オブジェクト指向プログラムにおける エイリアス解析・視覚化ツールの試作

近藤和弘　大畑文明　井上克郎

大阪大学大学院基礎工学研究科
〒 560-8531 大阪府豊中市待兼山町 1-3
Tel: 06-6850-6571
Email: kondou@ics.es.osaka-u.ac.jp

エイリアスとは，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，識別子が異なるが同じメモリ領域を指す可能性のある変数および式の集合である．我々は，解析結果そのものの再利用性，モジュール性に着目した，オブジェクト指向プログラムに対するエイリアス解析手法を提案し，Java エイリアス解析ライブラリとしてその手法を実現している．本研究では，このJava エイリアス解析ライブラリにユーザインタフェースを加えた，Java エイリアス解析ツールの試作を行った．ユーザインタフェース部の実現においては解析結果自身の視覚化にも重点を置いたものとなっている．
キーワード: エイリアス，視覚化，Java

Alias analysis and visualization tool for object-oriented programs

Kazuhiro Kondou, Fumiaki Ohata and Katsuro Inoue

Graduate School of Engineering Science, Osaka University
1-3 Machikaneyama, Toyonaka,
Osaka 560-8531, Japan
Tel: 06-6850-6571
Email: kondou@ics.es.osaka-u.ac.jp

Alias is a set of variables and expressions which possibly refer to the same location during execution. We had proposed an alias analysis method for object-oriented programs, which takes reusability and modularity of its results into account, and we had implemented this method as Javaalias analysis libraries. In this paper, we implement a prototype tool for analyzing aliases for Javaprograms as an user-interface to those libraries, which focuses on visualizing analysis results, too. We apply this tool to program debugging, and discuss its effectiveness.
Keyword: Alias, visualization, Java

1 まえがき

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，識別子が異なるが同じメモリ領域（オブジェクト）を指す可能性のある変数および式の集合（以降，単に**エイリアス集合**（*Alias Set*）と略す）であり，ここでは，同一識別子で同じメモリ領域を指す変数・式もエイリアス集合に含める．プログラムを解析しエイリアス集合を導き出す**エイリアス解析**（*Alias Analysis*）は，ポインタ変数，参照変数を持つ言語が広く利用されている現在，それらに対する依存関係解析（**プログラムスライス**（*Prgram Slice*）[13]），コンパイラ最適化では欠くことのできないものである．

また，現在のプログラム開発環境において，C などの手続き型言語だけでなく，JAVA[6]，C++[2] 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．その理由としては，オブジェクト指向モデルが持つ，拡張容易性，抽象化，カプセル化，モジュール性，再利用性，階層などが挙げられる．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，継承，動的束縛，ポリモルフィズムなど新しい概念が導入されており，それらに対するエイリアス解析の研究がなされている [15]．我々の研究グループでも，解析結果の再利用性，モジュール性を満たすエイリアス解析手法の提案を行い，JAVA エイリアス解析ライブラリとしてその手法を実現した．[14].

前述のようにエイリアス解析は他の解析のバックエンドとしての利用が多く，解析結果そのものをユーザに提示することは考慮されていない．しかしプログラムデバッグ，プログラム理解においては，エイリアス自身の視覚化も重要であると考えており，特に参照変数によるエイリアスが生成されるJAVA では，その解析結果がユーザのプログラム理解に結び付きやすい．

本研究では，JAVA エイリアス解析ライブラリへのユーザインタフェースとして，JAVA エイリアス解析ツールを試作し，具体的なデバッグ例を用いてその有効性を考察する．

以降，2. ではエイリアス解析について紹介し，3. では研究グループが提案したエイリアス解析手法を簡単に説明する．4. で今回試作したJAVA エイリアス解析ツールの設計と構成について述べ，5. でそのツールの使用例を用いて有効性を検証する．6. でまとめと今後の課題について述べる．

2 オブジェクト指向プログラムにおけるエイリアス解析

2.1 エイリアス解析

エイリアス解析は，大きく**FS エイリアス解析**（*Flow-Sensitive Alias Analysis*（以降，FS 解析と略す），**FI エイリアス解析**（*Flow-Insensitive Alias Analysis*）（以降，FI 解析と略す）の 2 つに分けることができる．

FS エイリアス解析 [10, 16] 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*）を利用する．図 1(a) に変数 c（太枠網掛部）の FS エイリアス（細枠部）を，図 1(b) にその計算に用いた到達エイリアス集合を示す．

FI エイリアス解析 [1, 12] 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（または，Point-to）グラフを利用する．図 1(a) に変数 c（網掛部）の FI エイリアス（網掛部）を，図 1(c) にエイリアスグラフを示す．

FS 解析はプログラム文の実行順を考慮しているため，FI 解析と比較し時間計算量，空間計算量を必要とするが，解析の精度は高い．本研究では FS 解析に着目するが，両者の詳細な比較は [11] でなされている．

2.2 オブジェクト指向プログラムにおけるエイリアス解析

既存のオブジェクト指向プログラムにおけるエイリアス解析手法は，既に提案されている手続き型言語のエイリアス解析手法をオブジェクト指向言語に拡張したものとなっているが，それらに関する問題と，その対処方法について考察する．

解析結果の再利用

到達エイリアス集合 $RA(s)$ は，文 s が含まれるプログラム全体の解析により導出されたものであるため，他の文 t が変更されたとき，$RA(s)$ を再計算しなければならない場合が多く存在する．これは，エイリアス解析結果のモジュール性，独立性が満されていないことに起因する．オブジェクト指向プログラムでは，継承機能など，言語自身が再利用を考慮したものとなっているため，記述されたクラスの利用範囲が特定のプログラムにとどまらない．また，クラス階層の上位に存在するクラスは属性，メソッドが汎化されているため一度定義されると変更されることは少ない．そのため，解析結果のクラスやメソッド単位でのモジュール化は，再計算コストの削減に有効であると考えられる．

同一クラスの異なるインスタンス属性の取り扱い

オブジェクト指向プログラムでは，異なるオブジェクト間での状態（属性）と振舞い（メソッド）は独立であるため，図 2 の例では {x.s, A::s}，{y.s, A::s} はエイリアス組と言えるが，同一クラスのインスタンスがその内部情報を共有する（具体的には，{x.s, y.s, A::s} がエイリアス組となる）ことは精度の低下につながる．このため，インスタ

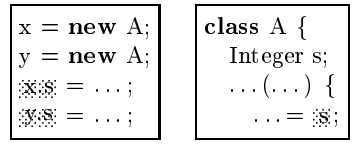


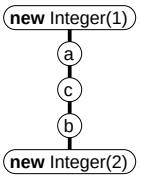
図 2: オブジェクト指向プログラム（JAVA）

```
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);
```

(a) プログラム

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

(b) 到達エイリアス集合



(c) エイリアスグラフ

図 1: FS エイリアス解析 と FI エイリアス解析

ンスごとにクラス内部のエイリアス情報を持たせる手法が考えられるが，単純にインスタンスの数だけエイリアス情報を生成すると多大な空間コストを要する．そこで，存在するエイリアス関係を次の二つに分ける．

- **内部エイリアス関係**（*Inner Alias Relation*）
  外部からの影響を受けないもの（手続き，メソッド，クラス内で閉じたエイリアス関係）
- **外部エイリアス関係**（*Outer Alias Relation*）
  外部からの影響を受けるもの（インスタンス独自のエイリアス関係やインスタンス間をまたぐエイリアス関係）

その上で，内部エイリアス関係のみ前もって解析し，外部エイリアス関係はエイリアス計算時に逐次導出する手法が考えられる．

3 AFG によるエイリアス解析手法

我々は，先に述べた再解析コストの軽減，エイリアス解析結果のモジュール化，内部エイリアス関係の表記のための，エイリアスフローグラフ（AFG）構築法，AFG によるエイリアス計算手法を提案している．[14]

エイリアスフローグラフ（*Alias Flow Graph*，*AFG*）は，FS エイリアス関係をグラフで表現したものであり，FS エイリアスの計算をグラフの到達問題に置き換える．クラス，メソッド単位で到達エイリアス集合によるエイリアス解析を行い，クラス AFG，メソッド AFG をそれぞれ構築する．各 AFG は独立したモジュールとして存在し，対応するクラス，メソッドが更新されない限り不変であり，二次媒体への記憶およびその再利用が可能である．AFG は内部エイリアス関係のみ保持しており，外部エイリアス関係はユーザからの要求があったとき逐次解析する．

以下，各手法について簡潔に述べる．詳細については [14] を参照されたい．

3.1 エイリアスフローグラフ（AFG）構築法

ここでは，AFG の構成要素である AFG 節点，AFG 辺の定義およびその抽出方法について述べる．

AFG 節点

AFG 節点（*AFG Node*）は，文とオブジェクトへの参照の組である．**オブジェクトへの参照**とは，インスタンス生成式，参照変数，ポインタ変数による間接参照式のいずれかを指す．ただし間接参照式は，C や C++などポインタを有する言語で用いる．また，これらの節点を特に**AFG 標準節点**（*AFG Normal Node*）と呼ぶ．

手続きの存在しない単一プログラムにおいては，AFG 標準節点のみ用いることで AFG を構築することができる．しかし，オブジェクト指向プログラムではメソッド，インスタンス属性が存在するため，これらを介したエイリアスの受け渡しを考慮しなければならない．そのため．表 1 に挙げる**AFG 特殊節点**（*AFG Special Node*）を新たに追加する．

AFG 辺

AFG 辺（*AFG Edge*）は，2 つの AFG 節点間の，同一変数参照，代入文，引数の対応による直接のエイリアス関係を表す．エイリアス関係自身は同値関係であるため無向辺でも実現可能ではあるが，エイリアス関係の成立過程を表すために有向辺で表現している．複数の AFG 辺で構成された経路は，2 節点間の（推移的に成り立つ）間接のエイリアス関係を表す．AFG 辺は到達エイリアス集合を利用して抽出される．

インスタンス属性，インスタンスメソッドの表現

オブジェクト指向プログラムでは，a.b.c や d.e() など，インスタンス属性やインスタンスメソッドを表現しなければならないため，インスタンス と その属性（メソッド）を表す AFG 節点間に親子関係を定義している．図 4 にその例（破線の有向辺）を示す．この関係はエイリアス計算時に利用される．

AFG の構築

AFG は，抽出された AFG 節点および AFG 辺を用い，メソッド，クラス単位で構築される．各メソッドを解析し構築される**メソッド AFG**（*Method AFG*）は，メソッドが定義されたクラスの**クラス AFG**（*Class AFG*）に属する．図 4，図 5 はクラス AFG を表しており，メソッド AFG を内包しているのが分かる．

表 1: 特殊節点

特殊節点	概要
Actual-Alias-in（AA-in）	実エイリアス引数（実引数により，メソッドに渡されるエイリアス）
Formal-Alias-in（FA-in）	仮エイリアス引数（仮引数により，メソッドに渡されるエイリアス）
Actual-Alias-out（AA-out）	実エイリアス引数（実引数により，メソッド呼び出し元に渡されるエイリアス）
Formal-Alias-out（FA-out）	仮エイリアス引数（仮引数により，メソッド呼び出し元に渡されるエイリアス）
Method-Alias-out（MA-out）	戻りエイリアス値（戻り値により，メソッド呼び出し元に渡されるエイリアス）
Method	メソッド呼び出し（メソッドの戻り値により，メソッド呼び出し元に渡されるエイリアス，AA-in（out）節点の親節点）
Instance-Alias-in（IA-in）	エイリアス属性（属性により，メソッド内に渡されるエイリアス）
Instance-Alias-out（IA-out）	エイリアス属性（属性により，メソッド外に渡されるエイリアス）

3.2 AFG によるエイリアス計算手法

方針

AFG によるエイリアス計算は，AFG のグラフ到達問題に置き換えられる．その計算は以下の方針に従う．

1. 親節点を持つ節点のエイリアスを導出する場合，まずその親節点のエイリアス計算問題を解決させる（これにより，エイリアス解析対象が特定インスタンスに限定される）
2. MA-out や FA-in（out）節点など，メソッド間をまたぐエイリアス関係の導出の際には，呼び出し先（呼び出し元）メソッドを探索し，対応する Method 節点や AA-in（out）節点から AFG 辺をたどる

オブジェクトコンテキスト

エイリアス集合 $\mathcal{P}$ に関する**オブジェクトコンテキスト**（*Object Context*）とは，$\mathcal{P}$ が子節点として持つ Method 節点から導出される，$\mathcal{P}$ が直接もしくは間接的に呼び出す可能性のあるメソッド集合を指し，Object-Context($\mathcal{P}$) と表す．

直接呼び出されるメソッドとは，Method 節点に対応するメソッドを指す（$\mathcal{P}$ から類推される型が複数クラスになる場合，対応するメソッドも複数になる可能性がある）．間接的に呼び出されるメソッドとは，直接呼び出されるメソッドが内部で呼び出す可能性のある同一インスタンスのメソッドを指す．

エイリアス計算例

エイリアス計算の例として，図 3 の参照変数 c（太枠網掛部）のエイリアスの抽出手順を説明する．

1. 参照変数 c に対応する AFG 節点から AFG 辺をたどり，Method 節点 result() に到達する．
2. Method 節点 result() は親節点 b を持つため，親節点 b のエイリアス計算を行い，Method 節点のエイリアス計算に関与するインスタンスを特定する．
   (a) 節点 b のエイリアス $\mathcal{B}$ を計算する（図 4）．
   (b) $\mathcal{B}$ に存在するインスタンス生成式から節点 b の型を類推する [Calc クラス]．

```
public class Calc {
  Integer i;
  public Calc() {
    i = new Integer(0);
  }
  public void inc() {
    i = new Integer
      (i.intValue() + 1);
  }
  public void add(int c) {
    i = new Integer
      (i.intValue() + c);
  }
  public Integer result() {
    return(i);
  }
}
```

```
class Test {
  Calc a, b;
  Integer c;
  Test() {
    a = new Calc();
    b = new Calc();
    a.inc();
    b.add(1);
    c = b.result();
  }
}
```

図 3: 文 c = b.result() の c に関するエイリアス（網掛部）

   (c) Object-Context($\mathcal{B}$) を計算する [{Calc::Calc(), Calc::add(**int** c), Calc::result()}].
3. 節点 b は Calc クラスのインスタンスへの参照であることから，Calc::result() の MA-out 節点から AFG 辺をたどる（図 5）．
   (a) AFG 辺をたどり，IA-in[i] に到達する．
   (b) 属性 i のエイリアスに影響を与えうるメソッド，すなわち Object-Context(**this**) = Object-Context($\mathcal{B}$) の IA-out[i] 節点から AFG 辺をたどる（Calc::Calc()，Calc::add() も同様）．

図 3 には，c のエイリアス（網掛部）および Object-Context($\mathcal{B}$)（下線部）が示されているが，Object-Context($\mathcal{B}$) に含まれない Calc::inc() はエイリアス計算対象から除外されているのが分かる．

4 Java エイリアス解析ツール

本節では，AFG によるエイリアス解析手法を実現した JAVA エイリアス解析ツールについて説明する．

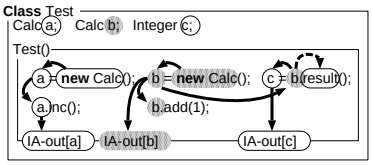


図 4: 図 3 の文 c = b.result() の b に関するエイリアス（網掛部）

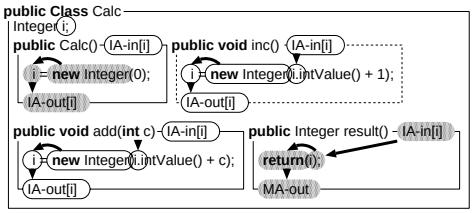


図 5: 図 3 の文 c = b.result() の b.result() に関するエイリアス（網掛部）

4.1 構成

ツールの構成を図 6 に示す．ツールは解析部（JAVA エイリアス解析ライブラリ）とユーザインタフェース部（以降，UI 部と略す）で構成されており，解析部は UI 部からの要求に応じてエイリアス計算を行ない，その結果を表示部に渡す．UI 部はユーザの要求に応じて解析部にエイリアス計算を依頼し，その結果を表示する．

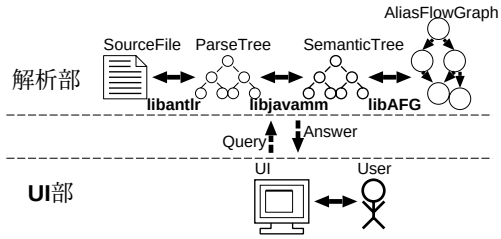


図 6: ツール構成

解析部（Java エイリアス解析ライブラリ）

解析部は C++で記述されており，libANTLR，libjavamm，libAFG の 3 つのライブラリで構成されている．

libANTLR 字句解析，構文解析，構文解析木へのインタフェース[¶1]

libjavamm 意味解析，意味解析木へのインタフェース

libAFG AFG 構築，AFG へのインタフェース，エイリアス計算

ユーザインタフェース（UI）部

UI 部は C++で記述されており，GUI ライブラリとして Gtk−−[5] を使用している．以降，UI 部の設計およびその機能について述べる．

[¶1] ANTLR[4] は，言語 L の文法 $\mathcal{L}$ を与えることで L の字句解析，構文解析ルーチンを C++もしくはJAVA で生成するツールである．

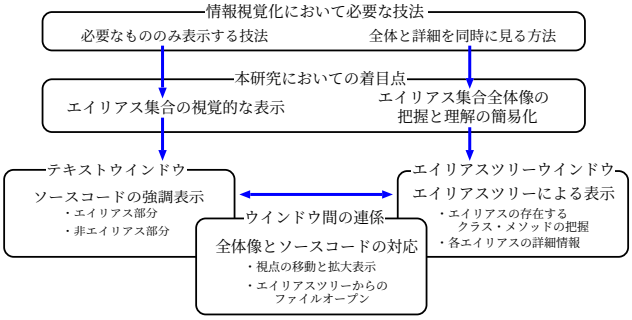


図 7: UI 部の設計方針とその実現

4.2 ユーザインタフェース設計

関連研究

文献 [9] に，情報の視覚化において必要な技法として以下のものが挙げられている．

1. 必要なもののみ表示する技法
2. 全体と詳細を同時に見る方法
3. 抽象的データの画面へのマッピング法
4. 自動レイアウト手法
5. 操作しやすいインタフェース

また，プログラム解析結果の視覚化に関する研究として，SeeSoft[7] をベースに AT&T で開発された SeeSlice[17] がある．SeeSlice はプログラムスライスの視覚化を目的とし，

1. 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 ⇔ 「必要なもののみ表示する技法」
2. プログラムを 3 階層（ファイル，手続き，文）に分類，局所的なソースコードの表示 ⇔ 「全体と詳細を同時に見る方法」

を実現している．SeeSlice は手続き型プログラムのスライスを対象としているが，今回行なうJAVA プログラムのエイリアス表示においても応用可能である．

設計方針

関連研究をふまえ，本研究では次の 2 点を UI 部の設計方針とする．

1. エイリアス集合の効果的な表示 ⇔ 「必要なもののみ表示する技法」
2. エイリアス集合全体像の把握と理解の簡易化 ⇔ 「全体と詳細を同時に見る方法」

図 7 に今回作成した UI 部の設計方針および実現方法を示す．以降，テキストウィンドウ，エイリアスツリーウィンドウ，ウィンドウ間の連係についてそれぞれ述べる．

4.3 テキストウィンドウ

エイリアス集合の導出により，対象プログラムは

- エイリアス集合に含まれる部分 ...**エイリアス部分**（*Alias part*）
- エイリアス集合に含まれない部分 ...**非エイリアス部分**（*Non-alias part*）

の 2 つに区分される．テキストウィンドウでは，ユーザの視点を，エイリアス部分に着目させなければならないが，エイリアス部分 と 非エイリアス部分の差別化の方法はユーザの目的によって様々である．とりわけ，エイリアスの性質上ソースコード中に占める割合は非エイリアス部分が圧倒的に多く，本研究では非エイリアス部分の視覚化に重点を置いた．以下，エイリアス部分，非エイリアス部分の表示についてそれぞれ述べる．

エイリアス部分

エイリアス部分は，**字体の変更**，**背景色の変更**により非エイリアス部分との差別化を計る．また，複数のエイリアス集合を同時に表示する際には，各エイリアス集合ごとに異なる背景色を使用する．

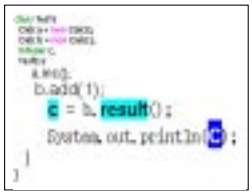
非エイリアス部分

非エイリアス部分は，目的に応じて選択可能な 3 つの表示方法を実現した．具体例は図 8 に示す．

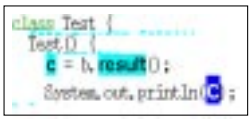
縮小 – 可変: エイリアス部分との距離に応じで字体の大きさを縮小する．これにより，全体の大まかな制御構造を把握しながらエイリアス部分に着目することができる（図 8(a)）．

縮小 – 線分化: エイリアス部分が存在しない行を線に圧縮し表示する．ソースコードの表示領域が削減されるため，エイリアス部分にのみ着目したいとき有用である．また，対象となる文のインデント位置およびその長さは保存されるため，非エイリアス部分の大まかな構造も把握することができる（図 8(b)）．

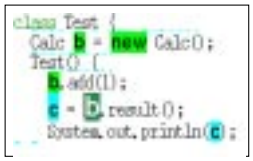
保存: エイリアス計算後もソースコードの表示形態を維持する．ソースコード全体を見ながらエイリアス表示を行う場合に用いる（図 8(c)）．



(a) 縮小 – 可変



(b) 縮小 – 線分化



(c) 保存（左は通常での保存 右は (b) の後での保存）

図 8: 非エイリアス部分の表示方法

4.4 エイリアスツリーウィンドウ

エイリアス部分は複数メソッドにわたって点在することが多く，複数のファイルおよぶことも少なくない．エイリアスツリーウィンドウは，テキストウィンドウでは難しいエイリアス部分全体の把握を支援する．

エイリアスツリーウィンドウは，エイリアスツリーと情報表示リストから構成されている．エイリアスツリーで，ノードを選択するとその情報が情報表示リストに表示される．エイリアスツリーは図 9 に示す構造を持ち，エイリアス指定（エイリアス計算対象となる式），クラス名，メソッド名，エイリアスがそのノードとなる．エイリアスツリーウィンドウの例は図 10 に示す．

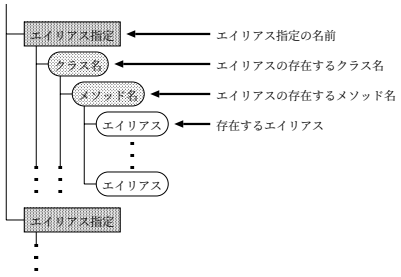


図 9: エイリアスツリー

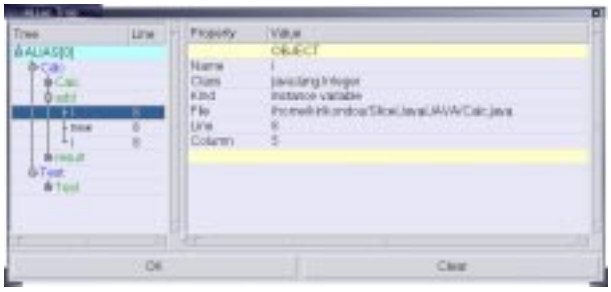


図 10: エイリアスツリーウィンドウ

4.5 ウィンドウ間の連係

テキストウィンドウおよびエイリアスツリーウィンドウは，視点は異なるが同じエイリアス集合を表現したものであり，これらを連係させることで解析結果をより効果的に把握することができる．2 つのウィンドウには表 2 に示すような要素間の対応関係が成り立ち，ツールではカーソル

表 2: ウィンドウ間の連係

	テキスト ウィンドウ	エイリアスツリー ウィンドウ
着目対象	エイリアス	ノード
場所情報（ファイル）	表示ファイル	ファイル名
場所情報（行）	カーソル位置	行番号
エイリアス集合の区別	背景色の違い	サブツリーの違い

移動，背景色変更，字体変更といった形で連係を実現している．

5 ツールの使用例 および 有効性の検証

我々は本ツールの利用目的としてデバッグ，プログラム理解，プログラム保守を考えているが，現時点では実験によるツールおよびエイリアス解析の有効性評価を行っていない．本節ではデバッグフェーズでのフォールト位置特定を例として取り上げその有効性を検証する．

フォールト事例

JDK 付属のサンプルコードに SpreadSheet.java（サイズ: 約 1000 行）という簡単な表計算を行うJAVA アプレットがある．このプログラムを実行した際，図 11 のようなフォールトが発生したとする．表中のセル C1（カーソル

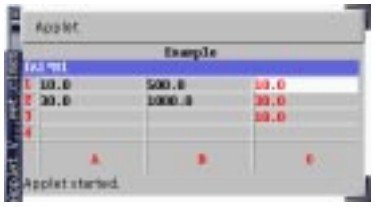


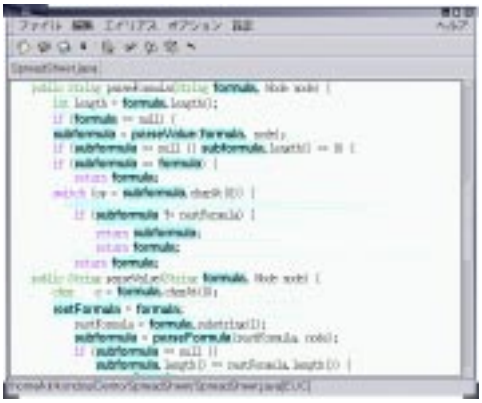
図 11: SpreadSheet アプレットのフォールトを含む実行例

部）は $fA1 * B1$，つまりセル A1(10.0) とセル B1(500.0) の積が表示されなければならない．ところがセル C1 は 10.0 と出力されており，期待される値 5000.0 となっていない．

ツールによるフォールト位置特定

ここでは，実際にツールを利用してフォールト位置を特定するまでの過程を順に説明する．

1. セル内の計算式を解析する Cell::parseFormula() メソッドの第 1 仮引数である，String 型の参照変数 formula のエイリアスを計算する（図 12）．
2. 図 12 から，次のことが分かる．
   - エイリアスは Cell::parseFormula() メソッドと Cell::parseValue() メソッドにのみ存在すること
   - ソースコード中のすべてのエイリアス部分（非エイリアス部分の線分化によりソースコードの表示領域が 10%程度に縮小されたことによる）



(a) テキストウィンドウ

(b) エイリアスツリーウィンドウ

図 12: 参照変数 formula のエイリアス

3. エイリアスツリーウィンドウの情報表示リストを参照しながらエイリアスを含む文のみを調べていくと，Cell::parseValue() メソッドの戻り値が formula となっていることから，この文がフォールトの原因であることが突き止められる（図 13 の情報表示リストからも，この式がメソッド仮引数であることが確認できる）．

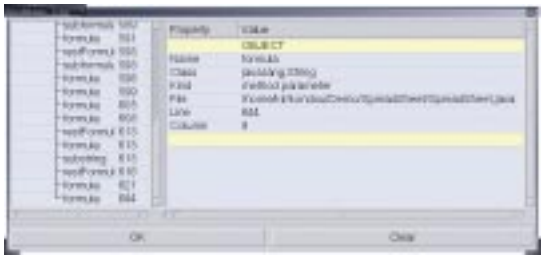


図 13: 式 formula の情報表示リスト

4. フォールト位置付近を調べ，正しくは restFormula を返さなければならないことを類推する（図 14）．
5. 戻り値を restFormula に変更することで，図 15 のように正しい実行結果が得られる．

このようなエイリアス解析結果の視覚化によって，1000 行程度のプログラムであってもデバッグ対象領域をしぼり込んでフォールト位置特定を行うことが可能である．ここではフォールト位置特定を例として用いたが，プログラム理解などにも応用可能であると考えられる．

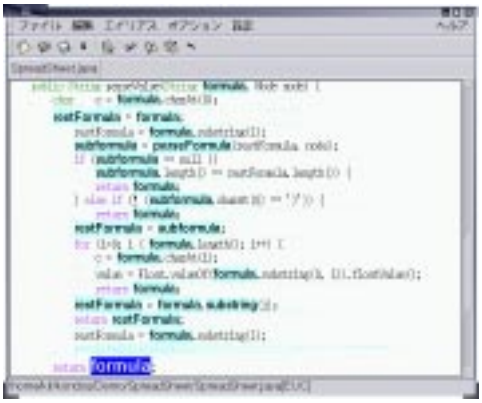


図 14: 式 formula に視点の置かれたソースコード表示

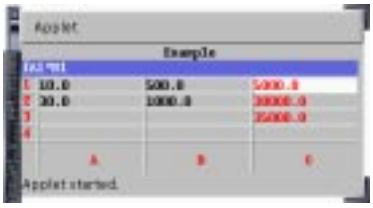


図 15: SpreadSheet アプレットの正常な実行例

6 まとめと今後の課題

本研究では，我々が提案したオブジェクト指向プログラムに対するエイリアス解析手法を実現した，JAVA エイリアス解析ツールの試作を行なった．本ツールは，解析部（JAVA エイリアス解析ライブラリ）とユーザインタフェース（UI）部から構成される．UI 部では，抽出されたエイリアス解析結果を，テキストウィンドウ，エイリアスツリーウィンドウ，ウィンドウ間の連係により，デバッグ，プログラム理解のための視覚化を行う．また，デバッグフェーズでのフォールト位置特定を例に，その有効性を検証した．

今後の課題としては，スライス表示ツールへの発展，ツールの有効性の評価，非エイリアス部分の表示方法の改良などが挙げられる．

参考文献

[1] B. Steensgaard, “Points-to analysis in almost linear time.” in 23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages, pp.32-41, 1996.

[2] B. Stroustrup, “The C++ Programming Language(Third edition),” Addison-Wesley, 1997.

[3] G. Booch, “Object-Oriented Design with Application,” The Benjamin/Cummings Pubrishinh Company, Inc, 1991.

[4] `http://www.ANTLR.org/`, “ANTLR Website.”

[5] `http://gtkmm.sourceforge.net/`, “Gtk−−.”

[6] J. Gosling, B. Joy, and G. Steele, 村上 雅章 [訳], “The Java 言語仕様”

[7] J. L. Steffen, S. G. Eick, and E. E. 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] 片山, 土居, 鳥居 [監訳], “ソフトウェア工学大事典,” 朝倉書店.

[9] 増井, “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ョップ, 1998.

[10] M. Enami, R. Ghiya, and L. J. Hendren, “Context-sensitive interprocedural points-to analysis int the presence of function pointers.” in SIGPLAN’94 Conference on Programming Language Design and Implementation, pp.242–256, 1994.

[11] M. Hind, and A. Pioli, “An empritical comparison of interprocedural pointer alias analysis,” in IBM Research Report #21058, 1997.

[12] M. Shapiro, and S. Horwitz, “Fast and accurate flow-insensitive point-to analysis,” in 24th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages, 1997.

[13] M. Weiser, “Program slicing,” in Proceedings of the 5th International Conference on Software Engineering, pp.439–449, 1981.

[14] 大畑, 井上, “オブジェクト指向プログラムにおけるエイリアス解析について,” 情処学研報, 2000-SE-126, pp.57–64, 2000.

[15] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo, “Flow insensitive C++ pointers and polymorphism analysis and its application to slicing,” in Proceedings of the 19th International Conference on Software Engineeringm, pp.433-443, 1997.

[16] R. P. Wilson, and M. S. Lam, “Efficient context-sensitive pointer analysis for C programs,” in SIGPLAN’95 Conference on Programming Language Design and Implementation, pp.1–12, 1995.

[17] T. Ball, and S. G. Eick, “Visualizing Program Slices,” in Proceedings of the 1994 IEEE Symposium on Visual Languages, pp.288–295, 1994.