である．

5. 関 連 研 究

本研究では，データ依存関係の定義そのものは変更せず，制御フローを考慮せずに計算したデータ依存関係の結果を，従来の計算方法と比較した．この方法と対照的なアプローチとして，Jász ら[7] による，データ依存関係を制御フローで近似する手法が提案されている．Jász らの方式は，ある命令が，別の命令よりも後に実行されるのなら，前の命令に何らかの影響を受けていると考える．単純に制御フローグラフ上での順序関係を用いるので，逐次的に実行される処理であれば，高速に正確な結果を得られるが，メッセージループからすべての処理を呼び出す種類のプログラムに対しては正確な結果を得られない．一方で，本研究のようにデータ依存関係のみを用いる手法では，処理の順序を考慮しないため，逐次的に行われる変数の更新に対して不適切なパスを抽出することがある．そのため，2 つの手法は互いに相補的な関係にあると考えられる．

この他の関連研究として，特定の目的に適用できるように，簡略化した依存関係解析を定義した手法が存在する．Groum[13] は，Java の 1 つのメソッド内部で生じるメソッド呼び出しとフィールドへのアクセスの順序関係を，循環を持たない有向グラフとして表現したものである．Groum は，メソッド呼び出しなどの順序に対するパターンマイニングを目的としているため，2 つの文が少なくとも 1 つの変数を共有していればそれら 2 つの文に依存関係があるという単純な定義を用いている．そのかわり，ループ構造によるデータ依存関係などは抽出しない．

Lawall ら[9] は，C 言語における整数型の定数に関するデータ依存関係を抽出する手法を提案している．実装上は制御フローを考慮している解析であるが，定数が代入されうる変数

に関するビット演算，比較演算のみに限定して解析を行い，不適切な定数比較を検出する手法となっている．

6. ま と め

本研究では，制御フローを考慮せず，すべての代入文から参照文へと辺を接続して得られるデータ依存関係であっても，90.9%のメソッドについては，制御フロー解析を用いた従来の解析手法と同一の結果を得ることができ，また，正確な結果を得たい場合に解析が必要なメソッドは 15.8%だけであることを示した．

今後の課題として，制御フローグラフを考慮しない場合に生じるデータ依存関係が，推移的な到達関係に与える影響と，エイリアス解析やプログラムスライシングなどの高度な解析手法に与える影響を評価することが挙げられる．プログラムスライシングのような手続き間のデータ依存関係解析は，計算に時間がかかることから，大規模プログラムに対してはほとんど適用されていない．制御フローグラフの構築を省略した高速なデータ依存関係に基づいて，プログラムスライシングなどの解析を実用上問題ない正確さで計算できるのであれば，大規模プログラムに対する高度な解析手法の適用が容易になると考えられる．

謝辞　本研究は，日本学術振興会科学研究費補助金基盤研究（A）（課題番号:21240002）の助成を得た．

参 考 文 献

1) Ambler, S.W.: Writing Robust Java Code. The AmbySoft Inc. Coding Standards for Java, v17.01d, Chapter 4 (2000).
2) Chen, K. and Rajlich, V.: Case Study of Feature Location Using Dependence Graph, *Proceedings of the 8th International Workshop on Program Comprehension*, pp.241–247 (2000).
3) Dean, J., Grove, D. and Chambers, C.: Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis, *Proceedings of the 9th European Conference on Object-Oriented Programming*, pp.77–101 (1995).
4) 悦田翔悟，石尾　隆，井上克郎：変数間データフローグラフを用いたソースコード間の移動支援，情報処理学会研究報告（SE），Vol.2011-SE-171, No.12, pp.1–8 (2011).
5) Horwitz, S., Reps, T. and Binkley, D.: Interprocedural Slicing Using Dependence Graphs, *ACM Transactions on Programming Languages and Systems*, Vol.12, No.1, pp.26–60 (1990).
6) 石尾　隆，仁井谷竜介，井上克郎：プログラムスライシングを用いた機能的関心事の抽出，コンピュータソフトウェア，Vol.26, No.2, pp.127–146 (2009).
7) Jász, J., Árpád Beszédes, Gyimóthy, T. and Rajilich, V.: Static Execute After/Before as a Replacement of Traditional Software Dependencies, *Proceedings of the 24th IEEE International Conference on Software Maintenance*, pp.137–146 (2008).
8) Komondoor, R. and Horwitz, S.: Effective, Automatic Procedure Extraction, *Proceedings of the 11th International Workshop on Program Comprehension*, pp.33–42 (2003).
9) Lawall, J. and Lo, D.: An Automated Approach for Finding Variable-Constant Pairing Bugs, *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pp.103–112 (2010).
10) Lindholm, T. and Yellin, F.: *Java Virtual Machine Specification, the 2nd Edition*, Prentice Hall (1999).
11) Milanova, A.: Light Context-Sensitive Points-to Analysis for Java, *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp.25–30 (2007).
12) Ngo, M.N. and Tan, H. B.K.: Detecting Large Number of Infeasible Paths through Recognizing their Pattenrs, *Proceedings of the 15th Symposium on Foundations of Software Engineering*, pp.215–224 (2007).
13) Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M. and Nguyen, T.N.: Graph-based mining of multiple object usage patterns, *Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, pp.383–392 (2009).
14) Parnin, C. and Rugaber, S.: Resumption strategies for interrupted programming tasks, *Software Quality Journal*, Vol.19, No.1, pp.5–34 (2011).
15) 柳　慶吾，石尾　隆，井上克郎：ソフトウェア部品利用例抽出のためのデータフロー解析手法の提案と評価，情報処理学会研究報告（SE），Vol.2010-SE-167, No.29, pp. 1–8 (2010).
16) Sridharan, M., Fink, S.J. and Bodik, R.: Thin Slicing, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pp.112–122 (2007).
17) Wang, X., Lo, D., Cheng, J., Zhang, L., Mei, H. and Yu, J. X.: Matching Dependence-Related Queries in the System Dependence Graph, *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pp.457–466 (2010).
18) Weiser, M.: Program Slicing, *IEEE Transactions on Software Engineering*, Vol.10, No.4, pp.352–357 (1984).
19) 先導的 IT スペシャリスト育成推進プログラム：http://it-spiral.ist.osaka-u.ac.jp/.