Implementation of Static Slicing Tool Using COINS Compiler Infrastructure

Tomohiro UENO*, Hirohide HAGA**

(Received April 18, 2014)

Mutation analysis is a method to evaluate software test cases set quality. In mutation analysis, mutant programs are generated by injecting bugs into original program intentionally. The quality of test cases set is evaluated whether the bugs are detected or not by the test cases set. We can generate mutant program by applying mutation operators, which are rules for injecting bugs to original programs, to statements in original program. However, when we apply all mutation operators to all statements in original program, the number of generated mutant programs becomes huge. When the number of generated mutant programs becomes huge, generation time, compilation time and execution time of mutant programs become enormous. Thus, it is necessary to reduce the number of generated mutant programs for practical application of mutation analysis. This problem comes from applying all mutation operators to all statements in original program. Therefore, reducing the number of mutation operators or the number of statements in original program to which we apply mutation operators will solve this problem. The method to reduce the number of mutation operators has already been studied by Offutt et al. The method to reduce the number of statements may be implemented by selecting statement to apply the mutation operators. In order to select statements, we propose the impact factor of each statement. Impact factor represents the degree of impact of each statement. Using static program slicing is a promising candidate of impact factor representation. Computing static slicing can be implemented by using information obtained from compiler. Therefore implementing slicing tool by using compiler development tool reduces the development time and cost. In this article, we adopted the COINS compiler infrastructure to implement statics slicing tool.

Keywords: mutation analysis, static program slicing, COINS, compiler Infrastructure

キーワード：ミューテーション解析，静的プログラムスライシング，COINS，コンパイラ・インフラストラクチャ

COINS を用いた静的プログラムスライシングツールの実装

上野　智弘，芳賀　博英

1. はじめに

現在，様々な機器にソフトウェアが組み込まれ，利用されている．それに伴い，ソフトウェアには，高い品質が求められるようになった．そのためには十分なテストをする必要があるが，ソフトウェアの品質の高さを保証するためには，バグ検出能力の高いテストセットを使わなければならない [1].

テストセットを評価する手法の 1 つに，ミューテ

*Graduate School of Science and Engineering, Doshisha University, Kyoto
Telephone: +81-774-65-6979, E-mail:tueno@ishss10.doshisha.ac.jp
** Department of Intelligent Information Engineering and Sciences, Doshisha University, Kyoto
Telephone: +81-774-65-6978, E-mail:hhaga@mail.doshisha.ac.jp

ーション解析がある[2]．この手法では，オリジナルプログラムに意図的にバグを埋め込んだミュータントを生成し，そのミュータントをテストセットが検出できるかどうかによって，テストセットの品質を評価する．ミュータントを生成する規則をミューテーションオペレータと言う．ミュータントは，ミューテーションオペレータをオリジナルプログラムに適用することで生成できる．しかし，すべてのステートメントに対してすべてのミューテーションオペレータを適用すると，生成されるミュータントの数が膨大になるという問題がある．生成されるミュータントの数が膨大になると，ミュータントの生成時間，コンパイル時間，実行時間が膨大になるので，品質評価に影響がない範囲で，生成されるミュータントの数を減らすことが望ましい．

この問題は，すべてのステートメントに対してすべてのミューテーションオペレータを適用しているため発生する．従って，ミュータントの数を減らすには，ミューテーションオペレータを減らす，あるいは，ミューテーションオペレータを適用するステートメントを減らすことで解決できると考えられる．前者の方法は，すでに Offutt らによって研究されている[3]が，後者の方法についてはまだ検討がなされていない．後者の方法は，あるステートメントが他のステートメントにどのくらいの影響度を持っているかを測り，その影響度を基に，ミューテーションオペレータを適用するステートメントを減らすことで実現できる可能性がある．その影響度を測ることができると考えられる手法の 1 つが，静的プログラムスライシング[4]である．

本論文は，上記した問題を解決するために使用できる静的プログラムスライシングツールの実装についての報告であり，対象言語は C 言語である．

静的プログラムスライシングツールの実装に必要な情報は，コンパイラで必要な情報と共通するものが多いので，コンパイラを開発する際に利用できるツールを使えると，工数を減らすことができる．そこで，コンパイラ・インフラストラクチャである COINS[5]を用いて静的プログラムスライシングツールを実装した既存研究を参考にし，発展させた[6]．

2. 静的プログラムスライシング

2．1 スライシングの概要

あるステートメントの実行が影響を与える可能性があるすべてのステートメントの集合，あるいは，あるステートメントの実行に影響を与える可能性があるすべてのステートメントの集合を静的プログラムスライスといい，その静的プログラムスライスを求めることを，静的プログラムスライシングという．静的プログラムスライシングは，スライシング基準という基準を基に，プログラム依存グラフという有向グラフを辿ることで，静的スライスを求めることである[4]．スライシング基準は，どのステートメントに関して，あるいは，どのステートメントのどの変数に関して静的スライスを求めるかを決めるための基準であり，

「スライシング基準 C ＝（u，V）は
（1）u はプログラム内のステートメント
（2）V はプログラム内の変数の部分集合」
で定義される[4]．

あるステートメントの実行が影響を与える可能性があるすべてのステートメントの集合である静的スライスを求めることを，フォーワードスライシング（Fig. 1），あるステートメントの実行に影響を与える可能性があるすべてのステートメントの集合である静的スライスを求めることを，バックワードスライシング（Fig. 2）という．

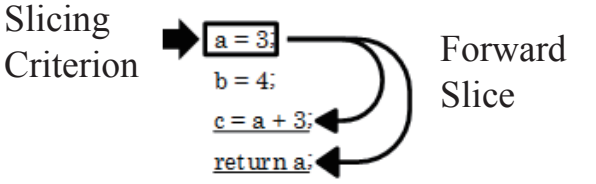


Fig. 1. Example of forward slice.

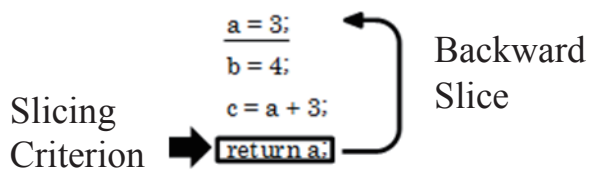


Fig. 2. Example of backward slice.

Fig. 3は，スライシング基準を1行目の変数“a”に設定してフォーワードスライシングを実行した例である．左側がフォーワードスライス，右側がプログラム依存グラフである．Fig. 3の左側の2行目は，抽出されないステートメント，1行目がスライシング基準のステートメントである．フォーワードスライシングは，プログラム依存グラフを順方向に辿ることで求められるので，Fig. 3のプログラム依存グラフを1行目のノードから順方向へ辿ると，Fig. 3のフォーワードスライスが抽出される．

Fig. 4は，スライシング基準を6行目の変数“a”に設定してバックワードスライシングを実行した例である．左側がバックワードスライス，右側がプログラム依存グラフである．Fig. 4の左側の2行目は，抽出されないステートメント，6行目がスライシング基準のステートメントである．バックワードスライシングは，プログラム依存グラフを逆方向に辿ることで求められるので，Fig. 4のプログラム依存グラフを6行目のノードから逆方向へ辿ると，Fig. 4のバックワードスライスが抽出される．

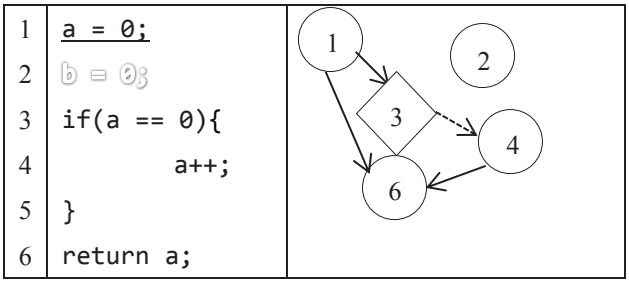


Fig. 3. Example of forward slicing.

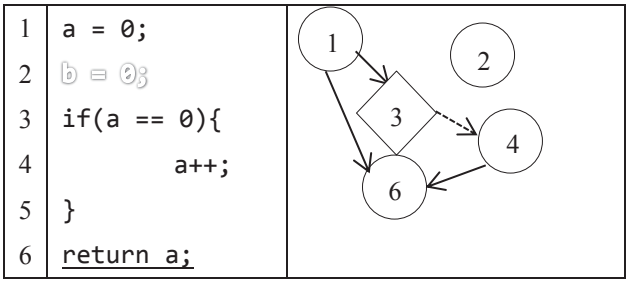


Fig. 4. Example of backward slicing.

静的プログラムスライシングの処理手順は，以下の通りである[6]．

1．制御依存グラフを作成する．
2．データ依存グラフを作成する．
3．1と2を統合して，プログラム依存グラフを作成する．
4．スライシング基準を基に，静的プログラムスライシングを行う．

2．2　静的スライスの定義

静的スライスは，あるステートメントの実行が影響を与える可能性があるすべてのステートメントの集合（フォーワードスライス），あるいは，あるステートメントの実行に影響を与える可能性があるすべてのステートメントの集合（バックワードスライス）である．

静的スライスの定義は次のように与えられる．

「プログラムPのスライシング基準がC＝(u，V)のとき，プログラムPの静的スライスは，以下の3つの条件を満たす，任意の実行可能なプログラムP’である．

1)　P’はPから0個以上のステートメントを削除することにより得られたものである．
2)　入力xに対してプログラムPが停止するならば，プログラムP’も停止する．

また，プログラムPに入力xを与えたときの実行系列をEX，プログラムP’に入力xを与えたときの実行系列をEX’とすると，

3)　すべての変数v∈Vに対して，実行系列EX’においてステートメント u を実行する直前における変数 v の値が，実行系列EXにおいてステートメント u を実行する直前における変数 v の値に等しい．

ただし，実行系列とは，ある入力を与えてプログラムを実行した場合，実行されたステートメントの列である[4]．」

2．3　制御依存グラフ

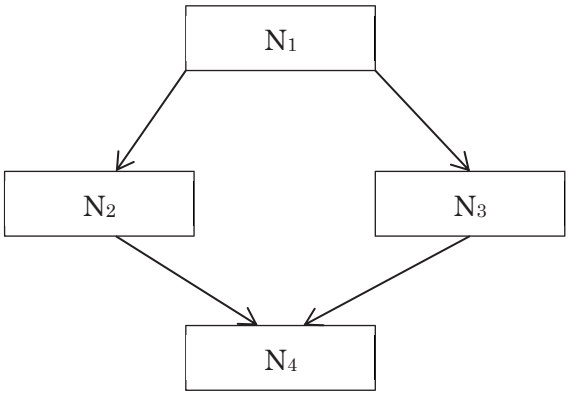
制御依存グラフは，プログラムの各ステートメントの間の制御の依存関係（制御依存）を示したグラフである．制御依存の定義は，

「X と Y というノードは制御フローグラフ上にあり， X からある辺を辿ると，その後必ず Y を通

るが，別の辺を辿ると Y を通らないとき，制御依存が X から Y にあるという [7].」
である．ただし，ノードとは，プログラム内のステートメント，制御フローグラフとは，以下の 3 つの条件を満たす有向グラフ G = (N，E，e)である [7].

(1) N はノードの集合で，ノードはプログラム内のステートメントからなる．
(2) E はエッジの集合で，エッジ(s，t)はステートメント s を実行した後，制御が直ちにステートメント t へ移る可能性があることを示す．
(3) e はノードで，プログラムを実行するときに最初に制御が移されるステートメントである．

制御依存が X から Y にある場合 CD(X，Y)と記述する．制御依存の例を Fig. 5 に示す．



(a)Control Flow Graph

Control Dependence in (a)
CD(N_1，N_2)
CD(N_1，N_3)

Fig. 5. Example of control dependence.

あるノードから辿り初めてそのノードの支配から外れるノードのことを支配辺境と言う. 制御依存グラフは，支配辺境を求めるアルゴリズムを用いて，すべてのノードの制御依存を求めることで，作成することができる. 支配辺境を求めるアルゴリズムを Fig. 6 に示す．ただし，支配とは，

「2 つのノード X，Y について，プログラムの入口から Y に達するどの路も，必ず X を通るとき，X は Y を支配する [7].」
である．X の支配辺境が Y のとき，DF(X) = {Y}と記述する．

ただし，この定義では，関数呼び出しを含む制御依存グラフは作成できない. 関数呼び出しを含む制御依存グラフを作るためには，上記に加えて，ソースプログラムの行番号を表す通常作成されるノードとは別に，Entry ノードを作成し，その Entry ノードから，関数内のステートメントに制御依存があるように辺を加える．ただし，他のステートメントから制御依存があるステートメントは，Entry ノードから制御依存があるように辺を加えない. そして，ある関数の関数呼び出しの行から，呼び出された関数の Entry ノードへ辺を加える．

2. 4 データ依存グラフ

データ依存グラフは，データ依存を示したグラフである．ここでデータ依存の定義とは，

「ステートメント X である変数を定義し，それが，その変数を参照するステートメント Y に到達する可能性があるとき，データ依存が X から Y にあるという [4].」
である．また，到達とは，

「ステートメント s が変数 x を定義し，かつ，ステートメント s からステートメント t に至るパス s，u1，u2，…，uk，t（k >= 0）が存在し，u1，u2，…，uk では変数 x を定義していない場合，ステートメント s の変数 x の定義がステートメント t に到達するという [6].」
である．データ依存グラフは，上述した定義に基づいて，すべてのノードのデータ依存を求めることで作成できる．データ依存が X から Y にある場合 DD(X，Y)と記述する．

配列を含むデータ依存グラフは，配列の添字式に変数を含むとき，配列全体に対する参照，あるいは，条件に依存する配列全体に対する定義とみなすことに注意する必要がある. それ以外は上記したものと同じである．

2. 5 プログラム依存グラフ

プログラム依存グラフの定義は，

「ソースプログラムの要素間の依存関係を有向グ

```
for each X From bottom to top
        DF(X) = φ
        for each Y ∈ succ(X) do
                if IDOM(Y) ≠ X then Add Y to DF(X)
        end for
        for each Z ∈ child(X) do
                for each Y ∈ DF(Z) do
                        if IDOM(Y) ≠ X then Add Y to DF(X)
                end for
        end for
end for

IDOM(Y)：Basic blocks that immediately dominated Y
succ(X)：Basic blocks that is successor of X
child(X)：In dominator tree, basic blocks that is children of X
```

Fig. 6. Algorism to calculate a dominance frontier.

ラフ化したものである[6].」

本論文でのプログラム依存グラフは，参考論文[6]と同様に，有向グラフのノードがステートメント，エッジが制御依存とデータ依存に相当し，制御依存グラフとデータ依存グラフの 2 つを統合して，作成できるものである．ソースプログラムに，関数呼び出しを含むプログラム依存グラフも同様である．

3. COINS

COINS コンパイラ・インフラストラクチャは，コンパイラを構成する基本機能のモジュールをすべて備え，それらの組み合わせを変えたり，一部のモジュールを新たに開発したりするだけで，新しいコンパイラを実現することができるコンパイラ基盤である[5]. それらのモジュールはすべて Java 言語で記述されている．COINS の構成は Fig. 7 のようになっており，線で囲まれた範囲が COINS の機能である．Fig. 7 のように，高水準共通中間表現 HIR(Higher-level Intermediate Representation)と低水準中間表現 LIR(Lower-level Intermediate Representation)を中核として，それらを扱う各種のモジュールからなっている．モジュールは，C・Fortran 言語解析, HIR→C, 基本最適化, HIR→LIR, ループ並列化，SMP(Symmetric Multi-Processor)並列化，LIR→C，SSA(Single Static Assignment)最適化，命令選択・レジスタ割付け・命令スケジュール・コード生成，SIMD 並列化がある．言語解析モジュールは，ソース言語プログラムを HIR に変換するモジュールである．HIR→C，HIR→LIR，LIR→C モジュールは，それぞれ，HIR から C へ，HIR から LIR へ，LIR から C へ変換するモジュールである．基本最適化，ループ並列化，SMP 並列化は，HIR に対して，最適化と並列化を行う．SSA 最適化，SIMD 並列化は，LIR に対して，最適化と並列化を行う. 新しい言語のコンパイラを開発するためには，新しい言語を解析し HIR に変換するフロントエンド部分を開発すれば良い.

次に，本論文で実装した静的プログラムスライシングツールで必要であった COINS から得られる情報を，(1)プログラムのすべての作成過程で必要であった情報，(2)制御依存グラフ作成で必要であった情報，(3)データ依存グラフ作成で必要であった情報の 3 つに分けて，順に説明していく．

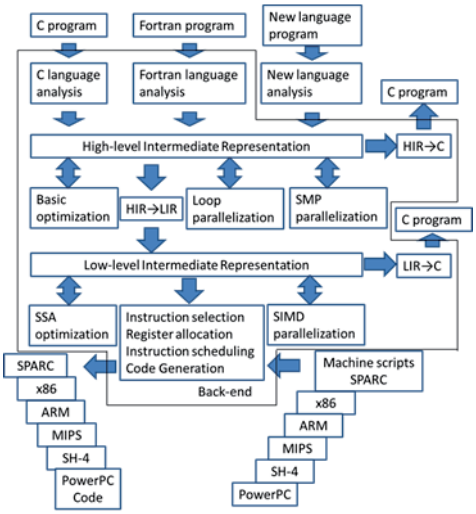


Fig. 7. Structure of COINS[5].

(1) プログラムのすべての作成過程で必要であった情報は，HIR と基本ブロックの 2 つである．HIR は，COINS で用いられているソースプログラムを変換した高水準中間表現であり，演算式，代入文，if 文，ループ文，整数型，浮動小数点型，配列などの情報を表現した抽象構文木である．ただし，HIR は処理操作を中心として表現するものであるため，宣言文の情報は表現していない．HIR からは，HIR のある部分がソースプログラムのどの行に対応しているかという情報と，プログラムポイントという情報を取得できる．プログラムポイントは，ソースプログラムでいう行に相当するものである．一方基本ブロックは，分岐を全く含まない連なったコードの集まりである．COINS では，関数ごとに基本ブロックを取得できる．

(2) 制御依存グラフ作成で必要であった情報は，制御フロー情報である．ある基本ブロックの親の基本ブロック番号と子の基本ブロック番号，逆転制御フローグラフ上である基本ブロックを支配する基本ブロック番号，直接支配する基本ブロック番号，逆支配木上である基本ブロックの子の基本ブロック番号などがある．これらの情報は，制御フローグラフから求めることができる情報である．

(3) データ依存グラフ作成で必要であった情報は，データフロー情報である．データフロー情報は，基本ブロックの中でどの変数が定義・参照されているかという情報や，ある基本ブロックの中で定義された変数が，どの基本ブロックまで到達するかという情報などである．

4. 実装

実装した静的プログラムスライシングツールの構成を Fig. 8 に示す．

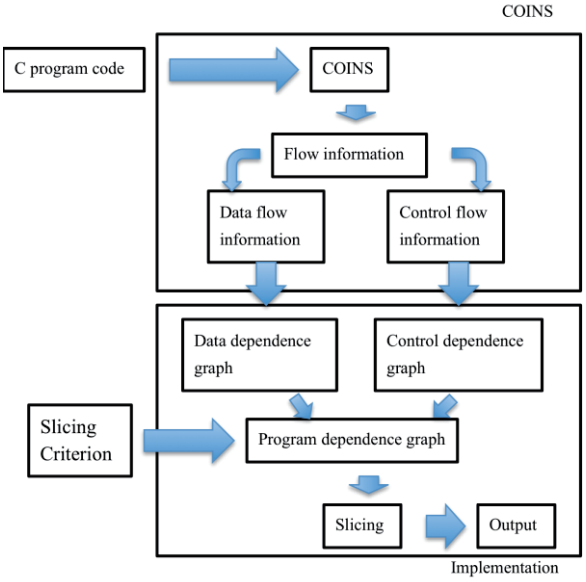


Fig. 8. Software architecture of slicing tool[3].

静的プログラムスライシングは，基本ブロック単位ではなく，ステートメント単位であるが，COINS から取得できる情報は，主に，基本ブロック単位である．従って依存グラフの作成前に，HIR のどこからどこまでが 1 ステートメントかをプログラムポイントで判断できるように，プログラムポイントを 1 ステートメントごとに保存している．また COINS から取得できる情報は関数単位であるため，制御依存グラフ，データ依存グラフの作成の段階では，関数呼び出しを考慮せず，プログラム依存グラフの作成の段階で，必要なノードや辺を追加する．この段階で使用した主な COINS のメソッドを Table 1 に示す．

Table 1. Methods in COINS used for global structure[8)]

COINS method name	
getBBlock (int pBlockNumber);	Get the basic block whose block number is pBlockNumber
getIndex();	Get the program point assigned to this node

4．1　制御依存グラフの生成

制御依存グラフは，2．3 節で説明した方法で作成する．最初に，支配辺境のアルゴリズムに必要な情報を COINS から取得し，次に COINS から取得した情報を支配辺境のアルゴリズムに入力する．そうすることで，基本ブロック単位の制御依存が出力される．最後に，基本ブロック単位ではなく，ステートメント単位の制御依存が必要であるから，プログラムポイントを基に，ステートメント単位の制御依存を求め，制御依存グラフを作成する．この段階で使用した主な COINS のメソッドを Table 2 に示す．

Table 2. Methods for constructing control dependency graph[8)]

COINS method name	
getPredList();	Get the predecessor list of this basic block
getPost DominatedChildren();	Get the list of basic blocks that is children of this block in dominator tree of reverse control flow graph
getImmediate PostDominator();	Get a basic block dominated by this block in dominator tree of reverse control flow graph

4．2　データ依存グラフの生成

データ依存グラフは，配列以外と配列に分け，データ依存を求めて作成する．配列以外は COINS からステートメント単位のデータ依存を取得できるのでそのまま利用する．配列は 2．3 節で説明した定義に沿ってデータ依存を求める．簡単に説明すると，最初に，HIR から配列が定義，参照されている変数名，添字式，プログラムポイントを求める．次に，配列が参照されているプログラムポイントと同じ基本ブロックから制御フローグラフを逆向きに辿り，配列が定義しているプログラムポイントを見つけると，そこから辺を加える．それを繰り返すことで配列のデータ依存が求めることができる．プログラムポイントを基準に求めているので，基本ブロック単位ではなく，ステートメント単位で求めることができる．このように，データ依存を作成している．この段階で使用した主な COINS のメソッドを Table 3 に示す．

Table 3. Methods for constructing data dependency graph[8)]

COINS method name	
getChild1();	Get the first child of this node
getChild2();	Get the second child of this node
getParent();	Get the parent of this node
getSym();	Get the symbol represented by this node
getType();	Get the type attached to this node
getUseDefChain (IR pUseNode);	Get UseDefChain having pUseDefNode as its use node
getDefList();	Get the list of Def nodes for this UDChain
getUseNode();	Get the Use node for this UDChian

4．3　プログラム依存グラフの生成

プログラム依存グラフは，2．3 節，2．4 節，2．5 節で説明した方法で作成する．ただし，関数呼び出しを含むプログラム依存グラフは，Table 4 のようなノードを追加し，関数の出入り口を明確にすることで，スライシング部分の実装を簡単にしている．

Table 4. Added nodes in PDG[4)]

Node name	
“Function name －E” node	There is control dependence from this node to statements in a function and from function call nodes to this node
“Function name －O” node	There is data dependence from return value nodes to this node and from this node to a function call node
“Function name －P” node	This node is to gather parameters in a function , children nodes of this node are “Function name -N number” nodes , and there is data dependence from a function call nodes to this node
“Function name －N number” node	This number is a parameter number, a parent node of this nodes is “function name -P” node , and there is data dependence from this node to nodes that refer to parameters

プログラム依存グラフの例を List 1 と Fig. 7 に示す．List 1 が Fig. 9 のソースプログラムである．ただし，実線が制御依存，点線がデータ依存を表す．Fig. 5 の制御依存とデータ依存について，それぞれ 1 つずつ説明する．11 行目を表すノードから F-E ノードへの制御依存の辺は，関数呼び出しを含むプログラム依存グラフで追加される関数を呼び出した行から呼ばれた関数の Entry ノードへの辺である．11 行目を表すノードから 12 行目を表すノードへのデータ依存の辺は，2．4 節で説明したように，11 行目で変数 c が定義され，12 行目で変数 c が参照されるまで，変数 c が再定義されていないため，存在する．

List 1. Sample source code

```
1  int F(int x, int y){
2          return x * y;
3  }
4
5
6  int main()
7  {
8          int a, b, c;
9          a = 0;
10         b = 2;
11         c = F(a, b);
12         return c;
13 }
```

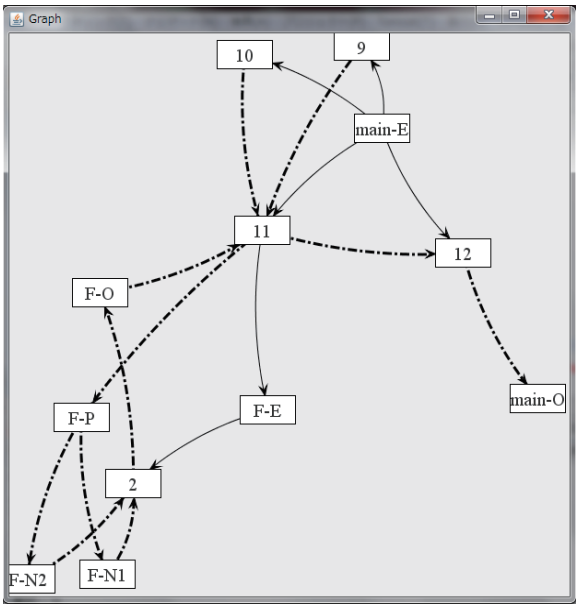


Fig. 9. PDG of List.1.

2．4 節のデータ依存の説明では，9 行目と 10 行目から 2 行目へデータ依存の辺が存在するはずだが，Fig. 7 では，それが存在していない．これは，関数ごとのプログラム依存グラフの統合を簡単にするため，本来のデータ依存の辺にしていない．しかし，スライシング部分の実装で，静的スライスが本来のものになるよう実装するので問題はない．

4．4　スライシング部分の実装

実装した静的プログラムスライシングツールは，

プログラム依存グラフの辺を順方向に辿るとバックワードスライシング，逆方向に辿るとフォーワードスライシングになる．ノードの探索は，深さ優先探索を用いている．ソースプログラムの行を示し，その行が関数呼び出しでない通常のノードでは，単純に辺を辿る．関数呼び出しを含むスライシングの場合，単純に辺を辿ると，設定したスライシング基準によっては，間違った静的スライスが求まる可能性がある．この問題は，すべての関数のプログラム依存グラフを統合してからスライシングを行うため，ノードの探索が呼び出された関数から関数を呼び出したノードへ正しく戻らないため生じる．解決するため，関数を呼び出したノードを保存し，呼び出された関数からノードの探索が戻ってくるとき，保存したノードと戻ってきたノードの探索が一致するときのみ，ノードの探索を続けるようにしている．ノードを保存するタイミングは，バックワードスライシングの場合，関数名－O ノードが次の探索ノードのとき，フォーワードスライシングの場合は，関数名－E ノードか関数名－P ノードが次の探索ノードのときである．再帰関数を含む場合を想定して，無限ループを起こさないように，1 回探索したノードは，探索しないようにしている．

5. 評価実験

評価実験の目的は，C 言語の基本的な機能である条件文，ループ文，配列，関数呼び出しを用いたプログラムに実装した静的プログラムスライシングツールを適用して，正しい結果を出力するかどうかを確かめることである．

5. 1 実験内容

2 つのプログラムで評価実験を行う．1 つ目は，月齢を計算するプログラム MoonAge である．このプログラムは，関数呼び出しがあり，行が 100 行以上，if 文 for 文 while 文が含まれている．2 つ目は，上記にはない switch 文と配列を含むプログラム Other である．MoonAge と Other のプログラムは，条件文，ループ文，配列，関数呼び出しを含むので，この 2 つのプログラムを評価実験に用いた．

6. 考察

実験結果と手作業で静的プログラムスライシングの定義に沿って静的スライシングを行った結果が一致したので，配列，if 文，for 文，while 文，switch 文，関数呼び出しを含むプログラムで，実装した静的プログラムスライシングツールは正しく動作することが確認できた．これは，COINS の生成する情報を適切に利用できたことを示している．

課題として，do-while 文，配列使用の一部，ポインタと大域変数，関数呼び出しを同じ行に 2 つ以上記述できない，COINS がソースプログラムを正しく解析できない，もしくは，解析しないことを含む場合に，静的プログラムスライシングツールが正しく動作しないことがあげられる．do-while 文は，COINS が do-while 文の入口である do の部分の行番号を HIR 上に表示するが，条件判定の while の部分の行番号を HIR 上に表示しない．従って，制御依存の辺をソースプログラムのどの行から引けばいいのかわからないため，対応していない．配列使用の一部というのは，例えば，int a[2]で宣言した 1 次元配列を，F(a)のように，ただの変数として記述することである．これは通常の配列使用の HIR 上での表現と異なるため，対応していない．これは，HIR 上での表現の変化に対応できれば，解決できると考えられるので，今後の課題である．ポインタと大域変数は，どの関数でも共通に定義・参照されるため，データ依存が複雑になる．従って，対応していない．関数呼び出しを同じ行に 2 つ以上記述できない理由は，スライシング部分で，どの関数の何番目の引数の変数からスライシングを行うかを判別できていないからである．これは，スライシング部分を改良すれば対応できると考えられるので，今後の課題である．COINS がソースプログラムを正しく解析できない，もしくは，解析しないことというのは，関数呼び出しの引数に使用した変数にその関数呼び出しの返り値を代入する，変数の宣言と同時に定義できない，同じ行に複数のステートメントを記述できないである．これらは，COINS がソースプログラムを正しく解析できない，もしくは，解析しないので対応していない．

課題はいくつか存在するが，C 言語の主な機能に対応できているので，実装した静的プログラムスライシングツールは，静的プログラムスライシングツールとして，一定の評価は得られると考えられる．

7. おわりに

本論文では，静的スライスを求める静的プログラムスライシングツールを実装した．本論文の目的は，ミューテーション解析の問題を解決するために使用できる静的プログラムスライシングツールを実装することである．静的プログラムスライシングツールの実装は，コンパイラを利用できると工数を減らすことができる．そこで，コンパイラ・インフラストラクチャである COINS を用いて静的プログラムスライシングツールを実装した既存研究を参考にし，発展させた．具体的には，参考にした既存研究では対応していなかった配列と関数呼び出しに対応した．考察で述べたが，この静的プログラムスライシングツールには，do-while 文に対応できていないなど，課題はいくつか存在する．しかし，実装した静的プログラムスライシングツールは，C 言語の基本的な機能に対応できている．従って，本論文の目的である，ミューテーション解析の問題点を解決するために使用できる静的プログラムスライシングツールとして，一定の評価は得られると考えられる．

参考文献

1) 上芝貴也，“ミューテーションテストにおける等価ミュータント検出機能の実装”，(2012).
2) A. Mathur， Foundation of Software Testing，(Addison-Wesley Professional， Boston， 2008)，p.502-651.
3) A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, C. Zapf, “An Experimental Determination of Sufficient Mutant Operators”, ACM Transactions on Software Engineering and Methodology，**5**[2]，99-118(1996).
4) 下村隆夫，プログラムスライシング技術と応用，(共立出版株式会社，東京都，1995)，p.1-23.
5) coins 開発グループ，COINS コンパイラ・インフラストラクチャ，http://coins-compiler.sourceforge.jp/，参照日時(2014/02/03).
6) 溝渕裕司，中谷俊晴，佐々政孝，“コンパイラ・インフラストラクチャを用いた静的プログラムスライシングツール”，http://www.is.titech.ac.jp/~sassa/papers-written/mizobuchi-nakaya-sassa-jssst03.pdf，参照日時(2014/05/22).
7) 中田育男，コンパイラの構成と最適化第 2 版，(朝倉書店，東京都，2009)，p.336-353.
8) COINS Compiler Infrastructure，http://sourceforge.jp/projects/coins-compiler/，参照日時(2014/02/10).