特別研究報告

題目

動的情報を利用した Java スライスシステム

～メソッド間動的データ依存関係の適用 と ユーザインターフェース部の試作～

指導教官
井上 克郎 教授

報告者
藤井 将人

平成 13 年 2 月 22 日

大阪大学 基礎工学部 情報科学科

平成 12 年度 特別研究報告

動的情報を利用した Java スライスシステム
～メソッド間動的データ依存関係の適用 と ユーザインターフェース部の試作～

藤井 将人

内容梗概

プログラム中のフォールト位置の特定を効率よく行なう手法として，プログラムスライスがある．このプログラムスライスを，現在のソフトウェア開発環境での利用が高まっているオブジェクト指向言語に適用することを考えたとき，オブジェクト指向プログラムには実行時に決定される要素が多く含まれることから，対象プログラムの実行を伴わない静的解析では十分な解析結果が得られない．しかし，対象プログラムの実行が前提となる動的解析では解析精度の向上は得られるもののコスト増加は避けられない．

そこで我々の研究グループでは，静的解析と動的解析を組み合わせた，オブジェクト指向プログラムに対する解析手法を提案し，Java を対象とするスライスシステムとして実現している．

本研究では，動的情報としてメソッド間動的データ依存関係を利用する解析手法の提案を行う．本手法を用いることで，Calling-Context と呼ばれる，メソッド呼出し文と呼出しメソッドとの対応関係を正しく抽出することが可能となり，上記手法から更にスライスサイズの削減を得ることができる．また，研究グループで開発している Java スライスシステムの GUI 部として，スライス表示ツールの試作も行なった．本ツールは，スライス抽出結果のソースコードへのマッピングを基本機能とし，解析情報を有効利用するためのさまざまな機能を実現している．本ツールを利用することにより，プログラムスライスに基づくプログラム理解・デバッグ支援が可能となる．

主な用語

プログラムスライス (Program Slice)
Calling-Context
メソッド間データ依存関係
ユーザインターフェース (GUI)

目 次

1 まえがき **4**

2 プログラムスライス **5**

2.1 スライス計算手順 5

2.1.1 Phase 1:依存関係解析 5

2.1.2 Phase 2:プログラム依存グラフの構築 6

2.1.3 Phase 3:スライス抽出 6

2.2 静的スライス 6

2.2.1 静的スライスを計算するアルゴリズム 6

2.3 動的スライス 8

2.3.1 動的スライスを計算するアルゴリズム 8

2.4 Dependence-Cache スライス 10

2.4.1 Dependence-Cache スライスを計算するアルゴリズム 10

2.5 各スライス手法の比較 11

3 動的情報を用いた Java スライスシステム **14**

3.1 システム概要 14

3.2 本研究内容 14

4 動的メソッド間データ依存関係解析 **16**

4.1 諸定義 16

4.2 Calling-Context 16

4.3 提案手法の概要 17

4.4 Add Method-Indexed-Edge アルゴリズム 18

4.5 Trace Indexed-Edge PDG アルゴリズム 21

4.6 評価実験 27

4.6.1 全ての依存辺に履歴を付加する手法 27

4.6.2 実験結果 29

4.6.3 考察 30

5 スライス情報表示ツール **32**

5.1 概要 32

5.2 表示ツールの機能 33

5.2.1 ツール内からプログラムのコンパイル・実行 33

5.2.2　スライス結果の効果的な表示　. 33
5.2.3　複数のファイルからなるプログラムのスライス結果全体像の把握 . . . 36

6　まとめと今後の課題 **37**

謝辞 **38**

参考文献 **39**

1 まえがき

プログラム P のプログラムスライス(*Program Slice*) とは，直観的に P 中のある地点 n およびある変数 v に対して，n における v の値に影響を与える P 中の各文や式の集合をいう．このスライスの抽出技法は Mark Weiser [1] によって提案され, 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，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.

当初，スライス計算の対象となるプログラム言語は，非常に単純な言語であったが，実用的な言語 (例えば，C 言語) にスライスを適用すると，さまざまな問題点が指摘されるようになった．その問題点の一つに**Calling-Context**[4] の問題があげられる．一般的に，Calling-Context とは，メソッド呼出し文と各メソッドとの対応関係を表すが，この Calling-Context を考慮に入れない解析を行なうと，スライスサイズが大きくなることが知られている．特に，近年ソフトウェア開発環境では，拡張容易性・抽象化・カプセル化・再利用性などをもつオブジェクト指向プログラミング言語が多く利用されているが，このオブジェクト指向プログラミング言語はデータのやりとりをメソッドを通して行なうために，プログラム実行時に多くのメソッドが呼び出されることが予想される．よって，Calling-Context 問題の解決がより重要となる.

本研究では，この Calling-Context 問題の解決を行なうために，メソッド間データ依存関係を動的に追跡しスライスサイズを削減する手法 (4 節参照) を提案する．ここで，メソッド間データ依存関係とは，メソッド間の依存関係を表す．本手法では，このメソッド間データ依存関係に履歴番号を付加し，その情報をもとにスライスの抽出を行なう.

同時に，抽出されたスライスを視覚的に利用するための，表示ツール (ユーザインターフェース,*GUI*) の試作 (5 節参照) も行なった．GUI により，解析情報を有効に利用し，スライス結果とソースコードとのマッピングを行なえるため，プログラムのテスト・デバッグなどの支援が可能となり，解析結果を有効に利用できる.

以降，2 節ではプログラムスライスについて紹介する．3 節では，動的情報を用いた Java スライスシステムについて紹介し，4 節で，そのシステムのプログラム解析手法に適応されている，動的メソッド間依存関係解析手法の提案する．また，5 節では，システムの一つであるスライス表示ツールの設計と説明を紹介する．6 節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

プログラムスライシング(*Program Slicing*)技術とは，プログラム P 中のある文 s におけるある変数 v に対して,v の値に影響を与える全ての文，もしくは，s における変数 w の定義に対して w が影響を与える全ての文の集合 $\mathcal{Q}$ のことである．この文の集合をプログラムスライスまたは単にスライス(*Slice*)と呼ぶ．プログラムスライシング技術は，Mark Weiser[1]によって提案され，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，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．

2.1　スライス計算手順

スライスの計算には，次のような手順をたどる．

Phase 1:　依存関係解析 (データ依存関係，制御依存関係)

Phase 2:　プログラム依存グラフ (*Program Dependence Graph,PDG*) の構築

Phase 3:　PDG を探索しスライスを抽出

2.1.1　Phase 1:依存関係解析

Phase1 では，まずソースコードを入力とし，プログラム中の文間の依存関係解析を行う(文献 [6] を参照)．依存関係には以下の制御依存関係とデータ依存関係の 2 種類がある．

- 制御依存関係
  今，ソースプログラム p 中の文 s_1，s_2 について考える．以下の条件を全て満たしているとき，文 s_1 から文 s_2 への制御依存 *(Control Dependence，CD)* があるという．
  - 文 s_1 が条件文である．
  - 文 s_2 が実行されるかどうかは，文 s_1 の結果に依存する．
- データ依存関係
  今，ソースプログラム p 中の文 s_1，s_2 について考える．以下の 3 つの条件を全て満たすとき，文 s_1 から文 s_2 へ変数 v に関してデータ依存 *(Data Dependence，DD)* があるという．
  - 文 s_1 で変数 v を定義している．
  - 文 s_2 で変数 v を参照している．

– 文 s_1 から文 s_2 への実行可能なパスで，そのパスにおいて文 s_1 から文 s_2 間に変数 v を再定義している文が存在しない，というものが存在する．

スライスのうち，依存関係解析を全ての可能な入力データに関して行なったスライスを静的スライス[1](2.2 節参照)，ある特定の入力データに関して行なったスライスを動的スライス[?](2.3 節参照) という．また，制御依存関係を静的に，データ依存関係を動的に解析して行なうスライスを**DC** スライス[3](2.4 節参照) という．

2.1.2 Phase 2:プログラム依存グラフの構築

Phase2 では，Phase1 での依存関係解析で得られた依存関係情報を元に，プログラム依存グラフ(Program Dependence Graph．以降，PDG と呼ぶ) を構築する．PDG とは，プログラム内の文間の依存関係を有向グラフで表現したものであり，その節点は，プログラムに含まれる条件判定部文，代入文，入出力文，メソッド呼びだし文を表し, その有向辺は２つの節点間の制御依存およびデータ依存関係を表す (それぞれ制御依存辺，データ依存辺と呼ぶ)．また，メソッド間にわたるデータ依存関係を表現するための特殊節点及び特殊辺も存在する [4, 6]．

2.1.3 Phase 3:スライス抽出

Phase3 では，Phase2 までに構築された PDG を用いてスライスを抽出する．スライスを計算抽出には，まず『どの文のどの変数に対してスライスを計算するか』を指定する．これをスライシング基準と呼ぶ．スライシング基準は，文と変数の組から成る．このスライス基準から，PDG を探索し，到達可能な節点の集合をスライスとして検出する．

2.2 静的スライス

2.2.1 静的スライスを計算するアルゴリズム

静的スライスの計算方法として [5] のアルゴリズムを用いる．このアルゴリズムは [4, 6] を参考にしてプログラム中の文の依存関係を調べ，それを元にプログラム依存グラフ(PDG)を作成し，その辺をたどることにより静的スライスを計算する．

- 諸定義

PDG はプログラム内の文の依存関係を表すグラフである．PDG の節点はプログラム中の各文及び if 文や while 文の条件判定部分を表し，辺は変数の影響を伝えるデータ依存関係及び条件文や繰り返し文の制御の影響を伝える制御依存関係を表す．

データ依存関係は，各頂点の到達定義集合 (Reaching Definitions，略して RD) 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，変数 v と頂点 s との組 $\langle v, s\rangle$ の集合である．これは，

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 ($\mathrm{DD}(s, v, t)$) があるという．

また，ある条件判定部分 s の結果により文 t の実行の有無が決定される時，s と t との間に制御依存関係 ($\mathrm{CD}(s, t)$) が存在するものとする．すなわち制御依存関係は if 文や while 文の条件判定部分からそれらの内部ブロックに属する文への影響であり，これはプログラムを解析すれば容易に求められる．

一般にプログラムには複数の手続きが定義されている．各手続き間には引数や大域変数を通じてデータ依存関係が生じる．これらのデータ依存関係を表すために，PDG にプログラム中の文とは直接対応しない節点 (特殊節点と呼ぶ) を用意する．

PDG の作成は，プログラムを解析し，プログラムの各文を PDG の節点に切りわけ，プログラム中の各文における RD を求め，それをもとにして PDG の各辺を生成することによって行なわれる [6].

上記の方法で生成された 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 に関するスライスを表す節点の集合 V は以下のようにして計算される．

1. $V \leftarrow \{n | n$ は S に対応する節点. $\}$

2. $N \leftarrow \{n | \langle v, n\rangle \in RD_{in}(S)\}$

3. $N \neq \phi$の間以下の動作を繰り返す．

   - $l \in N$ を一つ選ぶ．
   - $N \leftarrow N - \{l\}$
   - $V \leftarrow V \cup \{l\}$
   - $N \leftarrow N \cup \{k | (k \notin V) \wedge (k \xrightarrow{\star} l \in E \vee k \dashrightarrow l) \wedge (k$ が $g\text{–}in$ でない$) \wedge (k$ が $para\text{–}in$ 節点でない$)\}$(ただし $k \xrightarrow{\star} l$ は任意の変数の k から l へのデータ依存関係辺を示す.)
   - k が $para\text{–}in$ 節点でも $para\text{–}in$ 節点でもないなら
     $N \leftarrow N \cup \{k | (k \notin V) \wedge (k \xrightarrow{\star} l \in E \vee k \dashrightarrow l)\}$(ただし $k \xrightarrow{\star} l$ は任意の変数の k から l へのデータ依存関係辺を示す．)

2.3 動的スライス

2.3.1 動的スライスを計算するアルゴリズム

入力を与えてプログラムを実行した場合，実行された文の列を実行系列と呼ぶ．実行系列で p 番目に文の実行が行われた時点を実行時点 p と呼ぶ．動的スライス [?] を計算するために必要となる動的依存関係情報を実行系列に与える．動的依存関係を以下の様に定義する．

1. データ依存関係（**Definition-Use，DU**）
   p =Def(q, w) とは，実行時点 q より前で最後に変数 w を定義した実行時点が p であることを表す．最後に定義されたとは，$p < r < q$ であるような全ての実行系列 r において，変数 w が定義されていないことをいう．また，実行時点 p において実行された文を Ins(p) で表し，実行時点 p に対して，Def(p) は実行時点 p における文 Ins(p) の実行により定義された変数，Use(p) は実行時点 p における文 Ins(p) の実行により使用された変数の集合を表す．実行時点 p から実行時点 $q(p < q)$ に対してデータ依存関係 DU(p, q) があるとは，ある変数 $w \in$ Use(q) が存在して，p =Def(q, w) の場合をいう．データ依存関係 DU(p, q) は，実行時点 p で設定したある変数 w の値が実行時点 q で参照されたことを示している．変数 $w \in$ Use(q) に関してデータ依存関係があることを明示する場合には，DU$w(p, q)$ と表すこととする．

2. 制御依存関係（**Test-Control，TC**）
   まず，命令 t に対して，命令の集合 CtlExec(t) を次のように定義する．

   $$\text{CtlExec}(t) = \{\ \text{命令}\ s | \text{CD}(s, t)\}$$

   ただし，命令 t がループ命令の場合には命令 t も CtlExec(t) に含める．
   実行時点 p から実行時点 $q(p < q)$ に対して制御依存関係 TC(p, q) があるとは，

   $$p = \max\{\ \text{実行時点}\ i | i < q\ \text{かつ}\ \text{Ins}(i) \in \text{CtlExec}(\text{Ins}(q))\}$$

   の場合をいう．
   制御依存関係 TC(p, q) は，実行時点 q において命令 Ins(q) が実行されたことは，実行時点 p において実行された分岐命令あるいはループ命令 Ins(p) の制御移行に依存していたことを示している．

   また，実行系列 q における命令 Ins(q) が，関数，手続き内にある命令であるなら，その関数，手続きを呼び出した実行時点 q との間にも制御依存関係 TC(p, q) があるという．このとき実行時点 q における命令 Ins(q) は，関数，手続き呼び出し文である．

上記の依存関係を元に以下に示す依存関係情報を実行系列に与える．変数 v と実行時点 p との組 r を $\langle v, p \rangle$ と表す．ある実行時点 p の参照変数とその変数が定義された実行時点の組の集合を $VarREF(p)$ と書き．これを以下のように定義する．

$$VarREF(p) \equiv \{ \langle v, q \rangle \mid \mathrm{DU}v(q, p) \}$$

また，ある実行時点 p と制御依存関係がある実行時点 q の集合を $\mathrm{Control}(p)$ と表す．

次にこれまでの手順により依存情関係報与えられた実行系列を元にプログラム P の動的スライスを計算する．まず，どの実行時点のどの変数に関して動的スライスを計算するかを決定する．その実行時点を動的スライスのスライス基準といい $C = (x, r, V)$ と表す．x, r, V は以下のとおりである．

- x はプログラム P に与えた入力．
- r はプログラム P に入力 x を与えたときの実行系列における実行時点．
- V はプログラム P 内の変数の部分集合．

プログラム P のスライス基準 $C = (x, r, V)$ に関する動的スライスは以下の手順により求められる．

1. DS $\leftarrow \phi$
   CalcDsObj $\leftarrow \phi$
   CalcDsObj を動的スライスを計算する対象となる実行時点の集合とする．

2. DS $\leftarrow$ {Ins(r)}
   CalcDsObj $\leftarrow$ { 実行時点 q | ある変数 $v \in$ V に対して q=Def(r, v)}

3. $p \in$ CalcDsObj
   CalcDsObj $\leftarrow$ { 実行時点 q | $VarREF(p) \cup$ Contorol(p)} $\cup$ CalcDsObj
   DS $\leftarrow$ DS $\cup$ {Ins(p)}
   CalcDsObj $\leftarrow$ CalcDsObj - {p}

4. CalcDsObj が空集合でないなら 3 を行う．CalcDsObj が空集合なら DS がプログラム P のスライス基準 $C = (x, r, V)$ に関する動的スライスとなる．

2.4 Dependence-Cache スライス

静的スライス計算法は，スライスに含まれるべき文が排除されることを避けるために，全ての可能な入力データを考慮することになる．このため，実行に関係しない文 (デバッグ時に必要ないと考えられる文) もスライスに含んでしまうという欠点がある．対して，動的スライス計算法は，特定の入力データのみを考慮し実行系列を対象に解析を行うことで，静的スライスと比較して，スライスサイズの減少 (正確性の向上) が期待できる．しかし，実行系列はソースプログラムと比較して膨大な量になってしまう場合が多く，その保存と解析に多くの空間，時間コストを必要としてしまう．つまり，スライスの計算に必要なコストとスライスサイズ (正確性) はトレードオフの関係にある．

DC スライス [?] は，動的にデータ依存関係を解析することで，配列の添字，ポインタの参照先を把握する．一方，制御依存関係は静的に解析し，実行系列を保存することはしないため動的スライス抽出に比べて実行時間の短縮が得られる．

2.4.1 Dependence-Cache スライスを計算するアルゴリズム

実行中に，ある文 s である変数 v が参照された時，v がどの文 (t) で定義されたかが分かれば，$DD(t, v, s)$ どいうデータ依存関係があることが分かる．逆に言えば，v を定義してる文さえ分かればデータ依存関係を知ることができる．

そこで，全ての変数に対して，その値を定義したのはどの文か，という情報を持たせておき，その変数の参照があった場合には，その情報からデータ依存関係を把握する．

また，得られたデータ依存関係は各文に持たせる．各文 s は集合 $DDS(s)$ を持つとする．この $DDS(s)$ の要素は，2 つの要素からなる組であり，(“依存関係の原因となる変数 v”，“s がデータ依存している文”) となっている．

ある実行時点において，変数 v を最後に定義した文を $DefS(v)$ とする．ここで v は，動的に生成される変数も含めた全ての変数を考える．配列では，各要素にも $DefS$ を考える．

(1) ある文 s を実行する前に，$DDS(s) = \phi$ とする．

(2) 入力データを与えてプログラムの実行を行う．今，文 s が実行されたとする．

- 文 s で変数 v が参照[1] された場合，$DDS(s) \leftarrow DDS(s) \cup (v, DefS(v))$．とする．
- 文 s で変数 v が定義された場合，$DefS(v) = s$ とする．

[1] ポインタ変数が出現した場合，何が参照されたかを判断する際に注意が必要となる．例えば，代入文の右辺や出力文などに $**v$ という 2 階のポインタが現れた場合，v，$*v$，$**v$ が参照されたと考える．また，代入文の左辺に現れた変数は普通参照されたとは考えないが，例えば $*v$ という一階のポインタが現れた場合は，v を参照したと考える．

この作業によって得られた $DDS(s)$ は，全てのデータ依存関係を表しており，$DDS(s) = \{(v,t)|DD(t,v,s)$ が成り立つ $\}$ となっている．

(3) 集合 DDS を使って PDG を構築する．文 s の解析において，

- $DDS(s) \neq \phi$ の間，次の操作を繰り返す．
  - $DDS(s) \leftarrow DDS(s) - \{(v,t)\}$.
  - データ依存辺 $t \xrightarrow{v} s$ を引く．
- s が制御文であれば，その制御文内の全ての文 t に対して，制御依存辺 $s \dashrightarrow t$ を引く．

2.5 各スライス手法の比較

静的スライス，動的スライス，DC スライスの違いを 表 1 に示す．

解析手法の違いにより，一般に以下の様な性質となる．これらの性質は [3] において，実験によって実証されている．

スライスサイズ: 静的 $\geq$ DC $\geq$ 動的

実行前解析時間: 静的 $>$ DC $>$ 動的

実行時間: 動的 $>$ DC $>$ 静的

表 1: 各スライス手法の違い

	静的スライス	動的スライス	DC スライス
制御依存関係解析	静的	動的	静的
データ依存関係解析	静的	動的	動的
PDG 節点	文	実行系列	文

また，比較例に，C サンプルプログラムの，スライス基準 $(23,d)$ に関する各スライスを図 1 と 図 2 に示す．動的スライス・DC スライスの入力には，2 を与えたとする．

C サンプルプログラム

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 3:
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 7:
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
13:
14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;
19:
20:      for (i=0;i<SIZE;i++) {
21:          b[i] = a[i];
22:      }
23:
24:      printf("Input: ");
25:      scanf("%d", &c);
26:
27:      if (c >= SIZE) {
28:          c = c % SIZE;
29:      }
30:
31:      d = cube(b[c]);
32:
33:      if (d < 0) {
34:          d = -1 * d;
35:      }
36:
37:      printf("%d\n", d);
38:  }
```

静的スライス結果

```
 1:  #include <stdio.h>
 2:  #define SIZE 5

 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }

 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;

14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;

20:      for (i=0;i<SIZE;i++) {
21:          b[i] = a[i];
22:      }


25:      scanf("%d", &c);

27:      if (c >= SIZE) {
28:          c = c % SIZE;
29:      }

31:      d = cube(b[c]);

33:      if (d < 0) {
34:          d = -1 * d;
35:      }

37:      printf("%d\n", d);
38:  }
```

図 1: C サンプルプログラムの $(23, d)$ に関する各スライス結果 1

動的スライス結果

```
 1:  #include <stdio.h>
 2:  #define SIZE 5

 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }

 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;



16:      a[2] = 2;



20:      for (i=0;i<SIZE;i++) {
21:          b[i] = a[i];
22:      }


25:      scanf("%d", &c);









31:      d = cube(b[c]);









37:      printf("%d\n", d);
38:  }
```

DC スライス結果

```
 1:  #include <stdio.h>
 2:  #define SIZE 5

 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }

 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;

14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;

20:      for (i=0;i<SIZE;i++) {
21:          b[i] = a[i];
22:      }


25:      scanf("%d", &c);









31:      d = cube(b[c]);









37:      printf("%d\n", d);
38:  }
```

図 2: C サンプルプログラムの $(23, d)$ に関する各スライス結果 2

3 動的情報を用いた Java スライスシステム

現在のソフトウェア開発環境において，C などの手続き型言語だけでなく，Java, C++等いわゆるオブジェクト指向言語の利用が高まっている．我々の研究グループでは，特に近年利用が高まっている Java[12, 13, 15] を対象にした Java スライスシステムの提案・実装を行なった．

3.1 システム概要

Java では，配列の添字やオブジェクトの参照など，実行中に決定される要素を多く含むため，静的スライスだけでは限界がある．しかし，動的スライスは手続き型言語を対象とした動的スライス同様に実行系列を保存するため，解析に多大なコストがかかると考えられる．

そこで，我々の研究グループでは，Java プログラムスライスシステムの実現にあたり，DC スライスの適用を提案した [10]．しかし，この DC スライスでは解決できない**Calling-Context**[4](4.2 節参照）により生じる問題がある．この問題を解決するため，本研究では，4 節で説明を行なう，メソッド間動的データ依存関係解析手法のも提案をおこない，システムの解析部に実装を行なった．

システムの概要は以下のようになる．(図 3 参照)

1. ソースプログラムの制御依存関係を静的に解析し，コンパイルする．
2. プログラム実行時に，動的にデータ依存関係・メソッド間データ依存関係を解析し，PDG を構築する
3. GUI でスライス基準を指定し，スライスを抽出・表示する

3.2 本研究内容

本研究では，システムにおける，メソッド間動的データ依存関係解析(4 節参照) 手法の提案と，ユーザインターフェース(GUI)(5 節参照) の試作を行なった．

4 節では，この手法を提案した背景と概要，およびアルゴリズムについて説明を行なう．5 節では，GUI でスライスを表示するために必要となる機能についての説明を行なう．

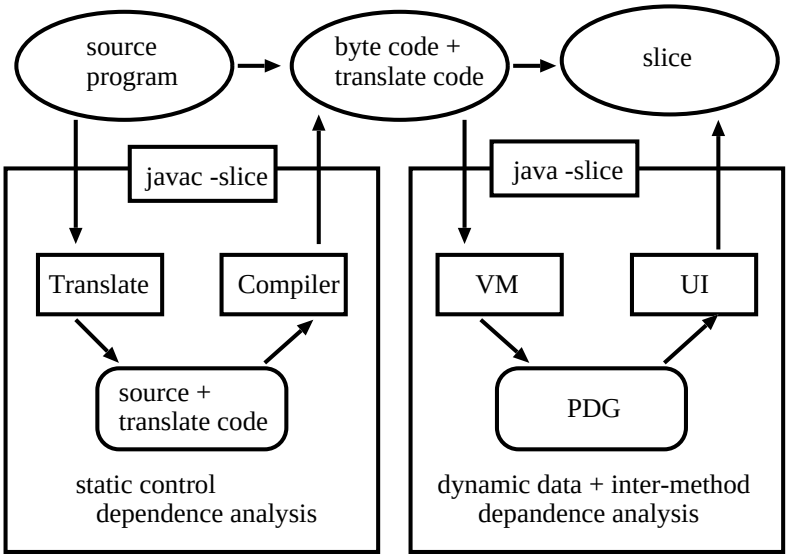

図 3: Java スライスシステムの概要

4　動的メソッド間データ依存関係解析

本節では，まず動的メソッド間データ依存関係解析の手法の提案を行なう．まず，提案にいたる背景となった**Calling-Context**[4] についての説明を行ない，その後に手法内容・評価実験の説明を行なう．

4.1　諸定義

- メソッド間データ依存辺

メソッド間データ依存辺とは，PDG 上でメソッド間をわたる依存辺のことである．プログラムにおいて，メソッドが呼び出された時やメソッドから呼出し先に戻る時に引かれる．しかし実際には，メソッド呼出し文が PDG の節点となり依存辺が引かれるのではなく，各メソッドごとに特殊節点 [?] が用意され，その節点のあいだに依存辺が引かれることになる．これは，各メソッドは引数や大域変数を通してデータ依存関係が生じるためである．特殊節点には 表 2 のようなものが存在する．

表 2: 特殊節点

global-def-in	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，その手続き内で定義/参照している大域変数に対して，ひとつずつある
global-def-out	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，定義/参照しているの大域変数に対して，ひとつずつある
global-out	関数/手続き呼出しによって関数が実行された後に，その関数から制御が戻って来るときに，その関数内の大域変数に与えた影響を呼出した関数 (手続き) に与えるための節点である．
parameter-in	関数/手続き呼出しの引数を表現するための節点である．各引数ごとに用意される．
parameter-out	関数/手続き呼出しの引数が，呼ばれた関数により，影響を受けている場合 (例えば，参照渡しで値が変更されたような場合)，呼んだ関数に制御が戻ってきたときに，その影響を伝えるための節点である．
actual-in	手続き/関数の実引数の手続き/関数内部への影響を伝えるための節点である．
actual-out	手続き/関数の実引数の手続き/関数外部への影響を与えるための節点である．
func-return	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，各関数呼出しにひとつずつある．
func-exit	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．

4.2　Calling-Context

Calling-Context[4] とは，メソッド呼出し文と，各メソッドとの対応のことである．次の 図 4 を例に説明する．

図 4 の main の文 6 変数 c について，データ依存辺がある文を考える．Calling-Context を考慮にいれると，文 5 でメソッド m を呼んだので，main に戻る時も文 5 に戻る．しかし，Calling-Context を考慮にいれないと，文 5 でメソッド m を呼んだのにもかかわらず，main

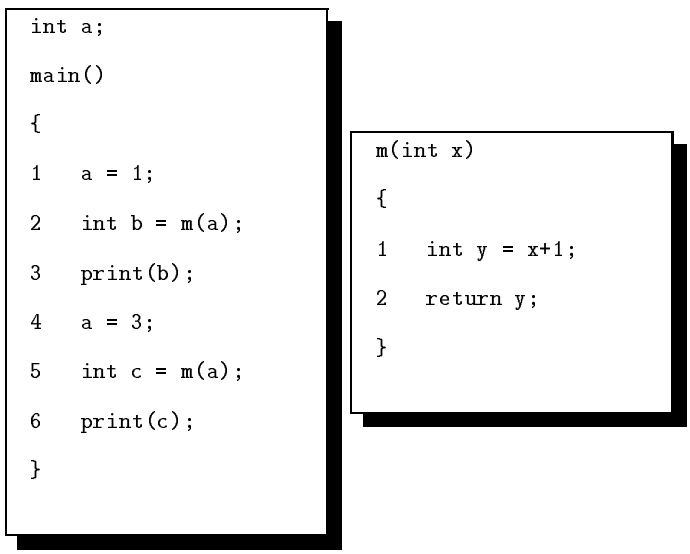

図 4: Calling-Context

に戻る時に文 2,5 のどちら戻ればいいのかわからず，両方に戻ってしまう．この場合，スライス結果に余分な文 1,2 が含まれてしまう．

Java において全てのプログラム活動は特定のクラス内部で行なわれる．さらに，各クラスにはいくつかのメソッドが入っているとみなすことができ，メソッドは Java の基本的な実行単位であるといえる．よって，プログラムの実行中何度もメソッドが呼び出されることが想定される．

よって，Calling-Context を考慮に入れた解析を行なうことにより，スライスサイズを削減できる．

4.3 提案手法の概要

Calling-Context を考慮にいれるために、メソッド呼び出し文と呼び出されるメソッドとの対応がわかっていなければならない。そのため本研究では、PDG 構築時依存辺を引く際に、その辺がメソッド間データ依存辺であるとき、辺に実行された履歴番号 (index) を付加するという手法を提案する。

本手法では，PDG 構築時にメソッド間データ依存辺が引かれるたび，その依存辺に履歴番号を付加しておき，PDG 探索時にその番号にそって，探索を行なうことにより，Calling-Context を考慮したスライスを抽出することが可能となる．まず $index = 0$ としておき，メソッド間データ依存辺 (以降，メソッド呼び出し側からメソッド側への依存辺を CALL 辺，メソッド側から呼び出し側に戻る依存辺を RETURN 辺とする) が引かれる度に、index をインクリメントして付加する。PDG 探索時には，この付加された番号を元に，CALL 辺と

RETURN 辺の依存関係を解析し，対応関係が存在する辺のみ探索を行なうようにする．

次の 4.4 節では，PDG 構築時に index(番号) を付加するアルゴリズムを，4.5 節では，index を付加された PDG をたどり，スライスを抽出するアルゴリズムを説明する．

4.4 Add Method-Indexed-Edge アルゴリズム

Add Method-Indexed-Edge アルゴリズムでは、PDG 構築時にメソッド間データ依存辺に index を付加する方法を示す。まず，メソッド呼びだし文や retun 文 (または，return にあたる文) が実行された時，そのメソッドにわたすデータの数だけ特殊節点が呼び出し側，呼び出される側両方の PDG に構築される．この時 index の値をインクリメントしておく．そして，その特殊節点間を依存辺が引かれるわけであるが，この時に先ほどインクリメントしておいた index 値を依存辺に付加する．

ここで， 図 5 の Java プログラム[2]において，Add Method-Indexed-Edge アルゴリズムを使用し，構築される PDG を図で表したものを， 図 7 に示す．

```
int a = 1;  // global
main(){
   int b = 1;
   int c = m(b);
   print(c);
   if (c > 5){;
      int d = m(c);
      print(d);
    }
}
```

```
m(int x)
{
   int y = n(x+a);
   return y;
}
```

```
n(int s)
{
   int t = s*2;
   return t;
}
```

図 5: Java サンプルプログラム

[2] Java プログラムにおいて正確な出力文は,System.out.println(c) などと記述するが，ここでは省略し，print(c) と書く.

アルゴリズムADD METHOD-INDEXED-EDGE

前提 命令 s に対する PDG 節点を n_s とする

入力 静的制御依存関係解析が行なわれた Java バイトコード

出力 メソッド間の DD-Edge に index 情報を付加した PDG

処理 PDG にメソッド間の DD-Edge に index 情報を付加する

(1) $call_number = 1$

(2) **while** 実行中 **then begin**

(3) **if** 命令 s が命令 t で定義した変数 x を参照する **then begin**

(4) $n_t \leftarrow n_s$ に x に関するデータ依存辺を引く

(5) **end**

(6) **else if** 命令 s が条件節 t に依存 **then begin**

(7) $n_t \leftarrow n_s$ に x に関する制御依存辺を引く

(8) **end**

(9) **else if** 命令 s がメソッド呼出し文 **then begin**

(10) $call_number = call_number + 1$

(11) **while** 依存辺が引かれていない特殊節点が存在 **then begin**

(12) $index = call_number$ を付加した依存辺を引く

(13) **end**

(14) **end**

(15) **else if** 命令 s が return 文などのメソッド終了文 **then begin**

(16) $call_number = call_number + 1$

(17) **while** 依存辺が引かれていない特殊節点が存在 **then begin**

(18) $index = call_number$ を付加した parameter 辺，return 辺を引く

(19) **end**

(20) **end**

(21) **end**

図 6: アルゴリズムADD METHOD-INDEXED-EDGE

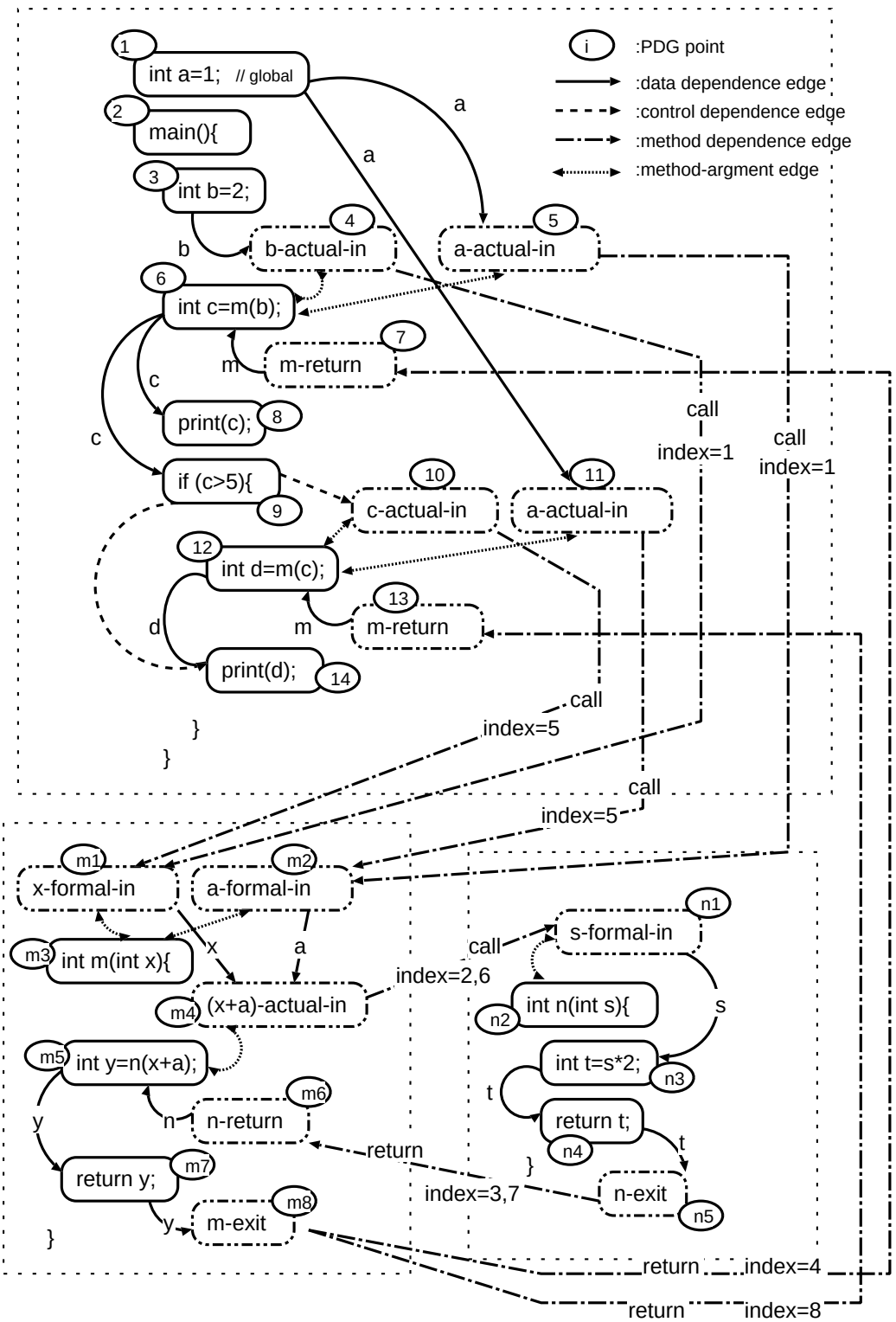

図 7: メソッド間データ依存辺の付け方

4.5 Trace Indexed-Edge PDG アルゴリズム

Trace Indexed-Edge PDG アルゴリズムでは、スライス抽出のためにメソッド間データ依存辺に index が付加された PDG をたどる方法を示す。

Calling-Context を考慮するためには，CALL 辺と RETURN 辺の対応関係が分かればよい．メソッド間依存辺に履歴番号 index が付加されているため，この index 値をもとに対応関係を求める．今，$index = n$ の RETURN 辺をたどってきたとする．すると，この RETUN 辺に対応する CALL 辺が必ず存在している．CALL 辺は一本以上存在し，その CALL 辺それぞれには少なく index が少なくとも 1 つは付加されているが，このうち今たどってきた RETURN 辺に対応する CALL 辺およびその index は

$index <= n$ を満たす **index** の最大値をもつ **CALL** 辺

となる．例えば，今 $index = 13$ が付加された RETURN 辺をたどりメソッド M に探索が移ったとする．その後，このメソッド M を呼び出した先に戻る時 CALL 辺の探索を行なうをわけであるが，この CALL 辺に $index = 4, 7, 10$ が付加されていたとする．すると，先にたどってきた RETURN 辺に対応する CALL 辺は $index = 10$ が付加された CALL 辺であることが分かる．よって，この CALL 辺だけを探索する必要がある．

また，たどってきた RETURN 辺の index 値が n，この RETURN 辺に対応する CALL 辺の index 値が m と判断した時，$m == n$ または $m == n - 1$ であるなら，この CALL 辺から RETURN 辺の間に，メソッド呼び出し文が存在しないが，$m < n - 1$ であるなら，CALL 辺から RETURN 辺の間に，メソッド呼び出し文が存在することになる．よって，この途中で実行されるメソッド呼出し文についても探索を行なわなければならない可能性が存在するので，$m <= x < n - 1$ となる x が付加されている依存辺をたどろうとした時には，探索を行なうようにすればよい．

実際には，index が付加された依存辺をたどろうとした時，その時点ではその依存辺が探索されるべきであるのかどうかは判断できない．よって，その index 値の値と到達節点の情報を保持するため，ハッシュ表 HashTabale を用意した．たどる可能性が存在する $index = n$ の依存辺を探索しようとした時，その時点では探索を行なわず，すべて HashTable[n] に節点情報を保存する．そして，$index = n$ の依存辺の探索を行なわなければならないということが分かった時点で，その HashTable[n] に保存されている節点から探索を再開する．

しかしまだこれだけでは不都合が生じる可能性がある．これは，PDG の探索は一度到達した節点は探索を行なわないという性質があるためである．次の 図 8 を例に説明する．

いま 図 8 の method A から探索が始まったとする．探索では，まず $index = 6$ の RETURN 辺をたどり method B に探索が移る. この時，たどってきた RETURN 辺に対応する CALL

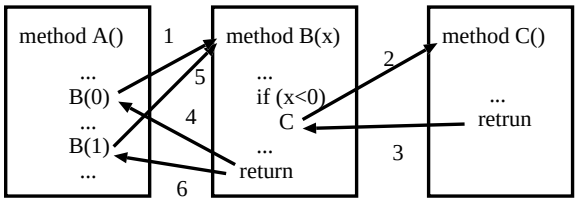

図 8: 探索が中断する例

辺は $index = 5$ の依存辺であることが分かるので，この時点で，$index = 3$ の依存辺はたどられないことになる．そして探索が進み，$index = 4$ の RETURN 辺をたどろうとする．しかしこの時，到達節点は既にたどられているため，それ以降の探索が行なわれない．すると，本来 $index = 3$ の依存辺もたどられ，method C もスライスに含まれるはずであるのに，含まれないような場合が生じる．よって，今たどってきた辺の index より小さい辺については，後でたどられる必要がある辺である可能性を持っているため，なんらかの方法でその情報を保持しておかなければならない．

よって本手法では，始めてそのメソッドの CALL 辺が探索される時に CALL 辺の index 値を全て保持しておく．そして，到達可能な節点が既に到達済みであるという状況になった時，保持しておいた CALL 辺の index 値から，対応関係のある index 値をもつ CALL 辺を捜し出し，2 つの依存辺の間にある探索されていない依存辺を探索できるようにする．

これらをまとめると次のようになる．今、PDG 上の接点 s にまで探索がすすみ、s まで探索を行なう時に $index == n$ の RETURN 辺をたどってきたとする。この時，s から到達可能な接点のうち

- 到達節点 t がまだたどられていない
  - index の付加されていない依存辺は探索を続ける
  - index の付加された CALL 辺の時，$CallIndex \leftarrow CallIndex \cup \{index\}$
  - $index = m$ の付加された依存辺で，$m == n$ または $m == n - 1$ である時，その時点で探索を行なわず，$HashTable[index] \leftarrow HashTable[index] \cup \{t\}$，$TraceIndex \leftarrow TraceIndex \cup \{index\}$ を行なう
  - i$index = m$ の付加された依存辺で，$m < n - 1$ 出ある時，その時点で探索を行なわない．しかし，後で探索を行なわなければならいない可能性がある節点であるので，$HashTable[index] \leftarrow HashTable[index] \cup \{t\}$，$CheckIndex \leftarrow CheckIndex \cup \{index\}$ を行なう．

    - さらに，依存辺がCALL辺ならCheckIndexの要素で$m <= x < n - 1$を満たすxをTraceIndexに追加
  - indexのついている依存辺でindexがn以上なら，探索を行なわない．

- 到達節点tがすでにたどられている
  - 到達節点がすでにたどられているため，n以下の依存辺がたどられない可能性が存在する．これを防ぐためにCallIndexで$m <= n$を満たすindex値mを探しだし，CheckIndexの要素で$m <= x < n - 1$を満たすxをTraceIndexに追加

なお，CallIndex・TraceIndex・CheckIndexはそれぞれ，CALL辺に付加されているindex値を保持するベクトル変数，必ず探索を行なわなければならないindex値を保持するベクトル変数，探索を行なおうとしたindex値を保持するベクトル変数である．

これをアルゴリズムとして記述したものを、 図9 のTrace Indexed-Edge PDGアルゴリズムと 図10 のCheck PDG-Pointアルゴリズムで示す。Trace Indexed-Edge PDGアルゴリズムは、スライスを抽出するためのアルゴリズムで、Check PDG-Pointアルゴリズムは、各接点での処理アルゴリズムである。

アルゴリズムTrace Indexed-Edge PDG

前提 PDG にメソッド間の DD-Edge に index 情報を付加されている
　到達するおそれのある節点情報を格納する HashTable[i],
　実行されなければならない，依存辺の index を格納する TraceIndex，
　探索を行なおうとした依存辺の index を格納する CheckIndex を用意

入力 スライス基準 v(=PDG 上節点 s)

出力 スライス結果

処理 メソッド間の DD-Edge に index 情報を付加した PDG をたどる

(1) **while** s から新たに到達可能節点 t が存在する **then begin**
(3)　**if** s からたどる PDG 辺 e に index が付加されていていない **then begin**
(4)　　**if** e が v に関するデータ依存辺または，制御依存辺 **then begin**
(5)　　Check PDG(t,0)　を実行
(6)　　**end**
(7)　**else if** e に index が付加されていている **then begin**
(8)　　Check PDG(t,index)　を実行
(9)　**end**
(10) **while** TraceIndex に要素が存在する **then begin**
(11)　TraceIndex の先頭の要素を取り除き，i に格納
(12)　**while** HashTable[i] に要素が存在 **then begin**
(13)　　HashTable[i] の先頭の要素を取り出し，s に格納
(14)　　Check PDG-Point(s,i)　を実行
(15)　**end**
(16) **end**
(17) T をスライス結果として出力

図 9: アルゴリズムTrace Indexed-Edge PDG

アルゴリズムCheck PDG-Point

前提 PDG にメソッド間の DD-Edge に index 情報を付加されている
　　このメソッドの CALL 辺の index 値を格納する CallIndex を用意
入力 PDG 節点 s, たどってきた依存辺の index 値 max
出力 既到達節点集合 T
処理 PDG 上の，到達するおそれのある節点情報を保存

Algorithm Check PDG-Point(s,max)

(1) $T \leftarrow T \cup \{s\}$
(2) **while** s から新たに到達可能節点が存在する **then begin**
(3)　**if** s からたどる PDG 辺 e に index が付加されていていない **then begin**
(4)　　**if** 到達節点 t が T に含まれない **then begin**
(5)　　Check PDG(t,max)　を実行
(6)　　**end**
(7)　　**else if** 到達節点 t が T に含まる　かつ $max \neq 0$**then begin**
(8)　　$\{n \mid n, \forall a \subseteq CallIndex \wedge a <= n <= max\}$ を満たす n に対し，
　　$\{m \mid \forall m \subseteq CheckIndex \wedge n <= m <= max\}$ を満たす全ての m について
　　$TraceIndex \leftarrow TraceIndex \cup \{m\}$
(9)　　**end**
(10)　**end**
(11)　**else if** e に index が付加されている **then begin**
(12)　　**if** 到達節点 t が T に含まれない **then begin**
(13)　　**if** 依存辺が CALL 辺 **then begin**
(14)　　$CallIndex \cup \{index\}$
(15)　　**end**
(16)　　**if** $index == max - 1$ または $max == 0$ **then begin**
(17)　　$HashTable[index] \leftarrow HashTable[index] \cup \{t\}$ かつ $TraceIndex \leftarrow TraceIndex \cup \{index\}$
(18)　　**end**
(19)　　**else if** $index < max - 1$ **then**
(20)　　$HashTable[index] \leftarrow HashTable[index] \cup \{t\}$
(21)　　**if** 依存辺が CALL 辺 **then**
(22)　　$\{m \mid \forall m \subseteq CheckIndex \wedge n <= m <= max\}$ を満たす全ての m について
　　$TraceIndex \leftarrow TraceIndex \cup \{m\}$
(23)　　**end**
(24)　　**end**
(25)　　**end**
(26)　　**else if** 到達節点 t が T に含まる **then begin**
(27)　　$\{n \mid n, \forall a \subseteq CallIndex \wedge a <= n <= max\}$ を満たす n に対し，
　　$\{m \mid \forall m \subseteq CheckIndex \wedge n <= m <= max\}$ を満たす全ての m について
　　$TraceIndex \leftarrow TraceIndex \cup \{m\}$
(28)　　**end**
(29)　**end**
(30) **end**

図 10: アルゴリズム Check PDG-Point

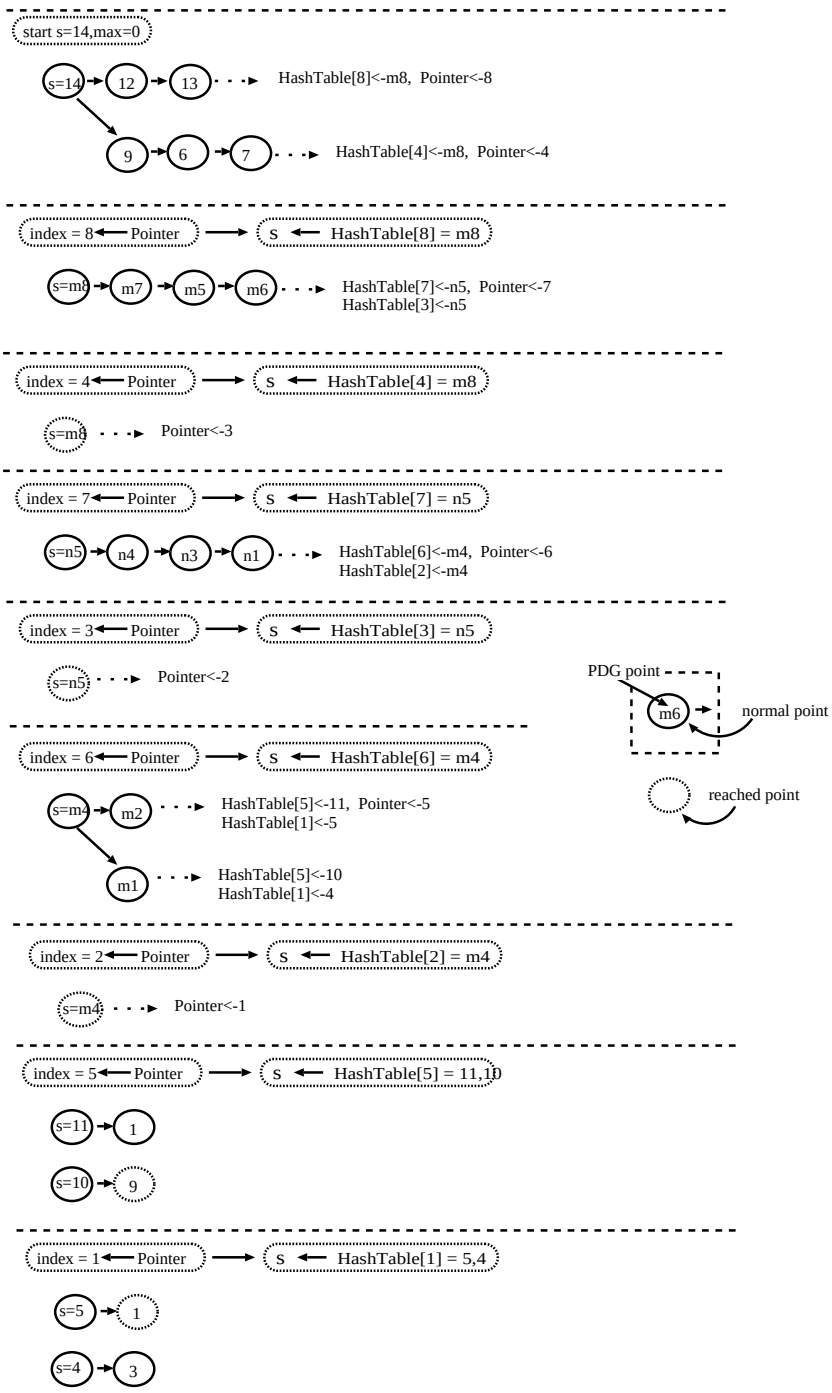

図 11: アルゴリズム適用例

4.6 評価実験

提案手法を Java プログラムスライスシステムに実装し，その評価を行なった．実験には，

- 本提案手法 (method-indexed Slice)
- 全ての依存辺に履歴を付加する手法 (all-indexed Slice)
- 履歴を付加しない手法 (DC Slice)

の 3 手法の比較を行なった．また，比較実験の対象となるプログラムには次のようなものを使用した．

- メソッド呼びだしが存在するプログラム
- ループが存在するプログラム
- ソートプログラム

まず，メソッド呼びだしが存在するプログラムを用いた実験では，本提案手法が Calling-Context 問題を解決できているかを確認する．次に，ループが存在するプログラムプログラムを用いた実験で，ループが存在する時の本提案手法のコストについて確認を行なう．最後にソートプログラムを用いて，一般的なプログラムに対してどのような結果が得られるかを確認する．なお，全ての実験においてスライス基準は同じ変数を指定するとする．

まず，4.6.1 節で全ての依存辺に履歴を付加する手法について簡単に説明を行なった後，それぞれの実験結果について考察を行なう．

4.6.1 全ての依存辺に履歴を付加する手法

この手法では，構築時に全ての依存辺に履歴の付加を行なう．プログラム実行時に，現在実行している文にあたる PDG 上の節点から次の節点に実行が移動するたびに index をインクリメントし，依存辺に付加する．PDG 探索時には，現在探索を行なっている節点から，探索可能な節点のうち，付加されている履歴が，今たどってきた index 値より小さい値の集合の中で，最大値をもつ依存辺を探索する．ただし，この手法では，依存辺の対応関係を履歴より調べることは不可能であるために，一度探索をおこなった節点にもう一度到達した場合でも，探索を続ける必要がある．

よって，探索アルゴリズムは次 図 12 のようになる．

アルゴリズムTrace all-indexed PDG

前提 全ての依存辺にindex情報を付加されている

入力 PDG上節点s, たどってきた依存辺のindex値max

出力 スライス結果

処理 PDGを探索しスライスを抽出

Algorithm Trace all-indexed PDG(s,max)

(1) **if** sがTに含まれない **then begin**

(2) $T \leftarrow T \cup \{s\}$

(3) **end**

(4) **while** sから新たに到達可能節点が存在する **then begin**

(5) **if** $index < max$を満たすindexの集合が存在 **then begin**

(6) index集合での最大値mについて Trace all-indexed PDG(t,m) を実行

(7) **end**

(8) **end**

図 12: アルゴリズムTrace all-indexed PDG

4.6.2 実験結果

- メソッド呼びだしが存在するプログラムに対する実験
  メソッド呼びだしが存在するプログラム (サイズ 21 行，呼び出し文 3 行) を用いた実験の結果は次の 表 3 ようになった．

表 3: メソッド呼びだしが存在するプログラムに対する実験

手法	method-indexed Slice	all-indexed Slice	DC Slice
スライスサイズ (行)	9	8	13
最大 index 値	6	20	-
PDG 構築時間 (ms)	3072	3566	2795
PDG 探索時間 (ms)	583	784	942

この実験結果より，Calling-Context を考慮した解析を行ない，スライスサイズが削減できていることが確認できた．さらにスライスサイズが小さくなった結果，PDG 探索の時間も短くなり，スライスを抽出するまでの時間コストは，履歴を付加しないものとほぼ同じとなった．

- ループが存在するプログラムに対する実験
  ループが存在するプログラム (サイズ 32 行) を用いた実験の結果は次の 表 4 ようになった．

表 4: ループが存在するプログラムに対する実験

手法	method-indexed Slice	all-indexed Slice	DC Slice
スライスサイズ (行)	14	14	14
最大 index 値	0	80	-
PDG 構築時間 (ms)	2769	3155	2718
PDG 探索時間 (ms)	1101	1230	1060

このプログラムでは，メソッド呼び出しが存在しないために，スライス抽出は履歴をつけないものとほぼ同じ結果となることが確認できる．しかし全ての依存辺に履歴を付加するプログラムでは，履歴を付加した分コストが大きなものとなっている．

- ソートプログラムに対する実験結果
  ソートプログラム (サイズ 241 行) を用いた実験の結果は次の 表 5 ようになった.

表 5: ソートプログラムに対する実験

手法	method-indexed Slice	all-indexed Slice	DC Slice
スライスサイズ (行)	42	40	83
最大 index 値	3316	19680	-
PDG 構築時間 (ms)	55817	331256	15016
PDG 探索時間 (ms)	7052	1275834	3489

  このソートプログラムは，再帰よるメソッド呼び出しやループを含んだプログラムである．よって，スライスを抽出するコストは，履歴を付加しない手法より本手法の方が大きなものとなっているが，この結果からはスライスサイズをほぼ半分に押えることができた．単純に計算して，スライスサイズを 1 行小さくするのに，時間コストが約 1 秒ほどのコスト増に押えられている．ループ・再帰がないようなプログラム (一般にはあまりないかもしれないが) では，さらにコストの削減ができると思われる．

デバッグにおいて，スライスサイズが小さければ小さいほど，デバグにかかるコストも少なくすむ．ループ・再帰が存在するプログラムに対してでも，スライスサイズを 1 行小さくするのに，時間コストが約 1 秒ほどのコスト増に押えられていることから，この実験結果に関すれば，本手法は有効であると思われる．

4.6.3 考察

上記の実験に対しては，比較的有効的な結果が得られた．ただ，時間の都合上実験が十分に行なうことができなかったため，この実験結果だけで，あらゆる場合において本提案手法が有効であると結論づけることはできない．しかし少なくとも，本提案手法を用いることで，Calling-Context を考慮した解析を行ない，スライスサイズの削減を実現していることは確かである．

デバッグにおいて，プログラムスライスを利用する場合，当然スライスサイズが小さいほど，デバッグが効率良く行なえる．先ほど記述したように，スライスサイズの削減にのみ注目すると，本提案手法は有効であるといえる．しかし，スライスの計算時間等含めた全ての

デバッグコストに対して，本提案手法が本当に有効であるかどうかを調べる必要があり，今後の課題として挙げられる．

5　スライス情報表示ツール

本節では，3 で説明した Java プログラムスライスシステムのユーザインターフェース部 (GUI) の説明を行なう．

5.1　概要

プログラム開発を支援するために，プログラムに関する情報を視覚的に表現することをプログラムビジュアライゼーションと呼ぶ．プログラムビジュアライゼーションは，情報検索，データ解析，プログラムデバッグなどに幅広く利用されており，プログラムスライス *(Prgram Slice)*[1] の分野においても利用されている．例としては，SeeSoft[7] をベースにして AT&T で開発された SeeSlice[8] などがある．SeeSlice で用いられている表示方法には，次のようなものがある．[9]

- プログラムを三階層 (ファイル，手続き，文) に分け，階層別に管理して表示
- 依存グラフ上でのスライス基準からの距離に応じた強調表示
- スライスを含まない手続きに関する情報の非表示
- スライスの全体像を表示するウインドウ
- 局部的なソースコードを表示するウインドウ

これらの手法はスライスの表示方法として，ビジュアライゼーションに必要な技法のうち，「必要なものみ表示する技法」と「全体と詳細を同時に見る方法」において特に優れているため，スライスの表示においても応用できる部分を多く含んでいると思われる．

ところが，SeeSlice で用いられている手法は，手続き型言語を対象としているが，今回スライス情報の表示を行なう対象となる言語はオブジェクト指向言語の Java である．そのため，オブジェクト指向の特徴を生かしながら表示することが求められる．さらに，動的情報を利用してスライスを求めるため，プログラムのコンパイル・実行ができなければならない．

以上より，本研究では次のような目的を持って、解析部から渡されたスライス情報の表示 (図 13 の GUI 部の作成) を行なった。

目的 **1:** ツール内からプログラムのコンパイル・実行

目的 **2:** スライス結果の効果的な表示 (「必要なもののみ表示する技法」の実現)

目的 **3:** 複数のファイルからなるプログラムのスライス結果全体像の把握 (「全体と詳細を同時に見る方法」の実現)

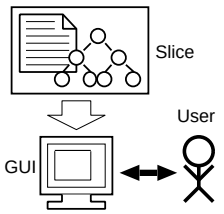

図 13: システム構成図 (略図)

これらの目的を実現するための機能を 5.2 で、その機能を実際に使用した例を画面写真とともに紹介する。

5.2 表示ツールの機能

Java スライスシステムにおける GUI の試作 [5, 11] を行なうにあたり，開発言語に Java[12, 13, 14] を使用し，ソースサイズは約 3100 行となった．本節では，Java プログラムにおけるスライスを表示させるために重要となる機能についての説明を行なう．

5.2.1 ツール内からプログラムのコンパイル・実行

ツール内で作成した Java プログラムおよび，開いた Java プログラムを，ツール内から実行 (コンパイル) することができる．メニューバーで，

「**Java**」⇒「**Execute**」**(Compile)**

を選択すると， 図 14 のように，実行コマンドを入力するためのフレームが表示される．このフレームの 「**Reference**」ボタンをクリックすると，実行したいファイルを選択することができ，選択すると， 図 14 のように，ディレクトリ，実行ファイル等が表示される．最後に **Execute Command** 欄に必要な引数等を記述し，OK ボタンをクリックすると，実行が開始する．

5.2.2 スライス結果の効果的な表示

解析の結果得られるスライス集合は，プログラム中のユーザが着目したい部分である．プログラム中のスライス集合に含まれる部分をテキスト上で強調表示することにより，スライス集合を容易に区別することが可能である．実現した強調表示は，通常テキストの背景色が白なのに対し，スライス集合部分の背景色に色を挿入することにより実現した．(図 15 参照)

ここで，ツールを用いて表示させるスライスを解析・表示する手順を説明する．メニューで

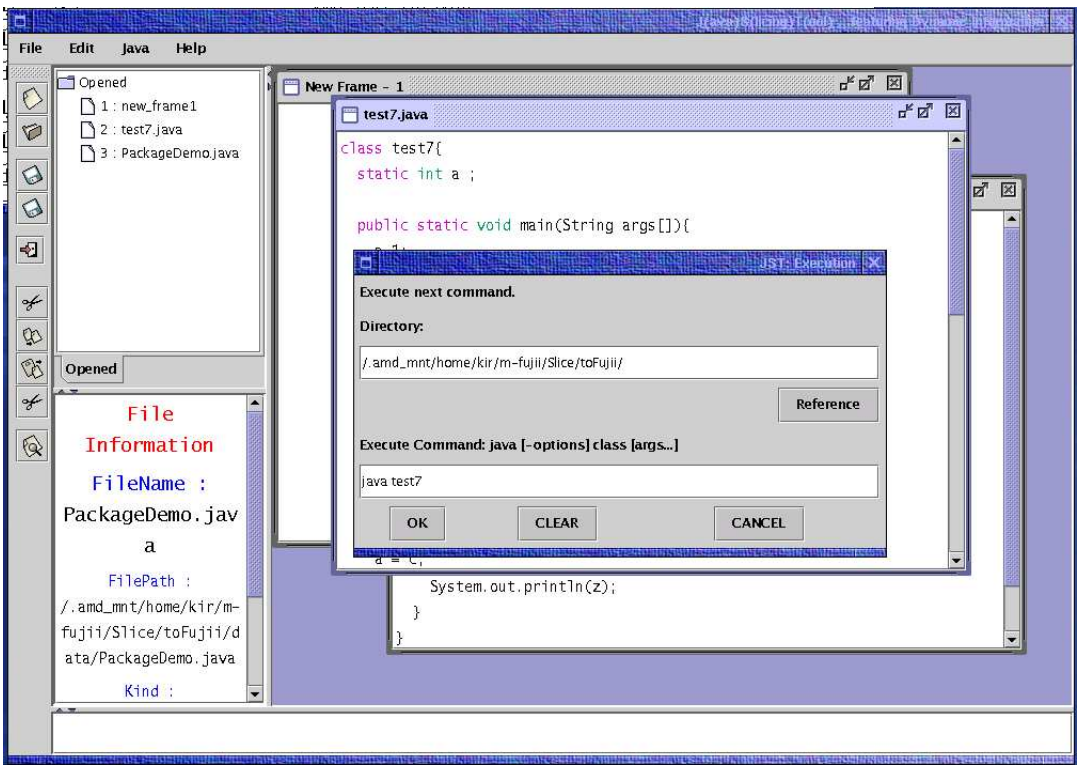

図 14: 実行

「**Java**」⇒「**Slice**」⇒「**Translate**」
を選択することにより，プログラムの解析が実行される．解析の対象となる Java プログラムは，アクティブなプログラム (一番前に表示されているプログラム) である．解析の終了後，PDG を構築する．これは，メニューで
「**Java**」⇒「**Slice**」⇒「**Get PDG**」
を選択するか，または解析直後に出力される「Complete Translate.Get PDG at once?」メッセージで「はい」を選択することで実行される．PDG を構築後，スライス基準を選択することで，スライス結果を得ることが可能となる．スライス基準の選択は，基準としたい変数上で，マウスの真中ボタンをクリックするか，基準にしたい変数を選択し，メニューで
「**Java**」⇒「**Slice**」⇒「**Criterion**」
を選択することにより，選択できる．さらに，選択した変数が配列のときは，配列の変数名だけか，その要素まで基準に含めるかを選択する (図 16 参照)．スライス基準の選択後，PDG をたどることにより，スライス集合を得ることができ， 図 16 のようにスライス結果として表示する．

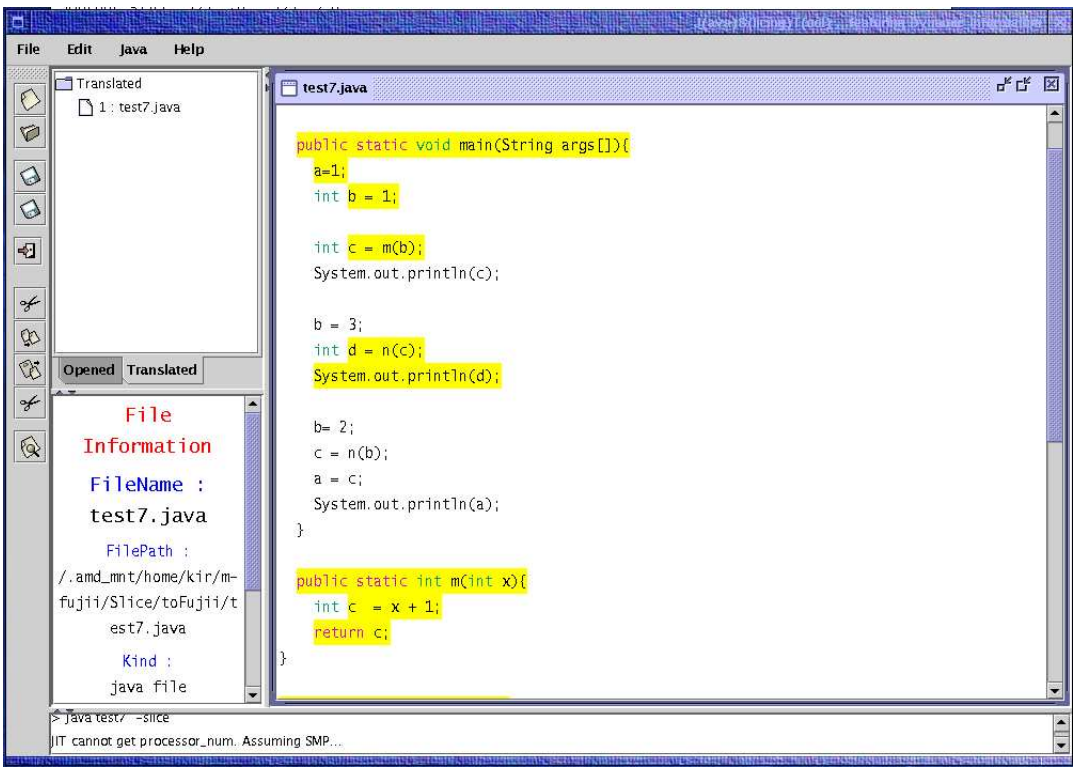

図 15: 実行

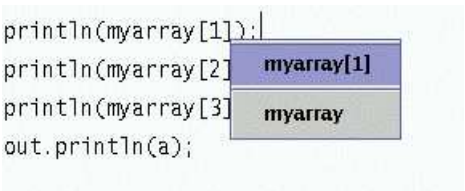

図 16: 配列の選択

5.2.3 複数のファイルからなるプログラムのスライス結果全体像の把握

Java プログラムが複数のファイルにわたる場合，開いていないファイルのプログラムについては，当然ながらスライス結果を表示させることができない．また，複数のファイルを開いている場合でも，どのファイルが解析プログラムと関連があるのか分からないと，効率が悪くなる．よって，解析したプログラムがどのファイルにわたっているかを表示させることにより，スライス結果全体像の把握を容易にできるようになる．

作成した GUI では，プログラム解析時にどのファイルを解析したかをツリーとして表示させることにより実現した． 図 17 において，右上のウインドウがそれに当たる．ツリーのノードには解析したファイル名が表示されており，ノード選択時に，選択したファイルがすでに開かれている時は，そのファイルをアクティブにし，開かれていない時は，そのファイルを開くかどうかの確認メッセージが出力される．

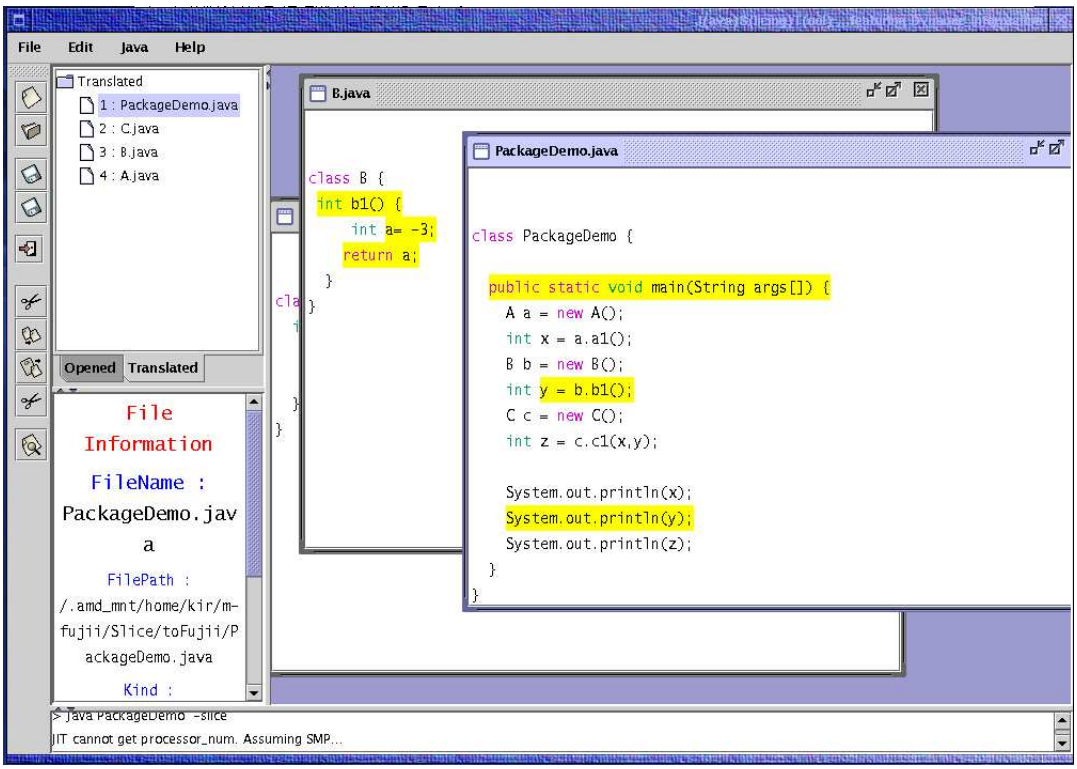

図 17: 複数ファイルからなるプログラムの解析結果

6　まとめと今後の課題

本研究では，メソッド間データ依存関係を動的に解析し，スライスサイズの削減を行なう手法を提案した．そして，本手法を，Java スライスシステムに実装し，他の手法との比較を行ない，本手法が Calling-Context を考慮した解析を行ない，スライスサイズの削減を実現している確認を行した．

また，スライス結果を効果的に表示・利用できるように，システムにおけるユーザインターフェースの試作を行ない，Java プログラムにおけるスライスを表示させる機能の作成を行なった．

今後の課題として，

- メソッド間データ依存関係解析手法
  - 大規模なプログラムに対する，本手法の有効性の評価
  - ループ・再帰を含むようなプログラムに対する解析の効率化
- ユーザインターフェース
  - より詳細な解析結果の表示（変数の定義されているメソッド名，行番号など）

などを行なうことがが挙げられる．

謝辞

本論文の作成において，常に適切な御指導および御助言を賜りました大阪大学大学院 基礎工学研究科 情報数理系専攻 ソフトウェア科学分野井上克郎 教授に深く感謝致します.

本論文の作成において，常に適切な御指導を賜りました 同 楠本真二 助教授に深く感謝致します.

本論文の作成において，常に適切な御指導を賜りました 同 松下誠 助手に深く感謝致します.

本論文の作成において，常に適切な御助言を頂きました 同 大畑文明氏に深く感謝致します.

本論文の作成において，常に適切な御助言を頂きました 同 廣瀬航也氏に深く感謝致します.

最後に，その他の面で様々な御指導，御助言を頂いた大阪大学大学院 基礎工学研究科 情報数理系専攻 ソフトウェア科学分野井上研究室の皆様に深く感謝致します.

参考文献

[1] Weiser M. : "Program Slicing", *Proceedings of the Fifth International Conference on Software Engineering*, pp. 439–449, 1981.

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

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

[4] Horwitz S. and Reps T. : "The Use of Program Dependence Graphs in Software Engineering", *Proceedings of the 14th International Conference on Software Engineering*, pp. 392–411, 1992.

[5] 佐藤, 飯田, 井上: "プログラムの依存解析に基づくデバッグ支援ツールの試作", 情報処理学会論文誌, *Vol. 37, No. 4*, pp. 536–545, 1996.

[6] 植田, 練, 井上, 鳥居: "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, No. 1*, pp. 11–22, 1995.

[7] Steffen G., Eick E., and Sumner Jr.: " Seesoft - a tool for visualizing line-oriented software statistics", *IEEE Trans. on Software Engineering*, Vol. 18, No. 11, pp.957–968, 1992.

[8] Thomas B., and Stephen G.E. : "Visualizing Program Slices", *in Proceedings of the 1994 IEEE Symposium on Visual Languages*, pp.288–295, 1994.

[9] 増井 俊之 : "情報視覚化の最近の研究動向", 電子情報通信学会第 *9* 回データ工学ワークシップ, 1998.

[10] 廣瀬 航也 : "静的情報と動的情報を用いた Java プログラムスライス計算法", 大阪大学基礎工学研究科修士学位論文, 2001.

[11] 近藤 和弘 : "JAVA プログラムのエイリアス情報表示ツールの試作", 大阪大学基礎工学部特別研究報告, 2000.

[12] Gosling J., Joy B. and Steele G., 村上 雅章 [訳] : "The Java 言語仕様", Addison-Wesley Publishers Japan, Ltd., 1997.

[13] Joseph O., トップスタジオ [訳] : “独習 Java”, SHOEISHA, 1999.

[14] Satyaraj P., 岩谷 宏 [訳] : “速習 JavaSwing プログラミング”, SOFTBANK publishing, 1999.

[15] Java 用語研究会 : “Java 用語辞典”, 技術評価社, 2000.