論　文

プログラム依存グラフの効率的な更新手法

高田　智規[†]　　佐藤　慎一[††]　　井上　克郎[†]

Incremental Update Method of Program Dependence Graph

Tomonori TAKADA[†], Shinichi SATO[††], and Katsuro INOUE[†]

あらまし　プログラム依存グラフ（Program Dependence Graph, PDG）は，プログラム中の文間の依存関係を表す有向グラフである．PDGの辺をたどることにより，ある文に関連する文の集合（プログラムスライス，スライス）を抽出することができる．PDGやスライスはデバッグ・保守などさまざまな用途に用いられる．一般に，PDGの計算には時間がかかる．しかし，頻繁にプログラムを変更しそのスライスを求めるような場合でも，これまではプログラムが変更されるたびにPDG全体を再計算していた．そこで本論文では，プログラムが変更されたときに，PDGのプログラムの変更箇所に対応する部分だけを更新するアルゴリズムを提案する．これにより，再計算の時間が軽減され，デバッグ等を効率的に行うことが期待される．また，既存のデバッグ支援システムに本手法の実装を行い，実際に本手法の有効性を確認する．

キーワード　プログラム依存グラフ，更新，プログラムスライス，デバッグ支援

1. まえがき

プログラム依存グラフ（Program Dependence Graph, PDG）は，プログラムの各文における変数間の依存関係を表す有向グラフである[1],[6],[7]．PDGの各節点はプログラム中の各文・条件式を表し，有向辺は依存関係（データ依存・制御依存）を表す．PDGの有効辺をたどることにより，プログラムスライス[2]～[7],[9],[13]～[16]と呼ばれる，プログラム中のある文に影響を受ける，または影響を与える文の集合を抽出することができる．プログラムスライスはデバッグやテスト，保守，プログラム合成などに利用されている．

これまでに，プログラムスライスを抽出しデバッグを効率良く進めるためのシステム[12]を開発した．このシステムは抽出したプログラムスライスを対象としてデバッグを行う機能をもっている．このシステムではプログラムに変更を加えるたびにPDG全体を変更されたソースプログラムの全体から再計算していたが（これをここではPDG再計算と呼ぶ），PDG再計算は複雑であり，それに多くの時間を費やしていた．これはインタラクティブにプログラムの修正や実行を行い，デバッグを効率良く行う際の大きな障害となっていた．

PDGのうちプログラムの変更箇所に関する部分だけを更新することができれば，PDG再計算に要していた時間が短縮され，作業効率の大幅な向上が期待できる．

プログラム内の部分的な変更が，その他の文に与える影響を調べるアルゴリズムは既に提案されている[8],[10],[11]．文献[8]のアルゴリズムでは，PDGの修正を容易にするためだけに，各頂点で多くの情報を保持しておく必要があった（例えば，到達定義集合など）．また，関数間解析を行っていない文献[10],[11]は，データフローグラフの効率的な更新法を提案しているが，いずれも関数間にまたがる解析に必須な引数や戻り値，大域変数による依存関係解析の伝搬方法が明示されておらず，実際のシステムで実現が困難である．

そこで，プログラムの部分的変更が行われたときに関数境界を越える依存関係を正しく計算し，効率良くPDGを再計算する手法を提案する．また，前述のデバッグ支援システムに提案する手法を実装し，PDGの再計算と本手法との実行時間を比較し，その有効性を確認する．

[†] 大阪大学大学院基礎工学研究科，豊中市
Graduate School of Engineering Science, Osaka University, Toyonaka-shi, 560-8531 Japan
[††] NTTデータ通信株式会社，川崎市
NTT Data Corporation, Kawasaki-shi, 210-0913 Japan

2. 諸 定 義

2.1 入力言語

本論文では以下のような言語を入力言語として考える．この言語には文として条件文（if 文），代入文，繰返し文（while 文），入力文（readln 文），出力文（writeln 文），手続き呼出し文，複合文（begin-end）がある．変数の型としてはスカラ型のみを考える．プログラムは，大域変数宣言，手続き（関数）定義，メインプログラムからなり，ブロック構造はない．手続き内では内部で宣言された局所変数と仮引数変数および大域変数のみが参照可能で，他の手続き内の局所変数は参照できない．手続きは，自己再帰的および相互再帰的に定義可能であり，その引数は，値渡しで扱われる．

2.2 到達定義

文 s における変数 x（の値）の定義が文 t に到達するとは，文 s が変数 x を定義し，かつ，s から t に至る実行パス $s, u_1, u_2, \ldots, u_k, t$ 中に変数 x を再定義しないような実行パス $u_1, u_2, \ldots, u_k$ が存在する場合を言う．

文 n における変数 v の定義が文 s に到達する場合，文 s には到達定義 $\langle n, v\rangle$ が存在すると言う．到達定義集合とは，文 s に到達するすべての到達定義からなる集合である．

2.3 依存関係

プログラム中の文間の依存関係として以下の2種類を考える．

（1） データ依存関係（Data Dependence）

変数 v を参照している文 t において到達定義集合中に到達定義 $\langle s, v\rangle$ が存在するとき，文 s から文 t に対して変数 v に関するデータ依存関係があると言う．

（2） 制御依存関係（Control Dependence）

文 s が条件文または繰返し文の条件式であり，文 s の条件判定の結果によって文 t を実行するか否かが直接決まるとき，文 s から文 t への制御依存関係があると言う．

2.4 プログラム依存グラフ

プログラム依存グラフ（Program Dependence Graph, PDG）は，プログラム内の各文間に存在する依存関係を有向グラフで表したものである．本論文では，文献[16]のアルゴリズムを用いて求めた PDG を対象とする．

節点として，プログラム内の文または条件式（条件文・繰返し文の条件判定部分）に対応した節点（文節点）および，手続き境界を越える依存関係の解析などに用いる特殊節点がある．

辺として，データ依存関係を表すものをデータ依存辺と言い，制御依存関係を表すものを制御依存辺と言う．以下，節点（文）s から節点 t への変数 v に関するデータ依存辺を $s \xrightarrow{v} t$，節点 s から節点 t への制御依存辺を $s \dashrightarrow t$ と表す．

これらの依存関係を表す辺のほかに，フロー辺と呼ばれる実行順序を表す辺がある(注1)．文 s から文 t に（他の文の実行なしに）直接制御が移る可能性があるとき，節点 s から節点 t へのフロー辺が存在し，$s \rightarrow t$ と表す．

パス $s, \ldots, t$ が存在するとは，節点 s からフロー辺を順方向にたどり節点 t へ到達するような経路が存在することを言う．

図1に PDG の例を示す，図中の DD はデータ依存辺，CD は制御依存辺，flow はフロー辺をそれぞれ表す．また，PDG に対して，表1で示す集合を定義する．

2.5 前定義節点

節点 r における変数 v の定義が節点 s の入口に到達している場合，節点 r を，節点 s の変数 v に関する前定義節点と呼ぶ．また，前定義節点の集合を前定

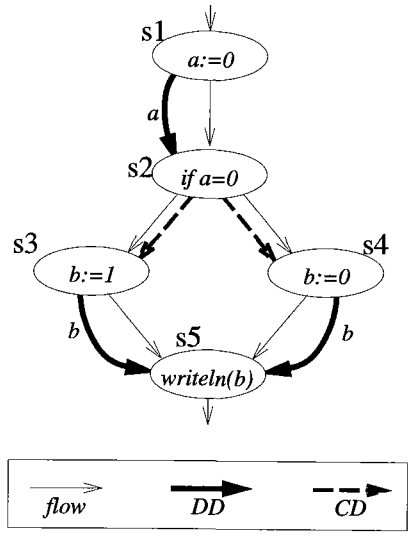

図1 PDG の例
Fig. 1 Example of PDG.

(注1)：フロー辺を PDG に含めない場合が多いが，ここではこのグラフを PDG と呼ぶ．

表1 PDGに対する集合
Table 1 Set for PDG.

$source(c,v)$	節点 c から変数 v に関するデータ依存辺をさかのぼって到達できる（有向辺を逆方向にたどった）節点の集合を表す（c 自身を含めない．以下同様）．
$target(c,v)$	節点 c から変数 v に関するデータ依存辺をたどって到達できる（有向辺を順方向にたどった）節点の集合を表す．
$prev(c)$	節点 c からフロー辺をさかのぼって到達できる節点の集合を表す．
$next(c)$	節点 c からフロー辺をたどって到達できる節点の集合を表す．
DD	データ依存辺の集合．
CD	制御依存辺の集合．
$FLOW$	フロー辺の集合．

義節点集合と呼ぶ．例えば，図1において節点 $s5$ の変数 b に関する前定義節点集合は $\{s3, s4\}$ となる．

2.6 プログラムの変更

プログラムの変更としてPDGの文節点一つに対し，以下の3種類の操作を考える．

- 文の削除

PDG上に存在する節点を削除する

- 文の挿入

新たに節点を作りPDGに挿入する

- 文の修正

節点はそのまま保存するが，その内容を修正する

3. アルゴリズムの概要

ここでは，ユーザーがプログラムの部分的変更を行う際，その作業の種類（文の削除/挿入/修正）を陽に指定し，それに従って制御依存辺を修正するものとする．例えば，条件文の節中に文を挿入した場合，条件文から挿入文に対して制御依存辺を追加する．逆に，条件文の節中の文を削除した場合，制御依存辺を削除する．このように制御依存辺の変化は，変更前の制御依存辺から容易に求めることができる．以降はデータ依存辺，フロー辺の変化についてのみ述べる．

文献[8]では，プログラムの変更に伴う到達定義集合の変化をフロー辺をたどり，後の節点に伝えていくことにより，変更によって変化した依存関係を再計算していた．しかし，この方法では各節点ごとに計算した到達定義集合を保存しておかなければならず，実装する際に非常に多くのメモリを必要としていた(注2)．

そこで，ここでは次のような工夫をした．PDGのデータ依存辺は解析時に到達定義集合をもとにして引かれている．よって，変更前のPDG中のデータ依存辺から，到達定義集合の一部を知ることができる．例えば，ある節点 s において $r \xrightarrow{v} s$ があったとき，変更前の s における到達定義集合の中に $\langle r, v\rangle$ が存在していたことになる．本手法ではプログラムが変更されたときに，到達定義集合を保存しておかなくても，このようなデータ依存辺の情報を利用し，依存関係の変化を求め，PDGの更新を行うことができる．

しかし，これだけでは関数境界を越える依存関係を正しく解析することはできない．そこで，関数内解析と関数間解析の2種類のアルゴリズムを考える．関数内解析アルゴリズムでは上記の手法により変更された文が含まれる関数内部のデータ依存辺を引き直す．関数間解析アルゴリズムでは変更による他の関数への影響を調べ，関数境界を越える依存関係を解析する．

4. 単一関数内解析アルゴリズム

まず，文の削除・挿入・修正それぞれに共通な前定義節点の発見のためのアルゴリズムを示し，次に削除・挿入・修正のアルゴリズムを示す．

4.1 前定義節点の発見

このアルゴリズムではフロー辺をさかのぼり，節点での定義・参照変数を調べることにより前定義節点を求める．このアルゴリズムは **4.2**〜**4.4** において用いる．

アルゴリズム FINDPREDEF

input: 節点 s，変数 v

output: 前定義節点集合 $PreDef(s,v)$

（1） $PreDef(s,v) \leftarrow \phi$

（2） $c \in prev(s)$ なる各 c に対して以下を順に実行．

（a） c において v が定義されていれば，

$$PreDef(s,v) \leftarrow PreDef(s,v) \cup \{c\}$$

（b） c において v が参照されていれば，

$$PreDef(s,v) \leftarrow PreDef(s,v) \cup source(c,v)$$

（c） c において v が定義も参照もされていなければ，$c \leftarrow prev(c)$ として 2a. 以下を実行．

4.2 削　　除

節点を削除することにより，その文における変数の定義が無効となる．そのような変数に関するデータ依存辺を引き直し，その後節点を削除する．

(注2)：通常，到達定義集合はデータ依存辺を引いた後は必要なく，保存する必要はない．

アルゴリズム DELETEVERTEX

input: 削除する節点 s

output: 更新された PDG

(1) s で定義されている各変数 v すべてについて，$DD \leftarrow DD \cup \{r \xrightarrow{v} t \mid r \in PreDef(s,v),\ t \in target(s,v)\}$

(2) $DD \leftarrow DD - \{r \xrightarrow{v} s \mid r \in source(s,v)\} - \{s \xrightarrow{v} t \mid t \in target(s,v)\}$

(3) $FLOW \leftarrow FLOW \cup \{p \rightarrow n \mid p \in prev(s),\ n \in next(s)\} - \{r \rightarrow s \mid r \in prev(s)\} - \{s \rightarrow t \mid t \in next(s)\}$

(4) 節点 s 自身を削除

4.3 挿　　入

このアルゴリズムではまず，フロー辺を付けかえる．次に，挿入する節点で定義する変数を参照する節点へデータ依存辺を引く．また，挿入によって依存関係のなくなるデータ依存辺を削除する．最後に，挿入する節点で参照する変数のデータ依存辺を引く．

アルゴリズム INSERTVERTEX

input: 挿入する節点 s，$prev(s)$，$next(s)$

output: 更新された PDG

但し，手続き・関数・プログラムの入口においてすべての変数を定義しているものとみなす．

(1) $FLOW \leftarrow FLOW \cup \{p \rightarrow s \mid p \in prev(s)\} \cup \{s \rightarrow n \mid n \in next(s)\} - \{p \rightarrow n \mid p \in prev(s),\ n \in next(s)\}$

(2) s で定義している各変数 v すべてについて，以下を実行．

(a) 各 $r \in PreDef(s,v)$，$t \in target(r,v)$ に対して，v を定義しないようなパス $s,\ldots,t$ が存在すれば，以下を実行．

i. $DD \leftarrow DD \cup \{s \xrightarrow{v} t\}$

ii. 任意の $r \in PreDef(s,v)$ に対して v を定義しないようなパス $r,\ldots,t$ が存在しなければ，

$$DD \leftarrow DD - \{r \xrightarrow{v} t \mid r \in PreDef(s,v)\}$$

(3) s で参照している各変数 u について，

$$DD \leftarrow DD \cup \{r \xrightarrow{u} s \mid r \in PreDef(s,u)\}$$

4.4 修　　正

このアルゴリズムは，削除アルゴリズムと挿入アルゴリズムを組み合わせることによって実現している．

変数集合 V を定義し変数集合 U を参照する節点 s を，変数集合 V' を定義し変数集合 U' を参照するように修正したとする．

アルゴリズム MODIFYVERTEX

input: 節点 s，変更後の定義変数集合 V'，変更後の参照変数集合 U'

output: 更新された PDG

(1) $FLOW \leftarrow FLOW$

(2) $v \in V - V'$ に対して，

$$DD \leftarrow DD \cup \{r \xrightarrow{v} t \mid r \in PreDef(s,v),\ t \in target(s,v)\} - \{s \xrightarrow{v} t \mid t \in target(s,v)\}$$

(3) $v' \in V' - V$ に対して，

(a) 各 $r \in PreDef(s,v'), t \in target(r,v')$ に対して，v を定義しないようなパス $s,\ldots,t$ が存在すれば以下を実行．

i. $DD \leftarrow DD \cup \{s \xrightarrow{v'} t\}$

ii. 任意の $r \in PreDef(s,v')$ に対して，v' を定義しないようなパス $r,\ldots,t$ が存在しなければ，

$$DD \leftarrow DD - \{r \xrightarrow{v'} t \mid t \in PreDef(s,v)\}$$

(4) $u \in U - U'$ に対して，

$$DD \leftarrow DD - \{r \xrightarrow{u} s \mid r \in PreDef(s,u)\}$$

(5) $u' \in U' - U$ に対して，

$$DD \leftarrow DD \cup \{r \xrightarrow{u'} s \mid r \in PreDef(s,u')\}$$

5. 複数関数間解析アルゴリズム

5.1 PDG

図 2 のプログラム proc は手続き inc の中で大域変数 a を参照・定義している．このプログラムの PDG は図 3 のようになる．関数（手続き）内部で関数外での大域変数を参照している場合は global-in と呼ばれる特殊節点を関数の入口に作り，関数内部で大域変数を定義している場合は global-out と呼ばれる特殊節点を関数の出口に作る．関数境界を越える依存関係がある場合はこれらの特殊節点を介してデータ依存辺を引く．

このような特殊節点を表 2 に示す．

```
program proc(input,output);
var a: integer;


procedure inc;
begin
  a:=a+1
end;


begin
  readln(a);
  inc;
  writeln(a)
end.
```

図2 プログラム proc
Fig. 2 Sample program proc.

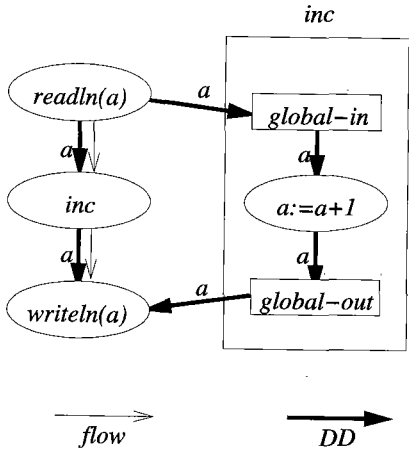

図3 プログラム proc の PDG
Fig. 3 PDG for sample program proc.

関数（手続き）中の文に対して変更を行った場合，その関数内で参照・定義している変数が変更前と異なる場合がある．このような場合は既存の特殊節点の削除や新たな特殊節点の挿入が必要になる．

このような変更に対応するため，文献[16]で PDG を計算する際に用いている確実定義集合，潜在定義集合，暗使用される変数集合を考える．

プログラムのある領域 S について，S を実行したときに必ずその値が定義される変数とその定義節点との組の集合を確実定義集合と呼び，$SuDEF(S)$ と表す．また，定義される可能性のある変数とその定義節点との組の集合を潜在定義集合と呼び，$PoDEF(S)$ と表す．

また，次に示す条件をすべて満たす変数 v を「手続き p で暗使用される変数」と呼び，その集合を $ImUSE(p)$ と表す．

- v は大域変数である
- p 内での v の参照地点に p 外の v の定義が到達する

表2 特殊節点
Table 2 Special vertex.

特殊節点		表記例
entry 節点	プログラムおよび手続き（関数）にひとつずつあり，そのプログラム（関数・手続き）の中のすべての文は entry 節点からの CD 関係がある	$f\text{-}Entry$
exit 節点	関数の戻り値を通して伝わる影響を検出するための節点で，各関数にひとつずつある	$f\text{-}exit$
global-in 節点	手続き外からの大域変数の影響を内部へ伝えるための節点で，手続きに，個々の大域変数に対して，ひとつずつある	$f_g\text{-}in$
global-out 節点	手続き内で定義された大域変数の影響をその外へ伝えるための節点で，手続きに，個々の大域変数に対して，ひとつずつある	$f_g\text{-}out$
paramter-in 節点	手続きの引数を通して伝わる影響を検出するための節点で，その引数それぞれにひとつずつある	$f_p\text{-}par$
parameter-out 節点	手続きの変数引数を通して伝わる影響を検出するための節点で，変数引数それぞれにひとつずつある	$f_p\text{-}par_{out}$

5.2 関数間解析

関数（手続き）f 中の文 s に対して変更を行うことを考える．

（1） s に関数内変更アルゴリズムを行う．

（2） f を直接，または間接に呼び出す関数および f 自身の集合を F とする．

（3） $\forall g \in F$ に対して，

$SuDEF(g) \leftarrow \phi$

$PoDEF(g) \leftarrow \phi$

$ImUSE(g) \leftarrow \phi$

（4） F 中各関数の $SuDEF$, $PoDEF$, $ImUSE$ を求める．

（5） （4）を F 中のすべての関数の $SuDEF$, $PoDEF$, $ImUSE$ が変化しなくなるまで繰り返す．

（6） 関数ごとに変更前後の $SuDEF$, $PoDEF$, $ImUSE$ を比較し，異なっていれば以下を実行する．

（a） 新たに定義・参照するようになった変数に関する global-out，global-in 節点を作り，関数内部のデー

タ依存辺を引く．

（b） 関数呼出し文に対して アルゴリズム ModifyVertex を用いてデータ依存辺を引き直す．その際，関数内で参照・定義する変数を呼出し文で参照・定義する変数として用いる．

（c） 呼出し文におけるデータ依存辺を global-in, global-out でのデータ依存辺とする．

6. 実行効率

6.1 実　　装

本アルゴリズムのうち，削除アルゴリズムおよび挿入アルゴリズムを文献[12]のシステムに組み込んだ．ソースコードはCで記述し，ユーザインタフェースには Tcl/Tk を用いた．本アルゴリズムを組み込むにあたり，フロー辺の追加や仕様変更，機能拡張などを行った．削除アルゴリズムは約1060行，挿入アルゴリズムは約570行である．また，システム全体では約13500行である．

削除，挿入・変更アルゴリズムの実行時間を表3に示す（SPARCstation 20, Memory 64 MB 上での実行時間）．但し，表3中の左側二つ（50行・100行）は単一関数，右側二つは複数関数のプログラムである．

これにより，本アルゴリズムによる PDG の変更がプログラムを変更した後に全体を再計算することに比べ，解析時間が大幅に削減されることがわかる．

6.2 計　算　量

PDG の再計算・更新にかかわる要素を表4に示す．集合演算にその要素数に比例する計算量が必要だとしたときの，PDG 再計算・本アルゴリズムの計算量は表5のようになる．

6.3 考　　察

PDG の再計算および更新アルゴリズムの時間計算量を示した．

本アルゴリズムの時間計算量が最悪となるのは PDG が完全グラフであり，かつ特殊節点に関する操作をすべての関数で行うときである．しかし，通常のプログラムでこのようになることはない．

また，一つの関数の大きさはプログラム全体の大きさとは独立で，ある定数以下であると仮定すると前定義節点の発見手続きは $O(1)$ で計算できる．

以上の仮定のもとでは，本アルゴリズムでは関数間の削除・挿入・修正は $O(V+E+S_t\cdot(G+L))$ で計算できる．一方，PDG 全体を再計算すると $O(S_t\cdot(G+L))$ になる．

PDG 再計算と本アルゴリズムでは解析の対象がそれぞれソースプログラム，PDG と異なるため，計算量の表現に用いられる変数が異なっている．本アルゴリズムでは関数境界を越える依存関係を PDG 再計算と同様に解析しており，表記上の計算量が大きくなる．

本アルゴリズムでは関数間の依存関係の変化を解析するために，関数で定義・参照する変数を収束するまで調べる．この部分の計算量は PDG 再計算と同じだけ必要である．しかし，PDG 再計算ではこの繰返しにおいてすべての節点でデータ依存辺の挿入を行っているが，本アルゴリズムでは節点の内容を調べるだけで，辺の挿入は行わない．

PDG 再計算のためにはプログラム全体の構文解析などが必要であるが，本アルゴリズムでは部分的な構文解析（必要ない場合もある）だけですみ，大部分の節点や辺の作成も必要ないため，現実的には PDG 再計算に比べて短い時間で解析できる．

また，本アルゴリズムのために PDG 中に保持しておく必要のある情報は関数ごとの定義・参照変数情報だけである．しかし，変更前に PDG を調べることによりこの情報も復元できるため，PDG のみあれば変更を行うことができる．これは，すべての節点で到達

表3 実行時間（単位は秒）
Table 3 Execution time (s).

	50行	100行	250行	430行
PDG 再計算	0.08	0.35	1.42	16.61
削除	0.01 以下	0.01 以下	0.01	0.07
挿入・変更	0.01 以下	0.01 以下	0.01	0.05

表4 PDG 再計算・更新にかかわる要素
Table 4 Component for recompute and incremental update of PDG.

P	手続きの総数
G	大域変数の総数
L	手続きの局所変数の最大値
S_i	手続き呼出しの総数
S_t	文の総数
V	PDG の節点の総数
E	PDG の有向辺の総数

表5 計　算　量
Table 5 Computational complexity.

PDG 再計算	$O(P\cdot S_t\cdot(G+L))$
単一関数内削除	$O((V+E)\cdot(G+L))$
単一関数内挿入	$O((V+E)\cdot(G+L))$
単一関数内修正	$O((V+E)\cdot(G+L))$
関数間削除・挿入・修正	$O(P\cdot S_t\cdot(G+L)+S_i\cdot G\cdot(V+E)(G+L))$

定義集合を保存しておくという文献[8]のアルゴリズムに比べ使用する作業領域面で効率が良い．

7. むすび

プログラムの部分的変更が行われた際に，PDGを部分的に変更する手法を提案した．また，既存のシステムに組み込みその高速実行性を確認した．我々のデバッグ支援システムはこの機能を組み込んだため，頻繁にソースプログラムの変更をしてもスライスを効率良く抽出できるようになり，よりインタラクティブにデバッグ作業を行えるようになった．

本手法は，今回実際に組み入れたシステムに限らず，PDGを利用しその一部が頻繁に変更される可能性のあるシステム使用時の作業効率を向上させることが期待される．

本論文ではPascal風言語を入力言語として扱ったが，2.4で示したPDGを用いて解析を行っていれば，他の手続き型言語にも適用可能である．

文　献

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

[2] P. Fritzson, T. Gyimothy, M. Kamkar, and N. Shahmehri, "Generalized algorithmic debugging and testing," Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, vol.26, no.6, pp.317–326, 1991.

[3] P. Fritzson, N. Shahmehri, and M. Kamkar, "Generalized algorithmic debugging and testing," ACM Letters on Programming Languages and Systems, vol.1, no.4, pp.303–322, 1992.

[4] 二村良彦，佐藤泰介，二木厚吉，玉木久夫，竹内彰一，安村通晃，古川康一，吉田紀彦，"プログラム変換，" chapter 4, pp.63–79, 共立出版，1987.

[5] B. Gallagher and R. Lyle, "Using program slicing in software maintenance," IEEE Trans. on Software Engineering, vol.17, no.8, pp.751–761, 1991.

[6] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," Proceedings of the 14th International Conference on Software Engineering, Dallas, Texas, Association for Computing Machinery, pp.392–411, 1992.

[7] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," ACM Trans. on Programming Languages and Systems, vol.12, no.1, pp.26–60, 1990.

[8] 鍛冶武志，"プログラムの部分的変更にともなう依存解析グラフの更新手法，" 大阪大学基礎工学部情報工学科特別研究報告，1996.

[9] R. Lyle and M. Weiser, "Automatic program bug location by program slicing," Proceedings 2nd International Conference on Computers and Applications, pp.877–883, 1987.

[10] L. Pollock and L. Soffa, "An incremental version of iterative data flow analysis," IEEE Transactions on Software Engineering, vol.15, no.12, pp.1537–1549, 1989.

[11] G. Ryder and C. Pall, "Incremental data-flow analysis algorithm," ACM Trans. on Programming Languages and Systems, vol.10, no.1, pp.1–50, 1988.

[12] 佐藤慎一，飯田　元，井上克郎，"プログラムの依存関係解析に基づくデバッグ支援システムの試作，" 情処学論，vol.37, no.4, pp.536–545, 1996.

[13] T. Shimomura, "Bug localization based on error–cause–chasing methods," Transactions of Information Processing Society of Japan, vol.34, no.3, pp.489–500, 1993.

[14] 下村隆夫，"プログラムスライシング技術と応用，" 共立出版，1995.

[15] 高谷暢之，"出力の制限情報を利用したプログラム簡素化手法の提案，" 修士論文，大阪大学基礎工学部情報工学科，1994.

[16] 植田良一，練　林，井上克郎，鳥居宏次，"再帰を含むプログラムのスライス計算法，" 信学論（D-I），vol.J78-D-I, no.1, pp.11–22, Jan. 1995.

[17] M. Weiser, "Program slicing," Proceedings of the Fifth International Conference on Software Engineering, pp.439–449, San Diego, CA, 1981.

付　録

1. アルゴリズムの正当性

本アルゴリズムによってデータ依存辺がPDG計算のアルゴリズムと同様に引かれることを示す．以下，ある文 s の入口における到達定義集合を $RD_{in}(s)$，出口における到達定義集合を $RD_{out}(s)$ で表す．

1.1 削　除

削除する節点 s で変数を定義していない場合，削除によって s 以降の文に与える影響はない．この場合，データ依存辺を新たに追加する必要はなく，s と関連する辺を削除すればよい．

削除する節点 s で変数 v を定義し，$RD_{in}(s)$ 中に到達定義 $\langle r, v\rangle$ が存在したとする．この場合，削除前の $RD_{out}(s)$ 中には $\langle r, v\rangle$ は存在せず，$\langle s, v\rangle$ が存在する．

s がなければ，s 以降，v が再び定義されるまで到達定義集合中には $\langle s, v\rangle$ は存在せず，$\langle r, v\rangle$ が存在する．そのため，v を参照する節点 t でのデータ依存辺は $s \xrightarrow{v} t$ の代わりに $r \xrightarrow{v} t$ が存在する．

アルゴリズム DELETEVERTEX では $r \in PreDef(s,v)$，$t \in target(s,v)$ に対し $r \xrightarrow{v} t$ を追加する．ここで r，t は上記の節点 r，t に相当す

る，また，s に関するデータ依存辺を削除するため，$s \xrightarrow{v} t$ の代わりに $r \xrightarrow{v} t$ が新たに引き直されている．

1.2 挿　　入

挿入する節点 s で変数 v を定義し，変数 u を参照していたとする．

変数 v に関して，挿入アルゴリズムは削除アルゴリズムの逆を行う，つまり $r \xrightarrow{v} t$ を $s \xrightarrow{v} t$ にすることで PDG の更新が行える．但し，挿入を行う段階で s に関するデータ依存辺が存在しないため，到達定義集合に関する情報を復元するのにいくつかの問題がある．

一つ目の問題は，変数 v に関する前定義節点 r，参照する節点 t としたときに $r \ldots t \ldots s$ という実行順序となるときである．このとき，s が t 以前に実行されない可能性がある．

二つ目の問題は，s の挿入によって r での v の定義が t に到達するかどうかがわからないことである．

これらを解決するため，アルゴリズム InsertVertex ではパス $s,\ldots,t$ が存在する場合だけ処理を行う．また，すべてのパス $r,\ldots,t$ に対して，パス中に v を定義する文が存在しているならば r での v の定義は t に到達しないとして，$r \xrightarrow{v} t$ を削除している．これにより，すべての場合において PDG 全体を再計算するのと同じ PDG が計算できる．

1.3 修　　正

文の修正アルゴリズムは削除・挿入アルゴリズムを組み合わせたものである．アルゴリズム ModifyVertex は変更によって定義・参照されなくなった変数に対して削除アルゴリズムを，新たに定義・参照されるようになった変数に対して挿入アルゴリズムを適用している．

1.4 関数境界を越える場合

まず，4.2〜4.4 のアルゴリズムによって関数内部の解析を行う．次に，変更による他の関数への影響を調べる．

変更によって関数内部で定義・参照する変数が変化する可能性があり，これらが変化すればこの関数の呼出し文で参照・定義する変数が異なってくる．そこで，変更を受けた関数を直接または間接に呼び出す可能性のある関数の集合に対して，$SuDEF$，$PoDEF$，$ImUSE$ を求め，これらの値が収束するまで計算を行う．この方法は PDG 計算アルゴリズムの関数間解析と同じ方法であるため，これによって PDG 全体を再計算するのと同じ PDG が計算できる．

（平成 9 年 3 月 12 日受付，7 月 25 日再受付）

高田　智規

平 7 阪大・基礎工・情報卒．平 9 同大大学院修士課程了．現在 NTT ヒューマンインターフェース研究所勤務．在学中，プログラムスライスの研究に従事．

佐藤　慎一

平 6 阪大・基礎工・情報卒．平 8 同大大学院修士課程了．同年 NTT データ通信株式会社技術開発本部入社．ソフトウェア工学の研究に従事．

井上　克郎　（正員）

昭 54 阪大・基礎工・情報卒．昭 59 同大大学院博士課程了．同年同大・基礎工・情報・助手．昭 59〜61 ハワイ大マノア校・情報工学科・助教授．平 1 阪大・基礎工・情報・講師．平 3 同学科・助教授．平 7 同学科・教授．工博．ソフトウェア工学の研究に従事．