社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.
SS98-7 (1998-05)

関数呼び出し履歴を利用したプログラムスライス抽出技法の提案と実現

地平 稔† 西松 顯‡ 楠本 真二‡ 井上 克郎‡

† 奈良先端科学技術大学院大学情報科学研究科

E-mail: {minoru-j@is.aist-nara.ac.jp}

‡ 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 基礎工学研究科

Phone: 06-850-6571 Fax: 06-850-6574
E-mail: {a-nisimt, kusumoto, inoue}@ics.es.osaka-u.ac.jp

あらまし 本研究では，プログラムを実行して得られる動的な情報を静的スライスに組み合わせることで，動的スライスよりも小さな時間的・空間的コストで静的スライスの一部を抽出する技法を提案する．この技法では，動的な情報として実行された関数呼出し文の履歴情報を利用し，プログラム中から実行されなかった文の特定を行ない，実行されなかった文をプログラム依存グラフ (Program Dependence Graph) 上から除いてスライスを計算する．この技法を用いることで，既存の動的情報を用いるよりも小さなコストで，静的スライスよりも小さなスライスが得られる．

キーワード プログラムスライス，プログラム依存グラフ，関数呼び出し履歴

Program Slicing Technique Using Function Call History

Minoru Jihira†, Akira Nishimatsu‡, Shinji Kusumoto‡ and Katsuro Inoue‡

† Graduate School of Information Science, Nara Institute of Science and Technology

E-mail: {minoru-j@is.aist-nara.ac.jp}

‡ Graduate School of Engineering Science, Osaka University

1-3 Machikaneyama-cho, Toyonaka-shi, Osaka 560-8531, Japan

Phone: +81-6-850-6571 Fax: +81-6-850-6574
E-mail: {a-nisimt, kusumoto, inoue}@ics.es.osaka-u.ac.jp

Abstract

In this paper, we propose a new program slice, called CH Slice(Call-History restriction Slice), by combining static slice with the dynamic information obtained from the program excution. At first, we translate the program(source code) into program dependence graph(called PDG). Next, we execute the program and then preserve the function call history(CH) during the program execution, Finally, we compute CH slice with PDG and CH, as specifying the non-executed statement. Consequently, CH Slice is smaller and more precise than dynamic slice. The cost of construction of CH Slice is the same as static slice.

Key words Program Slice, Program Dependence Graph, Function Call History

1 まえがき

ソフトウェアシステムの大規模化，複雑化にともないソフトウェア開発における生産性，及び，品質向上の実現はソフトウェア工学における研究の主要な目標に位置付けられてきている．ソフトウェアの品質や生産性を向上させるためには，開発されたソフトウェアプロダクトだけでなく，その開発プロセスを対象として作業の改善を行うことが必要である．

一方，現実のソフトウェアプロジェクトではソフトウェア開発コストの50～80%をテスト工程に費やしているという報告がある．従って，ソフトウェア開発プロセスの改善を行うためには，テスト工程の改善を行うのが効果的である．テスト工程は故障の検出(テスト)と故障の原因であるフォールトの修正(デバッグ)の2つの作業から構成される．一般に，フォールト位置の特定がデバッグにおいて最も時間がかかる作業であると言われており [8, 9]，フォールトの位置を効率よく特定する方法の開発が重要となっている．

フォールトの位置を効率良く特定するための方法の一つとして，プログラムスライス技法(Program Slicing)を利用した手法が提案されている [15]．プログラムスライス技法はプログラム内のある文の実行に影響を与える全ての文を抽出する技術であり，抽出された文の集合をスライスと呼ぶ．プログラムスライス技法を利用するデバッグでは値の誤っている変数に対して，全プログラムにわたってスライスを求める事でデバッグ時に参照する範囲を限定し，そのスライスの中でフォールトとなっている文を捜し出す事で効率良いデバッグ作業が行なえる．

プログラムスライス技法はMark Weiser [15]によって提案された当初はプログラムのデバッグを支援するために使われていたが，現在では，さまざまな種類のプログラムスライスが提案され，デバッグだけでなくテストや保守，プログラム合成などにも利用されている．デバッグに利用する事を目的として，これまで静的スライス(static slice)，動的スライス(dynamic slice) [2, 7]，混合スライス(hybrid slice) [5]などのスライスが提案されている．しかし，上記のスライスは，不必要な文がスライスとして抽出されたり，計算するために非常に時間的・空間的コストが必要となる．

本研究では，プログラムを実行して得られる動的な情報を静的スライスに組み合わせることで，動的スライスよりも小さな時間的・空間的コストで静的スライスの一部を抽出する技法を提案する．この技法により抽出されたスライスを，CH スライス(Call-History reduction Slice)と呼ぶ．この技法では，動的な情報として実行された関数呼出し文の履歴(Function Call History) 情報を利用し，プログラム中から実行されなかった文の特定を行ない，実行されなかった文をプログラム依存グラフ(Program Dependence Graph，以降PDG) 上から除いてスライスを計算する．この技法を用いることで，既存の動的情報を用いるよりも小さなコストで，静的スライスよりも小さなスライスを計算できる．

以降，2.では研究の背景について述べる．3.では提案するプログラムスライス抽出アルゴリズムを述べる．4.では静的スライス，動的スライスとの比較について実行結果から分かったことを述べる．最後に，5.でまとめと今後の課題について述べる．

2 背景

ソフトウェア開発プロセスにおけるテスト工程は，

(1) 故障の検出(テスト)

(2) 故障の原因であるフォールトの位置特定と修正(デバッグ)

の2つの作業から構成される．フォールトの発見から除去に至る上記のプロセスにおけるスライスの利用を考える．(1)のテスト作業においては，故障の検出を行うためにプログラムを実行する．(2)のデバッグ作業においては，(1)で検出された故障箇所に対するスライスを計算し，スライス内のみをデバッグの対象とする事で，効率の良いデバッグ作業が行える．

スライスを計算するアルゴリズムは，大きく分けると以下の2つのステップからなる．

(Step1) プログラムの文間の依存関係を解析し，プログラム表現(Program Representation)の構築．

(Step2) Step1で構築されたプログラム表現からスライスを計算．

静的スライスは，Step1のプログラム(ソースコード)の文間の依存関係を解析することで，プログラム表現としてPDGを構築し，PDGからスライスを計算する方法が，提案されている．静的スライスは，全ての入力を考慮している為に，特定の入力で検出されるフォールトの場合には，フォールトに関係ない文まで抽出される事が多い．

動的スライスは，Step1で，プログラムを実行させ，実行した文間の依存関係を解析することで，プログラム表現としてDDG (Dynamic Dependence Graph)を構築し，DDGからスライスを計算する方法が提案されている．動的スライスは，特定の入力に関したプログラム表現を構築するので，静的スライスのように，フォールトに関係のない文がスライスに含まれるような事はないが，Step1でのDDG構築のために，プログラム実行時に各文で動的情報を記憶するため，非常に時間を

要する．さらに，PDGはソースコード中の文の数に比例するが，DDGは，実行された文の数に比例するため，PDGに比べDDGは非常に大きくなる事が多く，Step2においてDDGからスライスを計算するのに非常に時間を要するという問題点もある．

そこで，今回我々は，静的スライスを計算するために必要な時間・空間的コストで，動的スライスのように正確なスライスを得るスライス計算アルゴリズムを提案する．

3 CH スライス

今回我々が提案するCH スライスは，プログラムを実行したときに得られる動的情報を利用し，スライス計算時に動的情報を利用して，PDG上で実行されていない文を特定し，その文に関する依存関係を取り除く事で，静的スライスと同程度の時間・空間的コストで，動的スライスと同程度に正確なスライスとなる．今回提案するスライス計算アルゴリズムは以下のステップからなる．

(Step1) プログラム(ソースコード)の文間の依存関係を解析し，PDGの構築．

(Step2) プログラムの実行時に動的情報を保存．

(Step3) Step2で得られた動的情報とPDGからCH スライスを計算．

以降，上記の各Stepごとに，詳しい説明を行なう．

3.1 PDGの構築(Step1)

PDGとは，プログラム(ソースコード)内の文間の依存関係を表す有向グラフであり，その節点は，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各文を表し，独立な節点番号が与えられる．また，その有向辺は二つの節点の間のデータ依存(Data Dependence，DD)関係および，制御依存(Control Dependence，CD)関係を表す(図2参照)．今回提案するスライスでは動的情報(関数呼び出し文の履歴)から，PDGで実行されていない文を特定するために，PDGに新たに実行依存(Execution Dependence,ED) 関係を導入する．データ依存関係，制御依存関係，実行依存関係は次のように計算する．

- データ依存関係
  データ依存関係は，各頂点の到達定義集合(Reaching Definitions，以下RD) [1]を求めることによって得られる．PDG上でのある頂点tのRDとは，以下の条件を満たす全ての変数vと頂点sとの組$\langle v,s\rangle$の集合である．

  (1) プログラム中の文sで変数vを定義している．

  (2) プログラム中の二つの文sとtの間にvを必ず定義するような文がない．

  ことを示している．tのRDに$\langle v,s\rangle$が含まれ，かつtがvを参照する時，sからtへのデータ依存関係があるという．

- 制御依存関係
  命令sから命令tへの制御依存関係とは，命令sが条件判定文であり，その条件判定文の結果により命令tの実行の有無が決まる関係である．制御依存関係は，プログラム構造から容易に計算できる．

- 実行依存関係
  命令sが関数呼び出し文，または手続き呼び出し文のときに，命令sが実行されなかった場合，命令tが必ず実行されないとき，命令tは命令sに実行依存関係$\mathrm{ED}(s,t)$があるという．実行依存関係は，プログラムのネスト構造から容易に計算できる．

手続きの境界を越えて，スライスを計算できるようにするために，表1に挙げるような，プログラム中の文とは直接対応しない特殊節点を用意する．例えば，大域変数gと，仮引数pを取る関数fを持つプログラムには図1にあるように，exit節点f-*exit*，in節点f_g-*in*，out節点f_g-*out*，引数節点f_p-*par*が存在する．

表 1: 特殊節点

特殊節点		表記例
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-*exit*
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-*in*
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-*out*
引数節点	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-*par*

3.2 動的情報の保存(Step2)

今回提案する技法では，動的情報として関数呼出し文の実行履歴を用いる．関数呼出し文の実行履歴として，ある入力でプログラムを実行したときに，実行された関数呼出し文に対応するPDG上の節点の節点番号を保存する．この関数呼出し文の実行履歴をCHとする．CHは以下のようにして得られる集合である．

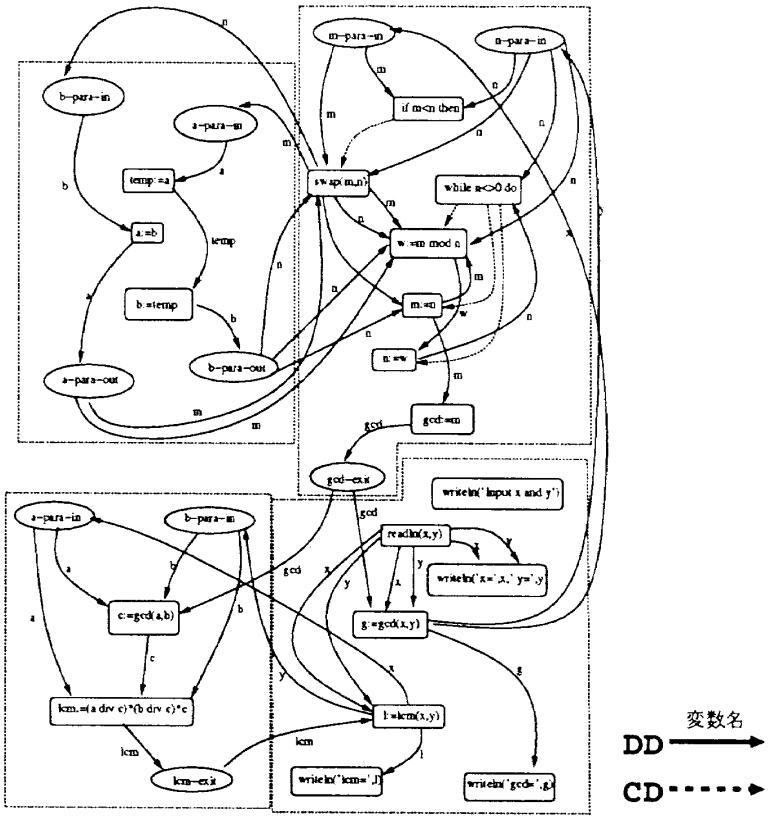

図 2: PDG の例

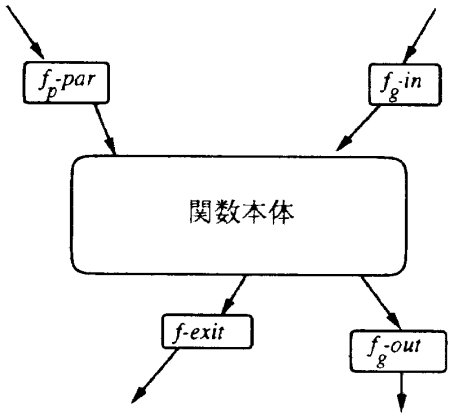

図 1: 関数 f に対する PDG の概略

(1) プログラム実行開始時に，CH ← ϕ に初期化

(2) if (実行している文 == 関数呼び出し文 or 手続き呼び出し文)
then CH ← CH ∪ {対応する PDG 上の節点番号 }

(3) if (プログラム終了)
then 終了
else (2) へ戻る

3.3 CH スライス計算 (Step3)

スライス計算時に，関数呼出し文の実行履歴と実行依存関係を用いて，実行されなかった文を特定する．文 t が実行依存している関数呼び出し履歴を CED(t) と書き，これを以下のように定義する．

$$\mathrm{CED}(t) = \{\ \text{関数呼出し文}\ s |\ \mathrm{ED}(s,t)\ \}$$

この CED(t) に含まれる文のうち一文でも実行されていない文がある場合，文 t は実行されていない．

$$\mathrm{CED}(t) \not\subseteq \mathrm{CH}$$

であるとき，文 t は実行されていないことがわかる．

この CED の情報は動的なものではないので，PDG を作成する際に，PDG の各節点に CED の情報を与えておくことができる．これにより，異なる入力による実行の場合であっても，関数呼び出し文の履歴の情報だけ新しく記憶すれば，実行されなかった文の特定が行なえる．このため PDG は一度作成すると変更の必要がなく，スタティックスライスと比べて時間的コストの負担が大きく増えることはない．

PDG を G，G に含まれる全ての辺の集合を E とすると，プログラム内の文 S の変数 v に関する CH スライス

を表す節点の集合Vは以下に示すような手順で計算できる．ただし，n_sはSに対応する節点である．

(1) $V \leftarrow \{n_s\}$

(2) $N \leftarrow \{n \mid n \xrightarrow{v} n_s\} \cup \{m \mid m \dashrightarrow n_s\}$

(3) $N \neq \phi$の間以下の動作を繰り返す．

(a) $n \in N$を一つ選ぶ．

(b) $N \leftarrow N - \{n\}$

(c) $\mathrm{CED}(n) \not\subseteq \mathrm{CH}$ なら(a)に戻る．

(d) $V \leftarrow V \cup \{n\}$

(e) nがin節点でも引数節点でもないなら次の操作をする
$N \leftarrow N \cup \{m | m \notin V \wedge (m \longrightarrow n \vee m \dashrightarrow n)\}$
(ただし$m \longrightarrow n$はmからnへのDD関係辺，$m \dashrightarrow n$はmからnへのCD関係辺を表す)

3.4 静的スライスとの時間的コストの比較

静的スライスとの時間的コストの比較について述べる．静的スライスは，ある入力に対する実行において実行されない文の依存関係まで，スライス計算の対象となる．一方，CHスライスは，実行されない文をスライス計算の対象外とする事で，静的スライスより正確なスライスとなる．静的スライスとの時間的コストを考えると，アルゴリズムの3.3の(c)の実行されなかった文を特定する部分が，静的スライスでは行なわれない部分であり，静的スライスよりもコストが増える部分である．この部分の計算にかかるコストについて述べる．この部分では集合$\mathrm{CED}(l)$の要素が全て，集合CHに含まれるときlは実行されたものとし，そうでないとき実行されなかったという判定を行なっている．この集合の比較にかかるコストがスタティックスライスよりも余分にかかるコストである．$\mathrm{CED}(l)$，CHはともにプログラム中に存在する関数呼出し文の一部であるため，これらの集合の比較にかかるコストは大きくはならない．

3.5 実例

上記の手順に従って図3のプログラムにたいしてCHスライスを実際に計算する．

- 入力に$d = 0$を与えて実行したときの，図3の19行目の文`writeln(a)`に対応するPDGの節点の変数aに関するCHスライスを図4の左に示す．
- 入力に$d = 1$を与えて実行したときの，図3の19行目の文`writeln(a)`に対応するPDGの節点の変数aに関するCHスライスを図4の右に示す．

4 分析・評価

CHスライスの評価を行なうために，既に静的スライス，動的スライスを抽出する機能を持った開発支援システム[11][10]にCHスライスを計算する機能を追加し，計算結果及び計算にかかる時間等の比較を行なった．

この開発支援システムが対象とする言語は，以下のような仕様のPascalのサブセットとする．この言語には，文として条件文(if文)，代入文，繰り返し文(while文)，入力文(readln文)，出力文(writeln文)，手続き呼び出し文，複合文(begin-end)がある．変数の型としてはスカラ型のみでポインタ型はない．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．

開発支援システムにCHスライスを計算する機能を追加し，実際に，100行，400行の簡単なプログラムに対して，静的スライス，動的スライスとCHスライスの計算を行ない比較評価を行なった．比較を行なったのは，スライスとして抽出されるサイズ(表2)，プログラムの解析時間(表3)，プログラムの実行時間(表4)，スライスの計算時間(表5)である．

表 2: スライスのサイズの比較

	100 行	400 行
静的スライス	27 文	175 文
動的スライス	14 文	139 文
CH スライス	22 文	156 文

表 3: プログラムの解析時間の比較

(P5-100MHz,memory 48MB)
(単位：ミリ秒)

	100 行	400 行
静的スライス	74	7603
CH スライス	76	7961

```
 1  program double_or_tripple(input,output);
 2  var a,b,c,d : integer;
 3  procedure nijyou(var a,b : integer);
 4  begin
 5    a:=b*b
 6  end;
 7  procedure sanjyou(var a,b : integer);
 8  begin
 9    a:=b*b*b
10  end;
11  begin
12    b:=3;
13    c:=3;
14    readln(d);
15    if d=0 then
16      nijyou(a,b)
17    else
18      sanjyou(a,c);
19    writeln(a)
20  end.
```

図 3: プログラム

```
 1  program double_or_tripple(input,output);
 2  var a,b,c,d : integer;
 3  procedure nijyou(var a,b : integer);
 4  begin
 5    a:=b*b
 6  end;
 7
 8
 9
10
11  begin
12    b:=3;
13
14    readln(d);
15    if d=0 then
16      nijyou(a,b)
17
18
19    writeln(a)
20  end.
```

```
 1  program double_or_tripple(input,output);
 2  var a,b,c,d : integer;
 3
 4
 5
 6
 7  procedure sanjyou(var a,b : integer);
 8  begin
 9    a:=b*b*b
10  end;
11  begin
12
13    c:=3;
14    readln(d);
15    if d=0 then
16
17
18      sanjyou(a,c);
19    writeln(a)
20  end.
```

図 4: 異なる入力による CH スライス

表 4: プログラムの実行時間の比較

(P5-100MHz,memory 48MB)

(単位：ミリ秒)

	100 行	400 行
静的スライス	391	475
動的スライス	429	3232
CH スライス	407	544

表 5: スライスの計算時間の比較

(P5-100MHz,memory 48MB)

(単位：ミリ秒)

	100 行	400 行
静的スライス	5.4	83
動的スライス	800	13000
CH スライス	3.1	117

時間的コストを静的スライス，動的スライスと比較すると，表2，表5から，静的スライスと同程度であり，動的スライスと比較してかなり小さいことが分かる．また抽出されるCH スライスのサイズは，静的スライスに比べ平均１５%程度小さくできることが分かる．しかし動的スライスと比べると大きいため，改善の余地がある．改善の方法としては，関数呼出し文の実行履歴以外の動的情報を用いて，実行されなかった文の特定を行うことが挙げられる．条件文の次の文に対する実行依存関係を調べ，それらの文の実行履歴を用いて，実行されなかった文の特定を行なうことでさらに動的スライスの結果に近づくことができると考えられる．

5 まとめと今後の課題

本研究では動的情報を用いて，実行されなかった文の特定を行い，それらの文をPDGから除外してスライスを計算することで，静的スライスよりも小さなPDGから，CH スライスを抽出する技法を提案した．

今後の課題としては，関数の呼出し履歴を動的情報として用いたが，用いる動的情報を工夫することで，さらなる効率の向上が可能であると考えられる．

参考文献

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

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

[3] Agrawal, H., Demillo, R.A. and Spafford, E.H.:"Debugging with Dynamic Slicing and Backtracking", Software-Practice and Experience, Vol.23(6), pp.589-616(1993).

[4] Cordy, J.R., Eliot, N.L. and Robertson, M.G.:"TuringTool:A User Interface to Aid in the Software Maintenance Task", IEEE Transactions on Software Engineering, Vol.16, No.3, pp.294-301(1990).

[5] Gupta, R. and Soffa, M.L.:"Hybrid Slicing : An Approach for Refining Static Slices Using Dynamic Information ", Proceedings of the third ACM Symposium on the Foundations of Software Engineering, pp.29-40(1995).

[6] Korel, b. and Laski, j., "Dynamic Program Slicing". *Information Processing Letters*, Vol.29, No.10, pp.155-163(1988).

[7] Korel, B. and Laski, J.:"Dynamic Slicing of Computer Programs", *J. Systems Software*, Vol. 13, pp. 187-195 (1990).

[8] Myers, G. J.: *The Art of Software Testing*, Wiley-Interscience(1979).

[9] 西松，井上:"依存関係解析に基づく開発支援システムへの動的スライス抽出機能の追加"，情報処理学会第55回（平成9年後期）全国大会 講演論文集(1), page418-419.

[10] 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",情報処理学会論文誌, vol.37, pp. 536-545(1996-4).

[11] Shimomura, T.:"Bug Localization Based on Error-Cause-Chasing Methods", Transactions of information Processing Society of Japan, Vol. 34, No. 3, pp. 489-500(1993).

[12] 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).

[13] 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", 電子情報通信学会論文誌, vol. J78-D-I, pp. 11-22(1995).

[14] Weiser, M.: "Program Slicing", Proceedings of the Fifth International Conference on Software Engineering, pp. 439-449(1981).