修士学位論文

題目

エイリアス関係を考慮した
Java プログラム用静的スライシングツール

指導教官
井上 克郎 教授

報告者
山中 祐介

平成 15 年 2 月 12 日

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

平成 14 年度 修士学位論文

エイリアス関係を考慮した Java プログラム用静的スライシングツール

山中 祐介

内容梗概

近年の大規模化・複雑化するプログラムに対して，プログラムスライシングと呼ばれる手法がプログラムの理解，デバッグの効率化を目的として提案されている．プログラムスライシングとは，プログラム文中においてある文のある変数の値に影響を与える文の集合を抽出する技術であり，これを利用することでプログラム中の特定の機能やエラーに関係がある可能性の高い部分を抽出することができる．これまでオブジェクト指向言語において，様々なプログラムスライシング手法の提案・実現がなされてきた．しかし，既存の手法は同一メモリ空間を指す可能性のある式間の同値関係であるエイリアス関係を十分に考慮されておらず，エイリアス関係の解析について曖昧なものがほとんどである．

本研究では，オブジェクト指向言語 Java を対象として，エイリアス関係を考慮した静的プログラムスライス計算手法を提案する．エイリアス関係を利用することで，オブジェクト指向言語特有の実行時決定要素の解析を含めた，より詳細なプログラムスライスの計算が期待できる．また，提案手法を我々の研究グループが開発している Java プログラム解析フレームワークにスライシングツールとして実装を行い，その有効性を解析コスト及び正確性の面から確認した．その結果，エイリアス関係がもたらす依存関係は数多く存在するため，エイリアス関係を考慮することが重要であることがわかった．このことから，オブジェクト指向言語に対して正確なスライス計算を行う上で提案手法は有効であるといえ，Java プログラムの理解や保守に十分な効果が得られると考えられる．

主な用語

プログラムスライス (Program Slice)
エイリアス (Alias)
静的解析 (Static Analysis)
Java

目 次

1 まえがき **4**

2 プログラムスライス **6**

2.1 プログラムスライス計算法 6

2.1.1 Phase1: 定義，参照変数の抽出 6

2.1.2 Phase2: 依存関係解析 6

2.1.3 Phase3: プログラム依存グラフの構築 9

2.1.4 Phase4: プログラム依存グラフによるスライス抽出 9

2.2 スライスの静的解析と動的解析 10

3 エイリアス **12**

3.1 エイリアスとプログラムスライス 12

3.2 エイリアスとコンパイラ最適化 12

3.3 FI エイリアス解析と FS エイリアス解析 14

4 エイリアス関係を考慮したプログラムスライス **16**

4.1 方針 16

4.2 アルゴリズム 18

4.2.1 Phase1: 定義，参照変数及びエイリアス集合の抽出 19

4.2.2 Phase2: 依存関係解析 19

4.3 適用例 24

5 スライシングツールの実現 **31**

5.1 Java プログラム解析フレームワーク 31

5.2 スライシングツール 32

6 考察 **36**

6.1 評価 36

6.1.1 スライスの定義 36

6.1.2 解析コスト 36

6.1.3 スライスサイズ 38

6.2 関連研究 38

7 むすび **39**

謝辞 **40**

参考文献 **41**

1　まえがき

近年，プログラム言語自身の高級化が進むとともに，プログラムの大規模化・複雑化が著しく進行している．そのようなプログラムに対して効率良く理解や保守などを行うための分野として**プログラム解析**（*Program Analysis*）がある．プログラム解析に用いられる手法には，**データフロー解析**（*Data Flow Analysis*）や**エイリアス解析**（*Alias Analysis*）などがあり，それらを利用してプログラムスライシングが実現されている．

プログラムスライシング（*Program Slicing*）は Weiser[12] により提案されたものであり，プログラム中のある文 s における変数 v（**スライス基準**（*Slice Criterion*）$< s, v >$ と呼ぶ）に対して v の値に影響を与え得る全ての文を抽出する技術である．結果として抽出された文の集合を**プログラムスライス**（*Program Slice*），または単にスライスと呼ぶ．プログラムスライスを用いることで，プログラムに存在する特定の機能やエラーに関係がある可能性の高い部分を抽出することができる．その結果，プログラムの理解やデバッグ時におけるエラー位置の特定などにかける負担が軽減でき，開発効率の向上が期待できる．

一般にスライスはプログラム文間に存在する**依存関係**（*Dependence Relation*）を解析することによって得られる**プログラム依存グラフ**（*Program Dependence Graph*）[16] を用いて計算される．依存関係は，プログラムの制御構造から計算される**制御依存関係**（*Control Depence Relation*）と変数の参照・定義の情報を用いて計算される**データ依存関係**（*Data Dependence Relation*）の 2 種類ある．また，スライスはすべての入力データ及び実行制御の可能性を考慮した依存関係解析による**静的スライス**（*Static Slice*）と，ある特定の入力データにおける実行系列の依存関係解析による**動的スライス**（*Dynamic Slice*）の 2 つ分けられる．本研究では静的スライスに着目する．

エイリアス解析は主にコンパイラ最適化 [1] に用いられ，エイリアス関係をプログラムの静的解析により求めることを指す．**エイリアス関係**（*Alias Relation*）とは，プログラム上の式（または部分式）の対が同一オブジェクト（メモリ領域）を指す可能性のある式の同値関係であり，エイリアス関係の集合を**エイリアス集合**（*Alias Set*）と呼ぶ．また，エイリアス関係は，引数の参照渡し，参照変数，ポインタを介した間接参照などによって生じるため，C++[3] のようなポインタを持つ言語や参照変数が数多く存在するオブジェクト指向言語の Java[11] などにおいては，プログラム理解にも効果的に利用することができる．

Java などのオブジェクト指向言語における，エイリアス関係が存在するようなプログラムに対して静的スライスの計算を行うためには，データ依存関係の解析において変数の参照・定義の情報を用いる際にその変数がどのエイリアス集合に属しているかを知る必要がある．エイリアス関係を無視して依存関係の抽出を行うと，エイリアス関係により異なる変数で同一のメモリ空間を変更した場合などにおいて，データ依存関係を正しく把握できないため，

著しく正確性に欠く場合がある．そのため，静的スライスの計算にはエイリアス解析を事前に行っていることが必要不可欠である．これまでオブジェクト指向言語に対して様々なスライス計算手法の提案及び実装がなされてきたが，既存のスライス計算手法はプログラム中に存在するエイリアス関係を十分に考慮されていない，もしくはエイリアス関係を考慮されていない．また，エイリアス関係が考慮されていてもその実装について曖昧にしているものがほとんどである．

そこで，本研究ではオブジェクト指向言語において，エイリアス関係を考慮した静的プログラムスライス計算手法を提案する．エイリアス関係を利用することで，既存の手法では抽出することができなかった依存関係の抽出が可能になる．その結果，より精度の高いプログラムスライスの計算を行うことが期待できる．

また，提案手法の実装を Java に対して行う．オブジェクト指向言語には，従来の手続き型言語にはない概念（**クラス**（*Class*），**オブジェクト**（*Object*），**継承**（*Inheritance*），**動的束縛**（*Dynamic Binding*），**多態**（*Polymorphism*）など）が導入されており，これらはプログラムの実行時に決定する要素である．静的にプログラムの解析を行うためにはエイリアス解析が有効 [2][13] とされているため，オブジェクト指向言語を対象としたプログラムスライスの計算には，エイリアス関係を考慮することが重要であると考えられる．我々の研究グループで開発を行っている Java プログラム解析フレームワークにスライシングツールを機能拡張することで実現し，その有効性を得られたスライスの妥当性，解析コスト，スライスサイズの 3 点から確認する．

以降，2 ではプログラムスライス及び既存の計算手法について述べ，3 ではエイリアスについて述べる．4 では提案手法であるエイリアス関係を考慮したスライス計算法について述べ，5 で Java プログラム解析フレームワークへの提案手法の実装について述べる．6 で提案手法と実装したスライシングツールの評価について述べ，最後に 7 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

プログラムスライス（*Program Slice*）は Weiser[12] により提案された．このプログラムスライスとは，**スライス基準**（*Slice Criterion*）（対 $< s, v >$ で示され，s は文，v は s で定義もしくは参照される変数を表す）に影響を与える可能性のある全ての文をプログラムから抽出する技術で，その結果取り出された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．

プログラムスライスは，プログラム中に存在するフォールトの位置特定に有効であるだけでなく，プログラム保守，プログラム理解などにも利用される．図 1 に，サンプルプログラム及びスライス基準 <16, b> に対するスライスを示す．

2.1　プログラムスライス計算法

スライス計算にはさまざまな手法が提案されてきたが，本研究ではプログラム依存グラフによるスライス抽出技法 [16] に着目する．この手法は，次の 4 フェーズで構成されている．

Phase1: 定義，参照変数の抽出

Phase2: 依存関係解析

Phase3: プログラム依存グラフの構築

Phase4: プログラム依存グラフによるスライス抽出

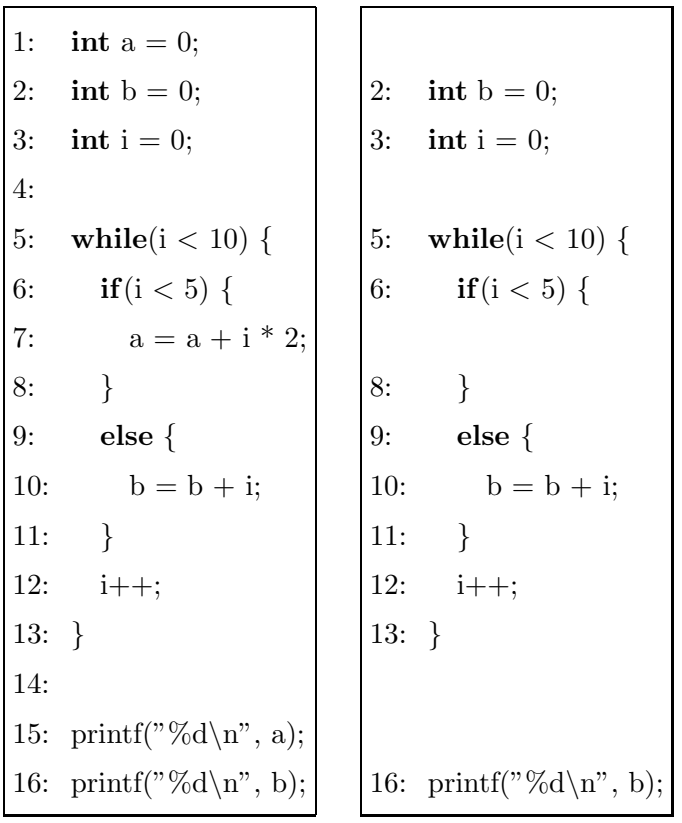
以下，各フェーズについて説明する．ただし，プログラムの構文解析，意味解析により**制御フローグラフ**（*Control Flow Graph, CFG*）[1] が得られているものとする．なお，ここで対象とする言語はポインタのない手続き型言語とする．

2.1.1　Phase1: 定義，参照変数の抽出

プログラムの各文で定義，参照されている変数を抽出する．このフェーズはプログラム文を一度走査するだけで終らせることができる．図 1 のプログラムにおける各文の定義変数，参照変数を表 1 に示す．

2.1.2　Phase2: 依存関係解析

Phase1 の結果を元に，プログラム文間に存在する**制御依存関係**（*Control Dependence Relation, CD Relation*），**データ依存関係**（*Data Dependece Relation, DD Relation*）[16] という 2 つの依存関係を抽出する．



サンプルプログラム　　<16, b> に対するスライス

図 1: スライスの例

制御依存関係

制御依存関係とは，プログラム中の 2 文 s, t に関して，以下の条件を満たすときに存在する．

1. s は条件文（節）

2. t の実行は s の判定結果に依存する

制御依存関係を $CD(s,\ t)$ と表し，表 2 に図 1 のプログラムの制御依存関係を表す．

制御依存関係の計算は Phase1 の結果及び制御フローグラフから容易に行うことができる．

データ依存関係

データ依存関係とは，プログラム中の文 s から文 t の間に変数 v に関して，以下の条件を満たすときに存在する．

表 1: 図 1 の各文に対する定義，参照変数

文	定義変数	参照変数
1	a	
2	b	
3	i	
5		i
6		i
7	a	a, i
10	b	b, i
12	i	i
15		a
16		b

1. s は v を定義する
2. t は v を参照する
3. s から t への 1 つ以上の実行経路の中に，v の再定義が起こらない経路が少なくとも 1 つ存在する

データ依存関係を $DD(s,\ v,\ t)$ と表し，表 3 に図 1 のプログラムにおけるデータ依存関係を示す.

データ依存関係の計算は，文 t が変数 v を参照したときに変数 v を定義した文 s がどこであることがわかれば計算することができる．そのために，Phase1 の結果を用いてプログラム文中のある変数 v に対する**変数表**（*Variable Table*）$VT(v)$ を用意する．$VT(v)$ は変数 v が最後に定義された文番号を保持しており，あるプログラム文 s において次のような処理が行われる.

- 文 s で変数 v が定義された場合：
  $VT(v)$ の値を文 s の文番号に更新する.
- 文 s で変数 v が参照された場合：
  $VT(v)$ の文番号に対応する文 t から文 s に変数 v に関するデータ依存関係があることを抽出する.

図 2 に図中のサンプルプログラムにおける変数表 $VT(v)$ の推移表を示す.

表 2: 図 1 の制御依存関係

CD(5, 6), CD(5, 12), CD(6, 7), CD(6, 10)

表 3: 図 1 のデータ依存関係

DD(1, a, 7), DD(1, a, 15), DD(2, b, 10)
DD(2, b, 16), DD(3, i, 5), DD(3, i, 6)
DD(3, i, 7), DD(3, i, 10), DD(3, i, 12)
DD(7, a, 7), DD(7, a, 15), DD(10, b, 10)
DD(10, b, 16), DD(12, i, 5), DD(12, i, 6)
DD(12, i, 7), DD(12, i, 10), DD(12, i, 12)

実際に，データ依存関係の計算を行うためには，手続き呼び出し文が呼び出している手続き本体の解析が必要不可欠である．また，繰り返し文，再帰呼び出しによる再帰的な依存関係の計算も必要であるので，スライス抽出過程の中でも依存関係解析（特にデータ依存関係解析）は最も計算量を必要とするフェーズである．

2.1.3 Phase3: プログラム依存グラフの構築

Phase2 で抽出された依存関係を利用し，プログラム依存グラフを構築する．**プログラム依存グラフ**（*Program Dependence Graph, PDG*）[16] とは，プログラム内の文間の依存関係を表す有効グラフであり，その節点は，プログラムに存在する文（条件判定文，代入文，入出力文，手続き呼出文）を表し，その有向辺は 2 つの節点間の依存関係（制御依存関係，データ依存関係）を表す．

制御依存関係の辺は**制御依存辺**（*Control Dependence Edge, CD Edge*），データ依存関係の辺は**データ依存辺**（*Data Dependence Edge, DD Edge*）と呼ぶ．また，関数間にわたるデータ依存関係を表現するために特殊節点及び特殊辺 [15] も存在する．

プログラム依存グラフは Phase2 で抽出した依存関係から容易に構築可能である．図 3 に図 1 のプログラムに対するプログラム依存グラフを示す．

2.1.4 Phase4: プログラム依存グラフによるスライス抽出

スライス基準に対するスライスを抽出する．スライス基準 $< s, v >$ に対するスライスとは，s に対応した PDG 中の節点 N_s から，逆方向に制御依存辺及びデータ依存辺を経て推移的に到達可能な節点集合に対応する文の集合をいう．

スライスの抽出は PDG を辿るのみであるため，このフェーズで要する計算量はわずかな

```
1:   a = 0;
2:   b = 1;
3:   a = b + 2;
4:   c = a + b;
5:   b = 2;
```

サンプルプログラム

文番号	a	b	c
1	1	-	-
2	1	2	-
3	3	2	-
4	3	2	4
5	3	5	4

変数表の推移表

図 2: 変数表の推移の例

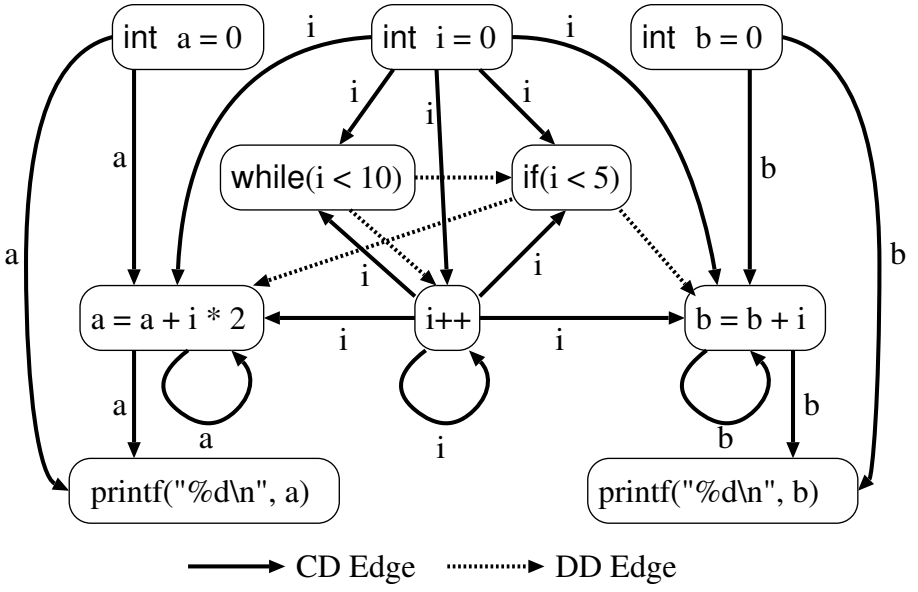


図 3: 図 1 のプログラム依存グラフ

ものである．図 4 に図 1 のプログラムにおけるスライス基準 <16, b> に対するスライス（網掛部）を示す．

2.2　スライスの静的解析と動的解析

スライスには，依存関係解析のフェーズにおける手法の違いにより，**静的スライス**（*Static Slice*）[12] と**動的スライス**（*Dynamic Slice*）[9] の 2 種類に大別される．

静的スライス

静的スライスとは，**静的解析**（*Static Analysis*）に基づき，入力データのすべての可能性を考慮した文間の依存関係が抽出される．その特徴としては，現実には短い時間で計算され

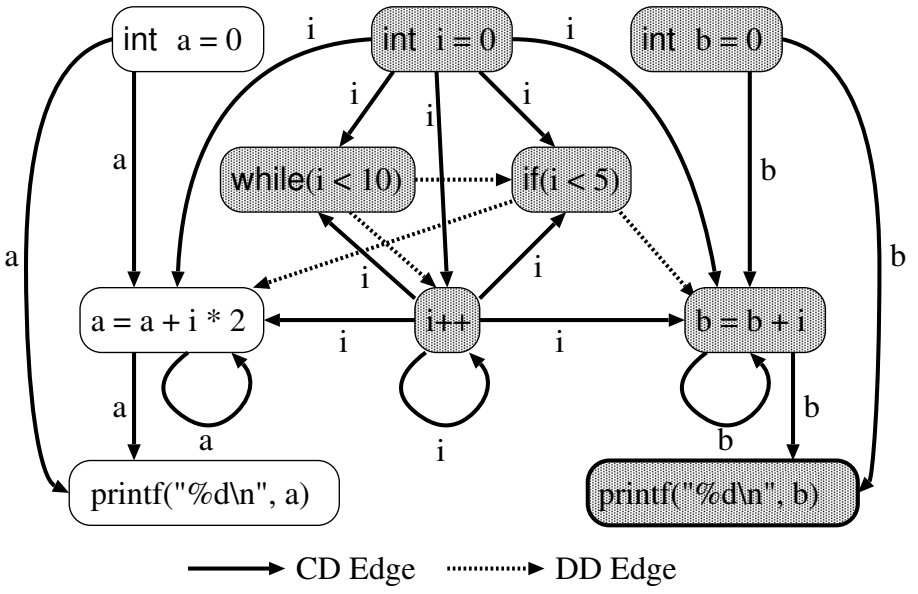


図 4: 図 1 のスライス基準 <16, b> に対するスライス

るというところにある．それに加え，プログラムに起こり得るすべての実行経路を考慮しているため，プログラムに存在する特定の機能を抽出したい場合に有効である．しかし，プログラム実行においてすべての実行経路が利用されるとは限らないため，実行時エラーの原因を把握するためのフォールト位置特定に対しては必ずしも効果的とはいえない．

動的スライス

動的スライスとは**動的解析**（*Dynamic Analysis*）に基づき，ある特定の入力データによる実行系列間の依存関係が抽出される．その特徴としては，解析対象を特定の実行経路に限定しているため，静的スライスに比べてスライスのサイズが小さくなるというところにある．また，実際に実行された部分の中からのみスライスが計算されるため，フォールト位置特定を効率よく行なうことができる．しかし，動的スライスの計算には，実行系列と依存関係を全て記憶しなければならないため，多大な時間及び空間コストを要するという問題点がある．

本研究の目標の一つとして，現実的な時間及び空間コストでのプログラム解析，ということを挙げている．その点から本研究では静的スライスに着目する．以降，スライスと呼ぶとき，特に断りがない限り静的スライスを指す．

3　エイリアス

エイリアス（*Alias*）とは，引数の参照渡し，参照変数，ポインタを介した間接参照などで生じる，同じメモリ領域（オブジェクト）を指す可能性のある式間の同値関係であり，**エイリアス関係**（*Alias Relation*）ともいう．エイリアス関係は同値関係であり，その同値類を**エイリアス集合**（*Alias Set*）と呼ぶ．また，プログラムの静的解析を行うことでエイリアス集合を求めることを**エイリアス解析**（*Alias Analysis*）といい，コンパイラ最適化 [1] や，プログラムスライスの計算欠くことのできないものである．

図 5 にエイリアスの例を示す．`Sample` 型の変数 `a`, `b` はともに参照変数であり，4 行目の代入によってこの 2 つの変数の間にエイリアス関係が生じている．図 5 中の網掛部がエイリアス集合である．

3.1　エイリアスとプログラムスライス

2 で述べたように，ポインタのない手続き言語におけるプログラムスライス計算は

Phase1: 定義，参照変数の抽出

Phase2: 依存関係解析

Phase3: プログラム依存グラフの構築

Phase4: プログラム依存グラフによるスライス抽出

という過程をたどる．ポインタ（参照）が存在するプログラムにおけるプログラムスライス計算では，Phase2 中のデータ依存関係解析において，各文でどの変数が定義，参照されているかだけでなくその変数がエイリアス集合に含まれているかという情報が必要になる．そのためデータ依存関係解析を行う際にはエイリアス解析が前提となっている．なぜならば，エイリアスの存在により，プログラムの異なるスコープ中の異なる識別子が同じメモリ領域を指す可能性があるため，エイリアス解析なしにデータ依存関係を抽出することはできない．

例えば，図 5 のプログラムにおいて，9 行目の変数 `n` に対してデータ依存関係を抽出する場合，変数 `a`, `b` にはエイリアス関係が生じているため変数 `n` は 6 行目ではなく 7 行目で定義されていると判定されなければならない．

3.2　エイリアスとコンパイラ最適化

コンパイラ最適化でのエイリアス解析の利用例を図 6 を用いて示す．

```
1:  class Sample {
2:      int n;
3:
4:      void increment() {
5:          n++;
6:      }
7:  }
```

```
1:  class Test {
2:      public static void main(String[] args) {
3:          Sample a = new Sameple();
4:          Sample b = a;
5:
6:          a.n = 3;
7:          b.increment();
8:
9:          System.out.println("a.n : " + a.n);
10:         System.out.println("b.n : " + b.n);
11:     }
12: }
```

図 5: エイリアスの例

```
1:  int a[], b[];
2:  void f(int i, int j) {
3:      int *p, *q;
4:      int x, y;
5:      p = &a[i];
6:      q = &b[j];
7:      x = *(q + 3);
8:      *p = 5;
9:      y = *(q + 3);
10:     g(x, y);
11: }
```

(a) プログラム（最適化前）

```
1:  int a[], b[];
2:  void f(int i, int j) {
3:      int *p, *q;
4:      int x;
5:      p = &a[i];
6:      q = &b[j];
7:      x = *(q + 3);
8:      *p = 5;
9:
10:     g(x, x);
11: }
```

(b) プログラム（最適化後）

図 6: (例) エイリアス解析とコンパイラ最適化

これは C 言語で書かれたサンプルプログラムであるが，解析によりポインタ変数 `p`，`q` はエイリアスを生成しないと判断でき，文 `y = *(q + 3)` の省略及び変数 `y` の変数 `x` への置き換えが可能となる．

```
1:  Integer a, b, c;
2:  a = new Integer(1);
3:  b = new Integer(2);
4:  c = b;
5:  System.out.println(c);
6:  c = a;
7:  System.out.println(c);
```

図 7: FI エイリアス解析

図 8: 図 7 の変数 c（7 行目）に関するエイリアスグラフ

3.3 FI エイリアス解析と FS エイリアス解析

エイリアス集合を導き出すエイリアス解析は，大きく **FI エイリアス解析**（*Flow-Insensitive Alias Analysis*），**FS エイリアス解析**（*Flow-Sensitive Alias Analysis*）の 2 つに分けることができる．以下，それぞれについて説明する．

FI エイリアス解析

FI エイリアス解析 [2] とは，プログラム文の実行順を考慮しないエイリアス解析手法をいい，**エイリアスグラフ**（*Alias Graph*）を利用する．エイリアスグラフは無向グラフであり，グラフの節点はメモリ領域を指す可能性のある変数及び式を，辺は代入や引数の参照渡しなどにより節点間に直接のエイリアス関係があることを表す．

図 7 に図中の 7 行目の変数 c（太枠部）に関する FI エイリアス集合（網掛部）を，図 8 にエイリアス集合の計算に用いたエイリアスグラフを示す．

FS エイリアス解析

FS エイリアス解析 [14] とは，プログラム文の実行順を考慮したエイリアス解析手法をいい，**到達エイリアス集合**（*Reaching Alias Set, RAset*）を利用する．プログラムのある文 s における到達エイリアス集合 $RA(s)$ の要素は，次の条件を満たすエイリアス集合である．

1. 文 s の実行直前に成立している
2. 文 s において識別子を介して参照可能なエイリアス集合である

```
1:  Integer a, b, c;
2:  a = new Integer(1);
3:  b = new Integer(2);
4:  c = b;
5:  System.out.println(c);
6:  c = a;
7:  System.out.println(c);
```

図 9: FS エイリアス解析

表 4: 図 9 のプログラムにおける到達エイリアス集合

文 (s)	到達エイリアス集合 ($RA(s)$)
1	ϕ
2	ϕ
3	{[(2, a), (2, **new** Integer(1))]}
4	{[(2, a), (2, **new** Integer(1))], [(3, b), (3, **new** Integer(2))]}
5	{[(2, a), (2, **new** Integer(1))], [(4, c), (3, b), (4, b), (3, **new** Integer(2))]}
6	{[(2, a), (2, **new** Integer(1)), [(4, c), (5, c), (3, b), (4, b), (3, **new** Integer(2))]}
7	{[(6, c), (2, a), (6, a), (2, **new** Integer(1))], [(3, b), (4, b), (3, **new** Integer(2))]}

到達エイリアス集合の各要素は (文番号, 式) の組で表される．図 9 に図中の 7 行目の変数 c（太枠部）に関する FS エイリアス集合（網掛部）を，表 4 にエイリアス集合の計算に用いた到達エイリアス集合を示す．

データ依存関係の計算にはプログラム文の実行順を考慮して行わなければならない．そのことから，エイリアス関係をデータ依存関係の計算に用いるためには，プログラム文の実行順を考慮した FS エイリアス解析が有効である考えられる．そこで本研究では，FS エイリアス解析に着目する．以降，エイリアス解析と呼ぶとき，特に断りがない限り FS エイリアス解析を指す．

4　エイリアス関係を考慮したプログラムスライス

3 で述べたように，参照（引数の参照渡し，参照変数，ポインタによる参照）が生じるプログラム言語のプログラムスライスの計算を行う際にはエイリアス解析を行うこと必要不可欠である．スライス計算においてエイリアスを考慮しなければならない例として，図 5 のサンプルプログラムのスライス基準 <9, n>（太枠部）に対するスライス（網掛部）を図 10 と図 11 に示す．図 10 はエイリアスを無視した場合で，図 11 がエイリアスを考慮した場合である．変数 a, b は 4 行目の演算によってエイリアスが発生してるので，6, 7 行目で定義及び参照されているメンバ変数 n は同じメモリ領域にある．そのため，スライス基準になっているメンバ変数 n は変数 a だけでなく変数 b にも影響を受けている．よって，4, 7 行目もスライスに含まれるのである．

これまでに多くのスライス計算手法が提案されているが，エイリアス関係を十分に考慮されていない．そこで，本研究ではエイリアス関係を考慮した静的プログラムスライス計算手法を提案する．エイリアス関係の情報を用いることによって，既存の手法では抽出することができないようなデータ依存関係の抽出を行うことが可能になる．その結果として，より精度の高いプログラムスライスの計算を行えることが期待できる．

また，本研究ではオブジェクト指向言語である Java を解析の対象としている．オブジェクト指向言語には，従来の手続き型言語にはない**クラス**（*Class*），**オブジェクト**（*Object*），**継承**（*Inheritance*），**動的束縛**（*Dynamic Binding*），**多態**（*Polymorphism*）などの概念が導入されている．これら概念はプログラムの実行時に決定される要素であり，静的にプログラムの解析を行うためにはエイリアス解析が有用 [2][13] とされている．

このことからも，対象をオブジェクト指向言語としたプログラムスライスの計算を行う場合，エイリアス関係を考慮することが必要であると考えられる．

4.1　方針

まず，スライス計算を行うための前提として，解析対象プログラムのエイリアス解析が終了しており，ある式のエイリアス集合が計算可能になっているとする．また，解析アルゴリズムは単一メソッド内が対象でありそのメソッド内で発生するメソッド呼び出しに関しては，呼び出し対象メソッドの解析が終了しておりその解析情報を利用するものとする．

提案手法の方針

エイリアス関係を考慮したスライス計算を行うため，プログラム依存グラフによるスライス計算の 4 つの過程のうち Phase1 及び Phase2 のアルゴリズムに対して変更を行う．アルゴリズム変更の方針について以下に示す．

```
1:  class Sample {
2:      int n;
3:
4:      void increment() {
5:          n++;
6:      }
7:  }
```

```
1:  class Test {
2:      public static void main(String[] args) {
3:          Sample a = new Sameple();
4:          Sample b = a;
5:
6:          a.n = 3;
7:          b.increment();
8:
9:          System.out.println("a.n : " + a.n);
10:         System.out.println("b.n : " + b.n);
11:     }
12: }
```

図 10: スライスの比較（エイリアスなし）

```
1:  class Sample {
2:      int n;
3:
4:      void increment() {
5:          n++;
6:      }
7:  }
```

```
1:  class Test {
2:      public static void main(String[] args) {
3:          Sample a = new Sameple();
4:          Sample b = a;
5:
6:          a.n = 3;
7:          b.increment();
8:
9:          System.out.println("a.n : " + a.n);
10:         System.out.println("b.n : " + b.n);
11:     }
12: }
```

図 11: スライスの比較（エイリアスあり）

- **Phase1 の変更**
  データ依存関係解析を行う際にエイリアス解析が必要である．そこで変数の参照及び定義の抽出に加え，プログラムの各文に含まれる式がどのエイリアス集合に属しているかを抽出する．

- **Phase2 の変更**
  制御依存関係，データ依存関係の定義は 2 で述べたものから変わりはない．制御依存関係の計算は，Phase1 の結果及び制御フローグラフを用いて行う．データ依存関係の計算は，変数表 $VT(v)$ に加え，エイリアス解析によって得られた情報を用いて，ある式 e がどのエイリアス集合（集合を区別するため，エイリアス ID を用意）に含まれるかという情報を保持する**エイリアス表**（*Alias Table*）$AT(e)$，あるエイリアス ID i に対応するエイリアス集合の型が持つメンバ変数の定義情報を保持する**メンバ変数表**（*Member Variable Table*）$MT(i)$ を用意する．

オブジェクト指向言語への対応

また，Java などのオブジェクト指向言語が持つ特性に対する方針について簡単に述べる．

- **動的束縛，多態**
  継承は動的束縛，多態などの特性をもたらしている．実行時決定要素であるこれら特性は，エイリアス解析を行うことで対象のオブジェクトの型やメソッドをある程度正確に推測できる．よって，スライス計算における型特定はエイリアス解析の情報を用いる．

- **スレッド**
  一つの制御ブロックとして扱うとする．

- **例外**
  本研究が提案する手法は単一メソッド内の解析を前提としており，例外のような複雑な制御関係は考慮していない．

4.2 アルゴリズム

前述した方針に基づき，提案手法（変更を行ったスライス計算手順 Phase1，Phase2）のアルゴリズムについて述べる．また，ここで用いるサンプルプログラムを図 12 に示す．

```
1:  class Sam1 {
2:      public static void main(String[] args) {
3:          Sam2 a = new Sam2();
4:          Sam2 b = new Sam2();
5:          Sam2 c = b;
6:
7:          a.n = a.n + 1;
8:          System.out.println(c.n);
9:          c.n = c.n + 1;
10:         c = a;
11:         c.n = c.n + 1;
12:         System.out.println(c.n);
13:     }
14: }
```

```
1:  class Sam2 {
2:      int n;
3:  }
```

図 12: サンプルプログラム

4.2.1 Phase1: 定義，参照変数及びエイリアス集合の抽出

プログラムの各文で定義，参照されている変数を抽出することに加え，各文に含まれている式（型が参照型である変数，メソッドの戻り値が参照型であるメソッド呼び出し）が属しているエイリアス集合を抽出する．

図 12 のサンプルプログラムに対する各文の定義変数，参照変数を表 5 に，プログラムに対するエイリアス集合を表 6 に示す．中括弧の囲みが 1 つのエイリアス集合を表している．

4.2.2 Phase2: 依存関係解析

プログラムの各文の間に存在する制御依存関係とデータ依存関係の抽出を行う．方針で述べた通り制御依存関係の抽出方法に変更はないが，データ依存関係の抽出方法に関して変更がある．方針で挙げた 3 つの表（変数表，エイリアス表，メンバ変数表）が実際にはどのような情報を保持しているかについて説明したのち，データ依存関係の計算アルゴリズムについて述べる．

変数表 $VT(v)$

プログラム文中のある変数 v に対して以下の情報を保持する．

表 5: 図 12 の各文に対する定義，参照変数

文	定義変数	参照変数
3	a	
4	b	
5	c	b
7	n	a, n
8		c, n
9	n	c, n
10	c	a
11	n	c, n
12		c, n

表 6: 図 12 に対するエイリアス集合

{(3, a), (3, **new** Sam2()), (7, a), (7, a), (10, c), (10, a), (11, c), (11, c), (12, c)},
{(4, b), (4, **new** Sam2()), (5, c), (5, b), (8, c), (9, c), (9, c)}

- **文番号**
  解析しているプログラム文の時点で最後に定義された文番号を表す（複数可）

- **エイリアス ID**
  変数 v が参照変数であれば，解析しているプログラム文の時点でどのエイリアス集合に属しているかを表す

エイリアス表 $AT(e)$

プログラム文中のある式 e（型が参照型である変数参照，またはメソッドの戻り値が参照型であるメソッド呼び出し）に対して以下の情報を保持する．

- **エイリアス ID**
  式 e が属しているエイリアス集合を表す

メンバ変数表 $MT(i)$

あるエイリアス集合に対応するエイリアス ID i に対して，以下の情報を保持する．

- **メンバ変数**
  エイリアス ID i に対応したエイリアス集合の型が持つメンバ変数を表す（複数可）

D1: – 解析対象の文 s に対して，以下の入力を与える
　　・定義されている変数 v（必須）
　　・変数 v が属するエイリアス集合のエイリアス ID i（選択）
　if i が入力されている
D2:　– メンバ変数表 $MT(i)$ から v に対応した変数表 $VT(v)$ を得る
D3: – 変数表 $VT(v)$ の文番号を文 s の文番号に変更する
　if v は参照変数である
　　i を $VT(v)$ のエイリアス ID の値に変更
　　if エイリアス表 $AT(v)$ のエイリアス ID と $VT(v)$ が不一致
D4:　　– $VT(v)$ のエイリアス ID の値を $AT(v)$ のエイリアス ID の値に変更
　　if メンバ変数表 $MT(i)$ が存在しない
D5:　　– $MT(i)$ を作成，変数表の文番号は文 s の文番号

図 13: 変数定義時のアルゴリズム

- **変数表**
  エイリアス ID i に対応したエイリアス集合の型が持つメンバ変数 v の変数表 $VT(v)$ を表す（上記のメンバ変数と 1 対 1 で対応）

計算アルゴリズム

これら 3 つの表を用いてデータ依存関係の計算を行う．データ依存関係の計算において，処理を行わなければならないときはなんらかの変数が参照及び定義されているときである．Java プログラムにおいてそれに対応するのは，直接的なものとして各プログラム文での変数の参照及び定義であり，間接的なものとしてメソッド呼び出しによる対象メソッド内での参照，定義である．よって，変数の参照及び定義，そしてメソッド呼び出しの 3 つに関しての計算アルゴリズムが必要である．以下，3 つパターンについての計算アルゴリズムをそれぞれ図 14，図 13，図 15 に示す．

ただし，解析対象の文中において変数やメソッド呼び出しが演算子”.” を含んでいた場合，演算子”.” によって分割しそれぞれに対して計算アルゴリズムを適用するのではなく一番最初に参照される変数に対してのみ適用される．

R1: – 解析対象の文 s に対して，以下の入力を与える
・参照されている変数 v（必須）
・変数 v が属するエイリアス集合のエイリアス ID i（選択）
if i が入力されている
R2: – メンバ変数表 $MT(i)$ から v に対応した変数表 $VT(v)$ を得る
R3: – 変数表 $VT(v)$ の文番号に対応した文 t を得る，
t から s へ v に関するデータ依存関係が存在することを抽出
if v は参照変数である
i を $VT(v)$ のエイリアス ID の値に変更
if エイリアス表 $AT(v)$ のエイリアス ID と $VT(v)$ のエイリアス ID が不一致
R4: – $VT(v)$ のエイリアス ID の値を $AT(v)$ のエイリアス ID の値に変更，
i の値を $AT(v)$ のエイリアス ID の値に変更
if メンバ変数表 $MT(i)$ が存在しない
R5: – $MT(i)$ を作成，変数表の文番号は文 s の文番号
if v は演算子"." の左辺で右辺に変数 v' が存在し，参照されている
R6: – v' と i を入力として R1 に移行
else if v は演算子"." の左辺で右辺に変数 v' が存在し，定義されている
R7: – v' と i を入力として D1 に移行
else if v は演算子"." の左辺で右辺にメソッド呼び出し m がある
R8: – m と i を入力として M1 に移行

図 14: 変数参照時のアルゴリズム

M1: – 解析対象の文 s に対して，以下の入力を与える
・メソッド呼び出し m（必須）
・メソッド呼び出し m が属するエイリアス集合のエイリアス ID i（選択）
while m の引数 v_1 が存在する
if v_1 は m で参照されている
M2: – v_1 を入力として R1 に移行
else if v_1 は m で定義されている
M3: – v_1 を入力として D1 に移行
while m 中で参照されている m の呼び出し元のメンバ変数 v_2 が存在する
M4: – v_2 と i を入力として R1 に移行
while m 中で定義されている m の呼び出し元のメンバ変数 v_3 が存在する
M5: – v_3 と i を入力として D1 に移行
if m の戻り値が参照型である
M6: – i の値を $AT(m)$ のエイリアス ID に変更
if m は演算子"." の左辺で右辺に変数 v が存在し，参照されている
M7: – v と i を入力として R1 に移行
else if m は演算子"." の左辺で右辺に変数 v が存在し，定義されている
M8: – v と i を入力として D1 に移行
else if m は演算子"." の左辺で右辺にメソッド呼び出し m' がある
M9: – m' と i を入力として M1 に移行

図 15: メソッド呼び出し時のアルゴリズム

変数	文番号　エイリアス ID
a	- -

解析前

変数	文番号　エイリアス ID
a	3 1

解析後

図 16: 3 行目解析時における変数表

表 7: 3 行目解析時におけるエイリアス表

式	エイリアス ID
3: a	1
3: **new** Sam2()	1
7: a	1
7: a	1
10: c	1
10: a	1
11: c	1
11: c	1
12: c	1

4.3　適用例

図 12 のサンプルプログラムに対し提案手法を適用してスライスの抽出を行う．以降，3, 5, 7 行目において実際に実行される処理について述べる．

3 行目

• Phase1

1. 変数 a が定義されていることを抽出.
2. 変数 a やクラスインスタンス化である **new** Sam2() は参照型であるから，これらが含まれるエイリアス集合を抽出.

• Phase2

1. Phase1 の結果から変数表 $VT(v)$（図 16 の解析前）とエイリアス表 $AT(e)$（表 7）を作成.
2. メソッド呼び出し（クラスインスタンス化 **new** Sam2()）の処理.

表 8: 3 行目の解析終了時におけるメンバ変数表

エイリアス ID	変数	文番号　エイリアス ID
1	n	3　　-

M1: 入力 m は **new** `Sam2()` で，i はメソッドが直接呼び出されているので入力されない．

M2–M5: このメソッドは引数やメンバ変数に対する処理が行われないので実行されない．

M7–M9: 戻り値は参照型 `Sam2` であるが，直接呼び出されているので実行されない．

3. 定義された変数 `a` の処理．

D1: 入力 v は `a` で，i は直接定義されているので入力されない．

D2: 入力 i がないので実行されない．

D3: 変数表 VT(`a`) の文番号を 3 に変更．

D4: 変数 `a` は参照変数で AT(`a`) と $VT(a)$ のそれぞれのエイリアス ID が”1” と” なし” で一致しないので実行される．VT(`a`) のエイリアス ID を 1 に変更．

D5: メンバ変数表 $MT(1)$ は作成されていないので実行される．文番号は 3.

3 行目の解析が終了した時点での変数表は図 16 の解析後，メンバ変数表はは表 8 のようになる．

5 行目

4 行目の解析が終了した時点での変数表，エイリアス表，メンバ変数表はそれぞれ表 9，表 10，表 11 に示すようになっている．

• Phase1

1. 変数 `b` が参照，変数 `c` が定義されていることを抽出．

• Phase2

1. 参照されている変数 `b` の処理．

R1: 入力 v は `b` で，i は変数が直接参照されているので入力されない．

R2: 入力 i がないので実行されない．

R3: 4 行目から 5 行目へ変数 `b` に関するデータ依存関係があることを抽出．

表 9: 4 行目の解析終了時の変数表

変数	文番号	エイリアス ID
a	3	1
b	4	2

表 10: 4 行目の解析終了時のエイリアス表

式	エイリアス ID	式	エイリアス ID
3: a	1	4: b	2
3: **new** Sam2()	1	4: **new** Sam2()	2
7: a	1	5: c	2
7: a	1	5: b	2
10: c	1	8: c	2
10: a	1	9: c	2
11: c	1	9: c	2
11: c	1		
12: c	1		

表 11: 4 行目の解析終了時のメンバ変数表

エイリアス ID	変数	文番号	エイリアス ID
1	n	3	-
2	n	4	-

表 12: 5 行目の解析終了時の変数表

変数	文番号　エイリアス ID	
a	3	1
b	4	2
c	5	2

R4–R8: b は参照変数であるが，AT(b) のエイリアス ID と VT(b) のエイリアス ID が”2” で一致，$MT(2)$ が存在する，演算子”.” の左辺でない，とどの条件にも一致しないので実行されない．

2. 定義されている変数 c の処理．

D1: 入力 v は c で，i は変数が直接参照されているので入力されない．

D2: 入力 i がないので実行されない．

D3: 変数表 VT(c) の文番号を 5 に変更．

D4: 変数 c は参照変数で AT(c) と $VT(c)$ のそれぞれのエイリアス ID が”2” と” なし” で一致しないので実行される．VT(c) のエイリアス ID を 2 に変更．

D5: メンバ変数表 $MT(2)$ は存在するので実行されない．

5 行目の解析を行うことによって変数表のみ 4 行目終了時点から変更されている．5 行目の解析終了時の変数表を表 12 に示す．

7 行目

- **Phase1**

1. 変数 a, n が参照，変数 n が定義されているということを抽出．

- **Phase2**

1. 参照されている変数 a の処理．

R1: 入力 v は a で，i は直接参照されているので入力されない．

R2: 入力 i がないので実行されない．

R3: 3 行目から 7 行目へ変数 a に関するデータ依存関係があることを抽出．

R4–R5: a は参照変数であるが AT(a) のエイリアス ID と VT(a) のエイリアス ID が”1” で一致，$MT(1)$ が存在するので実行されない．

R6: a は演算子"." の左辺であり，その右辺は参照されている変数 n であるので実行される．v' は変数 n，i は VT(a) のエイリアス ID の値 1.

R7–R8: R6 の条件に一致したので実行されない.

2. 上記 1. の入力で参照されている変数 n の処理.

R1: 入力 v は n で，i は 1.

R2: i が入力されているので実行され，メンバ変数表 $MT(1)$ の変数表 VT(n) を得る.

R3: 3 行目から 7 行目へ変数 n に関するデータ依存関係があることを抽出.

R4–R8: n は参照変数でないので実行されない.

3. 参照されている変数 a の処理.

R1: 入力 v は a で，i は直接参照されているので入力されない.

R2: 入力 i がないので実行されない.

R3: 3 行目から 7 行目へ変数 a に関するデータ依存関係があることを抽出.

R4–R5: a は参照変数であるが AT(a) のエイリアス ID と VT(a) のエイリアス ID が"1" で一致，$MT(1)$ が存在するので実行されない.

R7: a は演算子"." の左辺であり，その右辺は定義されている変数 n であるので実行される．v' は変数 n，i は VT(a) のエイリアス ID の値 1.

R6, R8: R7 の条件に一致したので実行されない.

4. 上記 3. の入力で定義されている変数 n の処理.

D1: 入力 v は n で，i は 1.

D2: i が入力されているので実行され，メンバ変数表 $MT(1)$ の変数表 VT(n) を得る.

D3: 変数表 VT(n) の文番号を 7 に変更.

D4–D5: n は参照変数でないので実行されない.

7 行目の解析を行うことによってメンバ変数表が変更されている．7 行目の解析終了時のメンバ変数表を表 13 に示す.

このようにして 12 行目までの解析を行うことによって，スライス計算手順の Phase1 及び Phase2 が終了し図 18 のサンプルプログラムに存在するデータ依存関係の抽出が完了する．表 14 に抽出したデータ依存関係の表を示す.

図 17 に Phase3 を実行することによって構築されるプログラム依存グラフを示す.

表 13: 7 行目の解析終了時のメンバ変数表

エイリアス ID	変数	文番号　エイリアス ID
1	n	7　-
2	n	4　-

表 14: 図 12 のデータ依存関係

DD(3, a, 7), DD(3, n, 7), DD(3, a, 10), DD(4, b, 5)
DD(4, n, 8), DD(4, n, 9), DD(5, c, 8), DD(5, c, 9)
DD(7, n, 11), DD(10, c, 11), DD(10, c, 12), DD(11, n, 12)

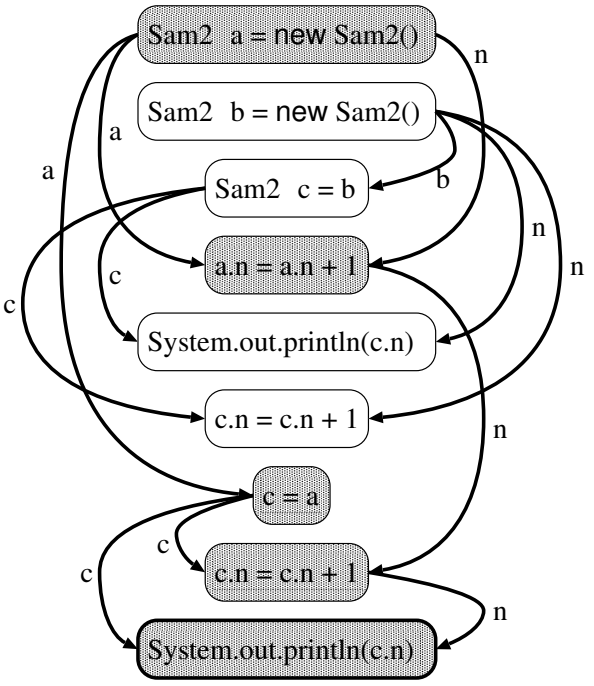
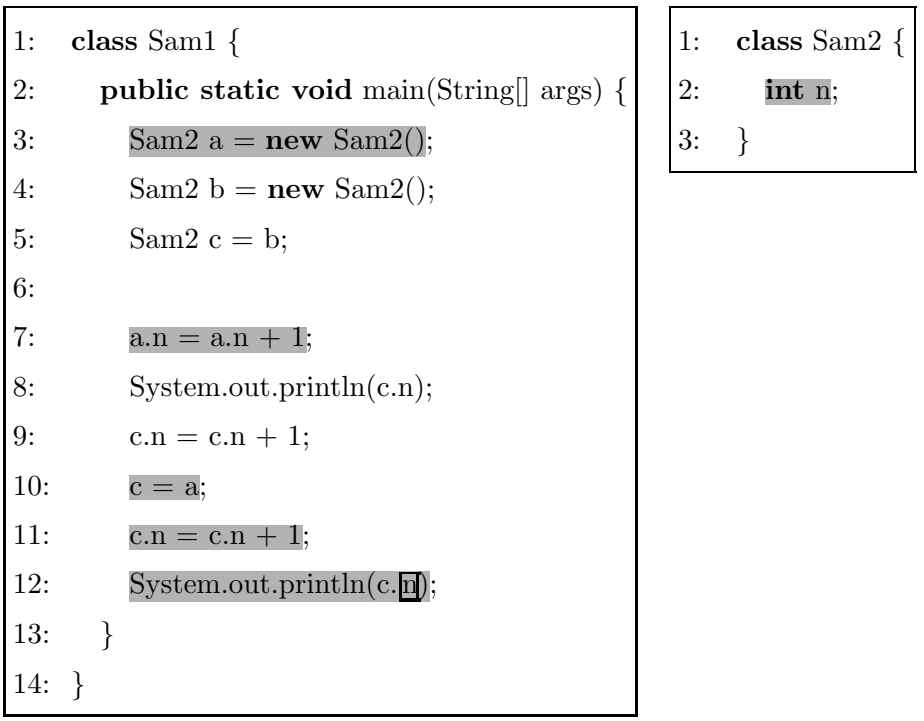


図 17: 図 12 のプログラム依存グラフ

図 18 にスライス基準 <12, n>（太枠部）に対するスライス（網掛部）を示す．また，図 17 のプログラム依存グラフの網掛部が対応する節点である．



```
1:  class Sam1 {
2:    public static void main(String[] args) {
3:      Sam2 a = new Sam2();
4:      Sam2 b = new Sam2();
5:      Sam2 c = b;
6:
7:      a.n = a.n + 1;
8:      System.out.println(c.n);
9:      c.n = c.n + 1;
10:     c = a;
11:     c.n = c.n + 1;
12:     System.out.println(c.n);
13:   }
14: }
```

```
1:  class Sam2 {
2:    int n;
3:  }
```

図 18: スライス基準 < 12, n > に関するスライス

5　スライシングツールの実現

4 で提案したエイリアス関係を考慮したプログラムスライスの計算手法を，スライシングツールとして実装を行った．実装先は，我々の研究グループで開発している Java プログラム解析フレームワークであり，ライブラリとしての追加実装である．

以降，Java プログラム解析フレームワークについて，そしてスライシングツールについて述べる．

5.1　Java プログラム解析フレームワーク

Java プログラム解析フレームワークとは，我々の研究グループで開発を行っている静的に Java プログラムの解析を行うためのライブラリ，ツール群である．このフレームワークは次に挙げる 4 つの部から構成されている．

- **解析木構築部**
  Java プログラムのソースコードを読み込み，意味解析木を構築する部である．この部は次の 2 つのライブラリから成っている．
  - **構文解析ライブラリ**
    Java プログラムのソースコードを読み込み，**字句解析**（*Lexical Analysis*）及び**構文解析**（*Syntax Analysis*）を行い，抽象構文木を構築する．
  - **意味解析ライブラリ**
    抽象構文木を読み込み，**意味解析**（*Semantic Analysis*）（識別子表を作成し，識別子に関する宣言と参照間の関係を抽出する）を行い，意味解析木を構築する．
- **XML データベース部**[20]
  Java プログラムのソースコードを解析することで得られた意味解析木が持つ，構文木情報及び意味情報を **XML**（*eXtensible Markup Language*）[4] を用いてデータベース化した部である．この部は次の 1 つのライブラリと複数のツールから成っている．
  - **XML-意味解析木 変換ライブラリ**
    XML 文書と意味解析木の相互変換を行う．事前にソースコードを解析し XML 文書化していれば，ソースコードからではなく XML 文書からの解析が可能となる．
  - **XML-Java 変換ツール**
    XML 文書から Java プログラムのソースコードへの復元を行うツールである．

- **XML-HTML 変換ツール**
  XML 文書からソースコードの HTML 表記への変換を行う．識別子の宣言と参照はリンクで表現されているのが特徴である．
- **XML-XML 変換ツール**
  XML 文書中のソースコード情報を直接編集するツールである．特徴として異なるスコープ上の同名の識別子を区別して変換することが可能である．

- **エイリアス解析部**[19]
  意味解析木を読み込み，エイリアス解析を行う部である．その解析はプログラムの実行順を考慮した FS エイリアス解析である．また，ユーザの要求に応じたエイリアス集合の計算を行う．この部には 1 つのライブラリから成っている．
  - **エイリアス解析ライブラリ**
    意味解析木を読み込み，**エイリアスフローグラフ**（*Alias Flow Graph, AFG*）及び**メソッド呼び出しグラフ**（*Method Flow Graph, MFG*）を構築する．エイリアスフローグラフとは，単一メソッド内のエイリアス関係を無向グラフで表現したものである．メソッド呼び出しグラフとは，クラス中に存在するメソッド間の呼び出し関係を有向グラフで表現したものである．

- **ユーザインターフェース部**
  ユーザインターフェースはテキストの表示及び編集の機能を持っており，これにより Java プログラムの編集が可能になっている．また，エイリアス集合の計算結果はテキストウィンドウに表示するだけでなく，ツリー表記をするエイリアスツリーウィンドウがある．

各種ライブラリの実装は C++で，XML 関連ツールの実装は **XSLT**（*XSL Transformations*）[18] で行っている．そしてユーザインタフェース部の実装は C++で行っており，gtkmm[8] 及び GTK+[7] ツールキットを使用している．

次に述べるスライシングツールを含めた Java プログラム解析フレームワークを図 19 に示す．

5.2 スライシングツール

スライシングツールの実装は，スライス計算部を Java プログラム解析フレームワークにライブラリとして追加実装することによって行った．

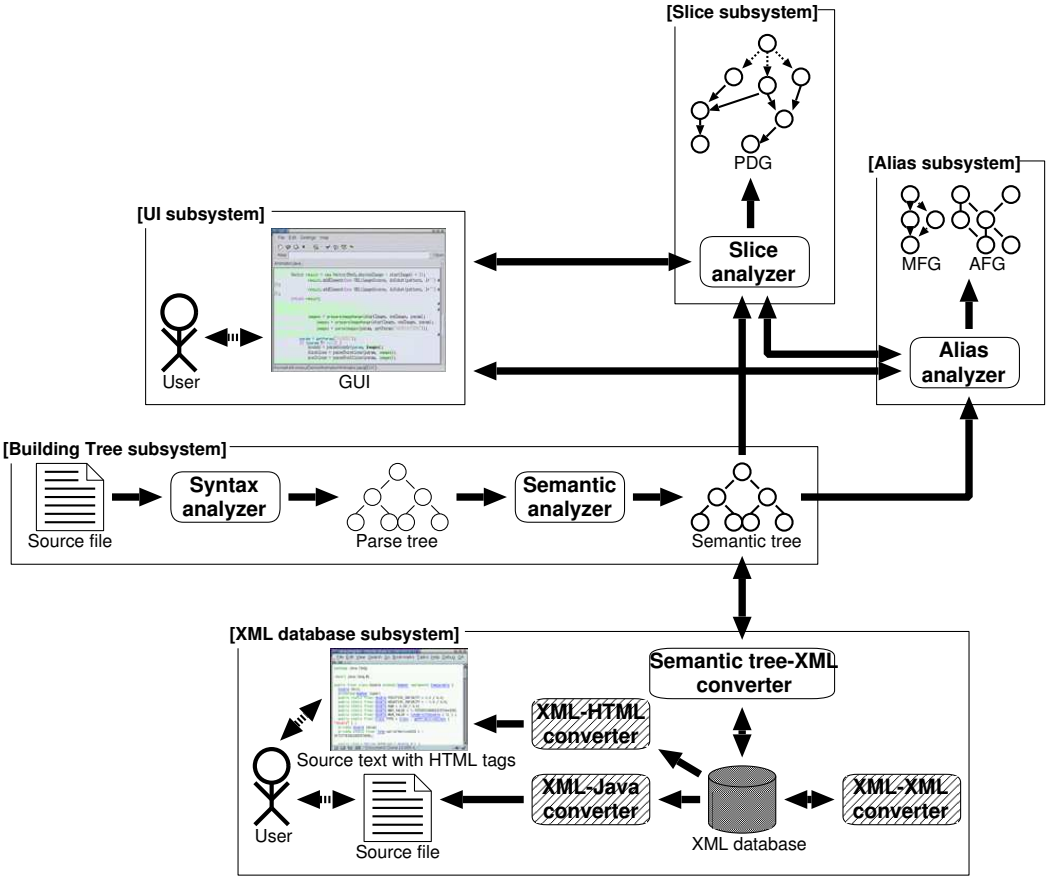


図 19: Java プログラム解析フレームワーク

スライス計算ライブラリ

スライス計算ライブラリは意味解析木を読み込み，エイリアス解析部からエイリアス集合の情報を取り出しながらプログラム依存グラフを構築する．また，ユーザ要求として与えられたスライス基準に対するスライスの計算を行い，その結果をユーザインターフェース部に渡し表示する．

ライブラリの実装は C++で行っており行数は約 4500 行である．

ツールの特徴

実装したスライシングツールの 1 つ目の特徴として，メソッド呼び出しをスライス基準として指定することが可能である．この指定の場合は，メソッド呼び出しの引数及びメソッド呼び出しが参照している呼び出し元のメンバ変数がスライス基準になっていると判断して行っている．

また 2 つ目の特徴として，抽出されるスライスの粒度を選択することが可能ということである．粒度の選択は以下の 2 種類から行える．

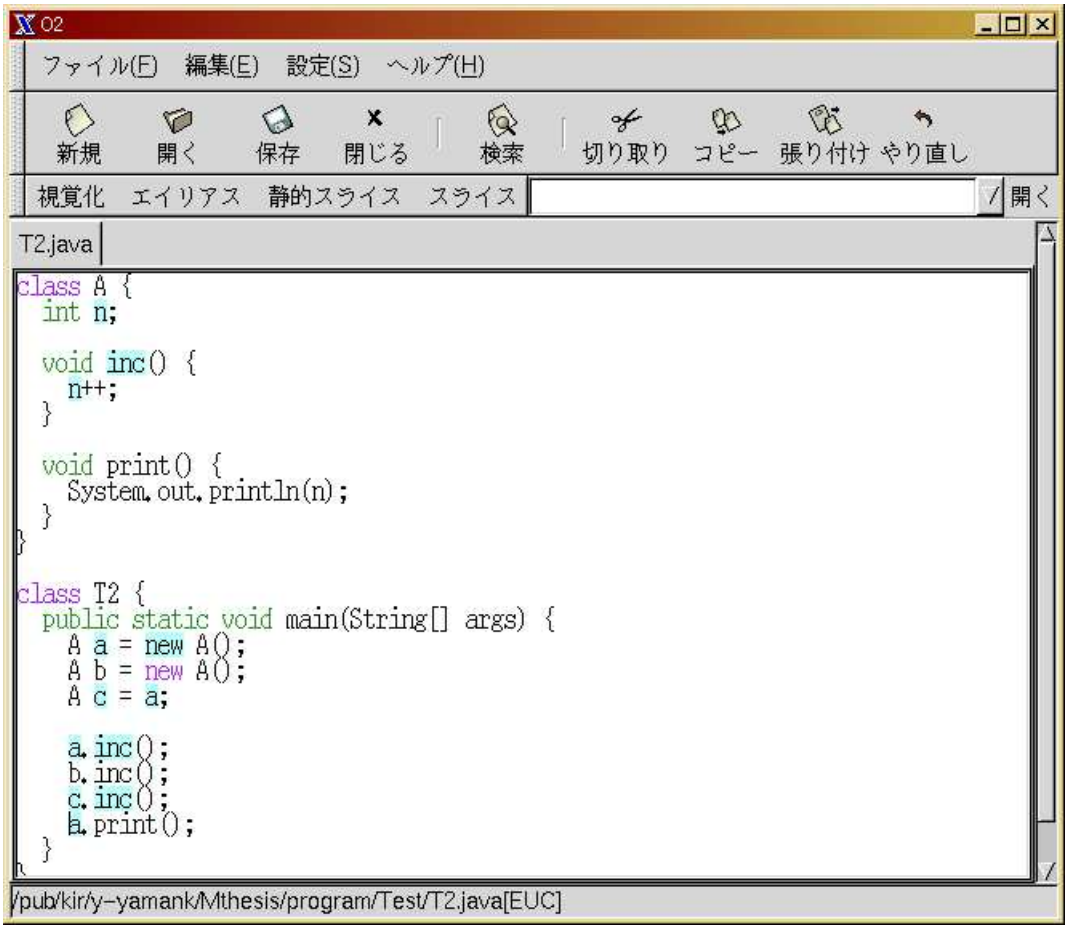


図 20: ツール上でのスライス（詳細）表示の例

- **詳細**
  スライス基準が参照変数である場合，メンバ変数を含めた参照変数全体のスライスを抽出

- **簡略**
  スライス基準が参照変数である場合，メンバ変数を含めないスライス基準の参照変数そのものだけのスライスを抽出

ツール上における詳細，簡略それぞれのスライス表示例を図 20，図 21 に示す．

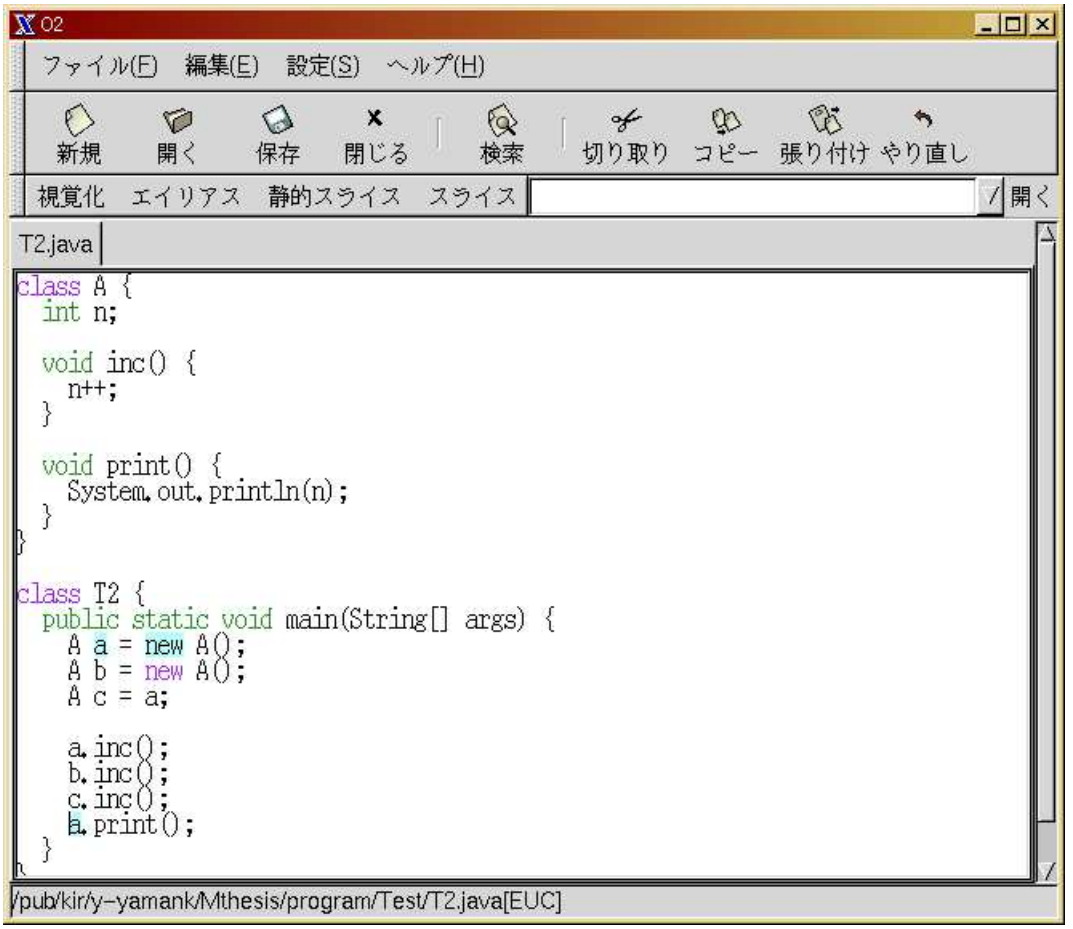


図 21: ツール上でのスライス（簡略）表示の例

6　考察

6.1　評価

6.1.1　スライスの定義

プログラムスライス [12] の定義は，

- 解析対象のプログラムの部分集合である
- 実行可能かつ，同じ入力を与えた場合，同じ状況になって終了しなければならない

というものである．ここでいう同じ状況というのは出力が同じというだけではなく，プログラムの異常終了も同じということを再現しなければならないということである．このプログラムスライスの定義の観点から提案手法を評価する．

図 18 のサンプルプログラムとのスライスの場合，スライス基準 < 12, n > における出力結果は同じになるであろうといえる．だが，このサンプルプログラムおけるスライスが実行可能であるかといえばその限りではない．なぜならば，元のプログラムにある変数 c の宣言が 5 行目にあるがスライスにはそれが含まれていないからである．この点に関しては，提案手法でスライスを抽出した後，手で宣言を加えれば実行可能なプログラムになるので，この条件はクリアすることができる．

しかし，複雑な制御構造が絡んできた場合，スライスの定義を満たすことは困難なものになる．実際，過去に提案されている数々のスライス手法も複雑な制御構造に対する解析について，スライスの定義を完全に満たすのではなく，スライス基準に影響を与え得る文の集合という点を満たしていればよいとしている．これについては我々も同様に考えている．それゆえに提案手法のスレッドや例外だけでなくメソッドの再帰呼び出しなどの制御構造への対応が今後の課題として挙げられる．

スライスの定義による評価をまとめると，提案手法はプログラムスライスの定義のうち，実行可能で同じ入力を与えた場合，元のプログラムと同じ状況で終了する，ということは満たしていないがスライス基準として与えた変数に影響を与え得る文の集合，ということは満たしている．

6.1.2　解析コスト

実装したスライシングツールを用いて解析時間，解析時使用メモリ量の観点から評価を行った．実験環境は表 15 に示す．

解析対象となるプログラムは，対象プログラムが属しているパッケージ中で参照されているもの及び参照されている JDK クラスライブラリである．ただし，PDG 構築における解析

表 15: 実験環境

CPU	Penium4 1.5Ghz
メモリ	512MB
OS	FreeBSD 5.0-CURRENT

表 16: 実験プログラムの概要

プログラム	ファイル数	クラス数	行数
P1	1	2	24
P2	3	3	323
P3	19	20	4302

時間と使用メモリ量は JDK クラスライブラリを除いた分である．

実験に使用したプログラムの概要を表 16 に示す．ここでのファイル数，クラス数，行数は参照している JDK クラスライブラリを除いている．P1 は本研究の実験用に作成したプログラムで図 20 で表示されているもの，P2 は簡易ドローツール，P3 は Apache が提供している XML パーサー，Xerces[17] である．

それぞれのプログラムに対する解析時間，メモリ使用量の結果を表 17，表 18 に示す．

表 17: 解析時間（単位：ms）

プログラム	意味解析木構築	エイリアス解析	PDG 構築	総解析時間
P1	12011.4	5175.8	18.8	17206
P2	45611.4	21580.8	598.6	67790.8
P3	12818.8	5302.8	1539.2	19660.8

表 18: 解析時使用メモリ量（単位：MB）

プログラム	意味解析木	エイリアス	PDG	合計
P1	132	16	1	149
P2	395	61	1	457
P3	135	14	7	156

P2 の総解析時間及びメモリ使用量の合計が P1, P3 と比べて非常に大きくなっているのは，P2 は GUI を搭載しているプログラム（P1, P3 は CUI）だからである．そのため JDK を含むソースコードの量が大きくなるため，この結果になったといえる．JDK に対する PDG を構築すれば解析コストがさらに変わってくることが予想されるが，現時点での必要な解析時

間は現実的なものだと考えられる．

6.1.3 スライスサイズ

P1 と P2 に対してエイリアス関係を考慮した場合としなかった場合とで，同じスライス基準に対してスライス計算を行った．計算結果として得られたスライスサイズの違いを表 19 に示す．

表 19: スライスサイズの比較（単位：行）

プログラム	エイリアス考慮	エイリアス無視
P1	8	6
P2	42	30

エイリアス関係を考慮してスライス計算を行った場合，エイリアス関係を無視する場合に比べてスライスサイズが P1, P2 共に大きくなっている．このことから，エイリアス関係を利用することでプログラム中に多数潜んでいる依存関係の抽出が可能になるといえる．

6.2 関連研究

Atkinson ら [5] はユーザの疑問点をオンデマンドに素早く解決するためのプログラム解析フレームワークを設計・実装しており，そのうちにスライシングツールも含まれているが，エイリアス関係をどのように取り扱っているかが詳しく言及されていない．

Liang ら [6] はオブジェクト指向言語を対象とする**システム依存グラフ**（*System Dependence Graph, SDG*）を利用し，特定オブジェクトに関するスライスを計算する手法を提案している．しかし，この手法はエイリアス解析を前提としているが，その解析方法については触れられていない．

Bergeron ら [10] は，Java のバイトコードを対象とするエイリアスを含めたデータ依存関係のグラフ (*A-DDG*) を定義し，A-DDG に基づくスライシングアルゴリズムを提案している．このアルゴリズムはバイトコード中に存在し得るエイリアス関係を考慮しているが，実装にまでは至ってない．

このように，プログラム解析やプログラムスライスに対して，エイリアス解析の必要性を論じ，手法として提案されているが実装を明確にしているものがない．その点から本研究は有効なものと考えられる．

7　むすび

本研究では，オブジェクト指向言語を対象とするエイリアス関係を考慮した静的プログラムスライス計算手法を提案した．提案した手法では，引数の参照渡し，参照変数，ポインタを介した間接参照などで生じるエイリアス関係を考慮することで，従来の依存関係解析手法では発見することのできない依存関係を抽出できるようになり，スライス計算の正確性が高まった．

また，提案手法をスライシングツールとして，我々が研究開発を行っている Java プログラム解析フレームワークに追加実装し，その有効性を妥当性し，解析コストの面から検証した．ツールの特徴としては，メソッド呼び出しをスライス基準として指定することが可能であること，そして抽出するスライスの粒度の選択が可能であることが挙げられる．

今後の課題として，

- スレッド，例外を含めた複雑な制御構造の解析への対応
- エイリアス関係，各種依存関係のデータベース化
- 大規模システムに対する適用実験

などが挙げられる．

謝辞

本研究の全過程において，常に適切な御指導および御助言を頂きました 大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻 井上 克郎 教授に心より深く感謝いたします．

本論文の作成において，逐次適切な御指導および御助言を頂きました 大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻 楠本 真二 助教授に心から感謝いたします．

本論文の作成において，適切な御指導および御助言を頂きました 大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻 松下 誠 助手に深く感謝いたします．

本研究を通して，適切な御助言を頂きました 大阪大学 大学院基礎工学研究科 情報数理系専攻（現 株式会社 東芝） 大畑 文明氏に深く感謝いたします．

本研究を通して，適切な御助言を頂きました 大阪大学 大学院基礎工学研究科 情報数理系専攻 横森 励士 氏に深く感謝いたします．

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 大学院情報科学研究科 コンピュータサイエンス専攻 井上研究室の皆様に深く感謝いたします．

参考文献

[1] A.V.Aho, S.Sethi and J.D.Ullman: “Compilers : Principles, Techniques, and Tools” (1986).

[2] B.Steensgaard: “Points-to analysis in almost linear time”, *In Proceedings of the 23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pp.32–41, Beach, Florida (1996).

[3] B.Stroustrup: “The C++ Programming Language (Third edition)” (1997).

[4] “Extensible Markup Language (XML) 1.1”, **http://www.w3.org/TR/xml11/**.

[5] D.C.Atkinson and W.G.Griswold: “The Design of Whole-Program Analysis Tools”, *Proceedings of 18th International Conference on Software Engineering*, pp.16–27, Berlin, Germany, March 25–20 (1996).

[6] D.Liang and M.J.Harrold: “Slicing Objects Using System Dependence Graphs”, *Proceedings of the International Conference on Software Maintenance*, pp.358–367, Washington, D.C. (1998).

[7] “GTK+ - The GIMP Toolkit”, **http://www.gtk.org/**.

[8] “gtkmm - the C++ interface to GTK+”, **http://gtkmm.sourceforge.net/**.

[9] H.Agrawal and J.Horgan: “Dynamic Program Slicing”, *SIGPLAN Notices*, vol.25, no.6, pp.246–256 (1990).

[10] J.Bergeron, M.Debbabi, M.Debbabi, M.M.Erhioui and B.Ktari: “Static Analysis of Binary Code to Isolate Malicious Behaviors”, *In Proceedings of the IEEE 8th International Workshops on Enterprise Security*, pp.184–189, Stanford University, California, USA, June 16–18 (1999).

[11] J.Gosling, B.Joy and G.Steele: “The JAVA™ Language Specification” (1996).

[12] M.Weiser: “Program Slicing”, *Proceedings of the 5th International Conference on Software Engineering*, pp.439–449, San Diego, California (1981).

[13] P.Tonella, G.Antoniol, R.Fuitem and E.Merlo: “Flow Insensitive C++ Pointers and Polymorphism Analysis and its Application to Slicing”, *Proceedings of the 19th International Conference on Software Engineering*, pp.433–443, Boston, Massachusetts (1997).

[14] R.P.Wilson and M.S.Lam: “Efficient context-sensitive pointer analysis for C programs”, *Proceedings of SIGPLAN'95 Conference on Programming Language Design and Implementation*, pp.1–12, La Jolla, California (1995).

[15] R.Ueda, K.Inoue and H.Iida: “A Practical Slice Algorithm for Recursive Programs”, *Proceedings of the International Symposium on Software Engineering for the Next Generation*, pp.96–106, Nagoya, Japan, February (1996).

[16] S.Horwitz and T.Reps: “The use of program dependence graphs in software engineering”, *Proceedings of the 14th International Conference on Software Engineering*, pp.392-411, Melbourne, Australia (1992).

[17] “xml.apache.org”, **http://xml.apache.org/**.

[18] “XSL Transformations (XSLT)”, **http://www.w3.org/TR/xslt/**.

[19] 大畑 文明, 近藤 和弘, 井上 克郎: “エイリアスフローグラフを用いたオブジェクト指向プログラムのエイリアス解析手法”, **電子情報通信学会論文誌**, vol.J84-D-I, no.5, pp.1–11 (2001).

[20] 山中 祐介, 大畑 文明, 井上 克郎: “プログラム解析情報の XML データベース化 –提案と実現–”, **コンピュータソフトウェア**, vol.19, no.1, pp.39–43 (2002).