オブジェクト指向プログラム変更時の影響波及解析手法の提案

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　Fax: 06-6850-6574
Email: {kondou, oohata, inoue}@ics.es.osaka-u.ac.jp

影響波及解析とは，プログラム変更の影響を受ける部分を識別する手法であり，回帰テストでのテストケース選択に利用される．我々は，プログラム理解，保守といった，より広い範囲での影響波及解析の利用を考えている．しかし，影響の定義はユーザが直面する状況によって様々で，一意に定まるものではなく，既存手法をそのまま利用することはできない．本研究では，Java を対象とする影響波及解析手法を提案し，その実装である Java 影響波及解析ツールを構築する．提案手法では，ユーザが影響を定義可能な枠組を提供することで，ユーザの目的に応じた解析結果を容易に導出することができる．
キーワード: 影響波及解析，オブジェクト指向プログラム，Java

Impact Analysis Method for Changes on Object-Oriented Programs

Kazuhiro Kondo, Fumiaki Ohata and Katsuro Inoue

Graduate School of Engineering Science, Osaka University
1-3 Machikaneyama, Toyonaka,
Osaka 560-8531, Japan
Tel: 06-6850-6571　Fax: 06-6850-6574
Email: {kondou, oohata, inoue}@ics.es.osaka-u.ac.jp

Change impact analysis is a method to identify subprograms that are affected by the changes by the user, which have been used on selecting test cases for regression testing. We consider that change impact analysis would be useful for program understanding and maintenance activities as well as regression testing; however, since the definition of the effect varies according to the user's situation, its definition is not unique. Thus, it is difficult for us to apply the existing methods to such activities. In this paper, we propose a change impact analysis method for Java programs and implement it. Using our method, the user can easily define the semantics of the effect, and we can compute the analysis results based on the defined analysis policy.
Keyword: Change Impact Analysis, Object-Oriented Program, Java

1 まえがき

ソフトウェア保守工程において，プログラム開発者はソフトウェアに対し多くの変更を行うが，その際，誤って欠陥を作り込んでしまう確率は 50%から 80% にも及ぶことが Hetzel により示されている [12]．その要因として，ソフトウェアに変更を加えたときには，変更していない部分に関しても何らかの影響が及ぶ可能性があることが挙げられる．また，近年のソフトウェアの大規模化，複雑化に伴い，ソフトウェア保守に要するコストも増大しており，ソフトウェアの理解性および保守性の向上は保守活動の効率化に大きな効果をもたらす．

変更の影響を受ける部分（**被影響部分**（*Affected Part*）と呼ぶ）を識別するための手法として，**影響波及解析**（*Change Impact Analysis*）が提案されている．影響波及解析の適用分野の代表例として，変更後のソフトウェアが仕様通りに動作するかを確認するための**回帰テスト**（*Regression Testing*）[11] への利用が挙げられる．回帰テストでは，影響波及解析に基づいてテスト対象を限定し，テストケースを必要最小限に抑えることができる．

我々は，プログラム理解，保守といった，より広い範囲での影響波及解析の利用を考えているが，影響の定義はユーザが直面する状況によって様々で，一意に定まるものではなく，既存手法をそのまま利用することはできない．また，近年のソフトウェア開発環境において多く利用されるオブジェクト指向言語には，クラス，継承，動的束縛，ポリモルフィズムなどの独自概念が存在する．そのため，これらを考慮した解析手法が望まれる．

本研究では，Java[6] を対象言語とした影響波及解析手法の提案を行う．提案手法では，被影響部分の探索ルールを選択可能にすることで，ユーザの目的に応じて様々な解析結果が取得できる．また，提案手法に基づいて開発中である Java 影響波及解析ツールについても述べる．

以降，2. では影響波及解析について紹介し，3. では提案する MOG，MAG を利用した影響波及の定義およびその計算手法について説明する．4. で現在開発中である Java 影響波及解析ツールについて述べる．最後に，5. でまとめと今後の課題について述べる．

2 オブジェクト指向プログラムに対する影響波及解析

本節では，影響波及解析について説明し，既存手法の問題点について考察する．なお，本研究ではオブジェクト指向プログラムを対象としている．

2.1 影響波及解析

影響波及解析の目的は，プログラム変更による被影響部分の抽出である．これまでにオブジェクト指向プログラムに対する影響波及解析手法がいくつか提案 [1, 4, 8, 13, 14] されておる．ここでは，既存手法を解析の粒度で大きく 3 つに分類する．

クラス単位: クラスを被影響部分の単位とする [13]．数百クラスにもおよぶような大規模ソフトウェアにおいて特に有効であり，解析も容易である．

メンバ単位: クラスのメンバ（メソッド，フィールド）を被影響部分の単位とする [8, 14]．オブジェクトの構成要素であるメンバが解析結果となり，直感的に理解しやすい．

文単位: 文を被影響部分の単位とする [1, 4]．**プログラムスライス**（*Program Slice*）[9] に基づいており，文中のある変数を**スライス基準**（*Slicing Criterion*）としてユーザが定めることで，スライス基準に影響を及ぼす文（**バックワードスライス**（*Backward Slice*））および，スライス基準が影響を及ぼす文（**フォワードスライス**（*Forward Slice*））が抽出できる．プログラムデバッグ時のフォールト位置特定などに利用される．

クラス単位の解析では，クラス内のどのメンバが実際に影響を受けているかを特定できず，影響の予測としては効果が薄い．文単位の解析では，制御依存関係，データ依存関係，エイリアス関係など，多くの解析が前提となり，解析コストは膨大なものとなる．

本研究では，メソッドの追加，削除，またシグニチャの変更などに対する被影響部分の抽出を目的としており，メンバ単位での解析に着目する．

2.2 影響波及解析の例

オブジェクト指向プログラムでは，従来の手続き型プログラムに比べ，変更箇所以外に影響を及ぼすような変更が数多く考えられる．最悪の場合，プログラム中のあるモジュールの修正が，他のモジュールでバグを発生させてしまうことがある．[3, 10] では，オブジェクト指向プログラムに対する変更は，メソッドの**オーバーライド**（*Override*），フィールドの**隠蔽**（*Hide*）といったオブジェクト指向特有の概念により様々な影響が引き起こされることが述べられている．

図 1 のサンプルプログラムにおいて，クラス `Student` にメソッド `toString()`（網掛部）を追加することを考える．`Student` は `Person` を継承しているため，`Student.toString()` は `Person.toString()` をオーバーライドすることになり，`Student` クラスのオブジェクトに対して `toString()` を呼び出しているメソッド `StudentData.add()`，`Test.test1()` は実行結果が変化する．また，`StudentData.add()` を呼び出しているメソッド `Test.test3()` も，推移的に実行結果が変化する．

影響波及解析の結果として抽出するべきものは，ユーザの目的によって様々であり、実行結果が変化するもの，オーバーライド関係が変化するものなどが考えられる．ここでは，それらのうち 2 つの具体例を挙げる．

```
public class Person {
  private String _name;

  public Person(String name) {
    _name = name;
  }
  public String toString() {
    return _name;
  }
}
```

```
public class Student extends Person {
  private String _id;
  private Collection _friends;

  public Student(String name, String id) {
    _name = name;
    _id = id;
    _friends = new Vector();
  }
  public void addFriend(Person person) {
    _friends.add(person);
  }
  public int getFriendCount() {
    return _friends.size();
  }
  public String toString() {
    return _id;
  }
}
```

```
public class StudentData {
  private Map _data;
  public StudentData() {
    _data = new HashMap();
  }
  public void add(Student student) {
    _data.put(student.toString(),
              new Integer(student.getFriendCount()));
  }
  public void output() {
    //output the list
  }
}
```

```
public class Test {
  public void test1() {
    Student s = new Student("Tim");
    System.out.println(s.toString());
  }
  public void test2() {
    Student s = new Student("Sam", "01");
    s.addFriend(new Person("Tom"));
    System.out.println("Sam: " + s.getFriendCount());
  }
  public void test3() {
    Student s = new Student("Sam", "01");
    StudentData data = new StudentData();
    data.add(s);
    data.output();
  }
}
```

図 1: メソッドの追加による影響（下線部：被影響メソッド）

回帰テストにおけるテストドライバの選択

回帰テストは，プログラムの機能の一部を実行させるためのテスト用モジュール（**テストドライバ**（*Test Driver*）と呼ぶ）の選択に基づいて行われ [5]，影響波及解析による，再実行を要するテストドライバの削減が求められる．Tip らが提案した手法 [2] によると，再度実行が必要なテストドライバの条件は次のようになる．

(a) 変更または削除されたメソッドをテストするドライバ
(b) オーバーライド関係の変化したメソッドへの呼び出し経路をテストするドライバ

クラス `Test` 中の各メソッドは，`Person`，`Student` 用のテストドライバとなっている．今回の変更はメソッドの追加であるため，`Student.toString()` の追加によりオーバーライド関係の変化した `Person.toString()` への呼び出しをテストするドライバを選択する．よって，解析結果として抽出すべきメンバは，`Test.test1()`，`Test.test3()` となる．

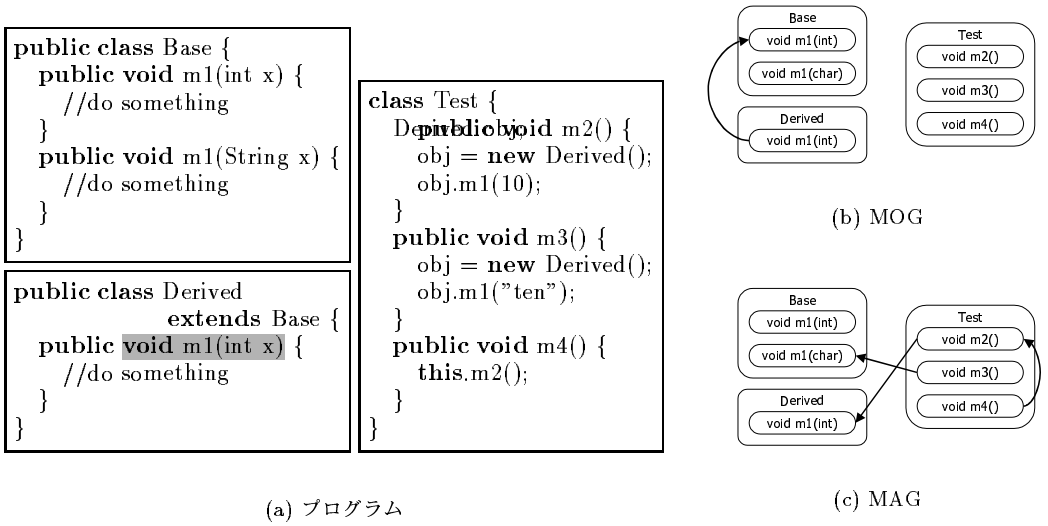
変更に対応するための修正個所の特定

変更による呼び出し先の変化は，メソッドオーバーライドの変化によるものである．変更によって，`Person.toString()` がオーバーライドされ，それにより `Student` クラスのオブジェクトに対する `toString()` による呼び出し先が変化するといった情報は，修正を行おうとするユーザにとって有用である．また，変更に伴う修正を行う場合は，これらのメソッドを直接呼び出してる部分を把握することが必要である．よって，解析結果として抽出すべきメンバは，`Person.toString()`，`StudentData.add(Student)`，`Test.test1()` となる．

2.3 既存手法の問題点

オブジェクト指向プログラムには，従来の手続き型言語にも存在するモジュール間の呼び出し関係に加え，継承関係，また，それによるオーバーライド関係などが存在するため，プログラム変更時には様々な影響が生じ得る．そのため、変更の影響をユーザが把握することは難しく、これらを考慮した影響波及解析手法 [2, 14] が必要となる．

また，既存の手法は回帰テストへの利用を目的としているため，影響の定義もそれを前提としたものになっている．しかし，回帰テストだけではなく，プログラム理解，保守といったより広い範囲での影響波及解析の利用を考えた場合，影響の定義はユーザにより様々であり，一意に決定すべきものではない．そのため，ユーザの目的に応じた影響波及解析が望まれる．



(a) プログラム

(b) MOG

(c) MAG

図 2: メンバオーバーライドグラフ（MOG) とメンバアクセスグラフ（MAG）

3 MOG，MAG による影響波及解析

本節では，クラスのメンバ間の関係を表現する二つのグラフを利用した影響波及解析手法の提案を行う．提案手法により，メソッドのオーバーライド，フィールドの隠蔽を考慮した影響波及解析の実現，およびユーザの様々な目的に対応可能な影響の定義を行うことができる．

3.1 方針

影響は，オーバーライドや呼び出しなどに変化が生じたメンバから発生し，呼び出し経路に従い波及するものである．回帰テストに利用されてきた既存の影響波及解析は，呼び出し経路を逆にたどる，つまり一部の呼び出し経路に限定した特殊な波及を考えていたとみなすことができる [2].

我々は，より一般的な影響波及解析，具体的には，様々な影響の発生検出と波及パターンを組み合わせ適用できる枠組を実現するため，グラフによるアプローチを考えた．グラフを用いてオーバーライド関係および呼び出し関係が表現できれば，影響の発生はグラフの変化で，影響の波及はグラフ探索で，それぞれ置き換えることができる．

本研究では，これらを実現するために，**メンバオーバーライドグラフ**（*Member Override Graph*，*MOG*）および**メンバアクセスグラフ**（*Member Access Graph*，*MAG*）を利用した手法を提案する．

MOG とは，メンバ間の**オーバーライド関係**（*Override Relation*）をグラフで表現したものである．これは，継承により親子関係となるクラスのメンバ間に存在する．具体的には，メソッドオーバーライド，抽象メソッドの**実装**（*Implement*），フィールドの隠蔽といった関係がそれに該当する．

MAG とは，メンバ間の**アクセス関係**（*Access Relation*）をグラフで表現したものである．これは，クラスのメンバ間に存在し，具体的には，メソッドの**呼び出し**（*Call*），フィールドの**参照**（*Use*）といった関係がそれに該当する．

3.2 メンバオーバーライドグラフ（MOG）

ここでは，MOG の構成要素である MOG 節点および MOG 辺の定義，MOG の構築方法について述べる．図 2(b) は，図 2(a) のプログラムから構築される MOG である．

MOG 節点（*MOG Node*）は，各クラスの各メンバに対応した MOG メソッド節点（*MOG Method Node*）と，各クラスの各フィールドに対応した **MOG フィールド節点**（*MOG Field Node*）の二種類から成る．MOG 節点は後述する MAG 節点と一対一で対応する．

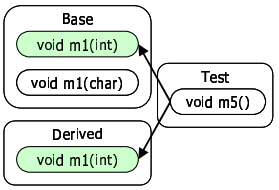
MOG 辺（*MOG Edge*）は，2 つの MOG 節点間のオーバライド関係を有向辺で表現したものであり，メソッドオーバーライドを表す **MOG override 辺**（*MOG override Node*），抽象メソッドの実装を表現する **MOG implement 辺**（*MOG implement Node*），フィールドの隠蔽を表現する **MOG hide 辺**（*MOG hide Node*）の三種類から成る．各辺は，オーバーライドするメンバからオーバーライドされるメンバに引かれる．図 2(b) では，MOG メソッド節点 `void Derived.m1(int)` から，`void Base.m1(int)` に MOG override 辺が引かれている．

MOG は，各クラスのメソッド，フィールド宣言の解析による MOG 節点の抽出，およびクラスの継承関係の解析による MOG 辺の抽出により構築される．

3.3 メンバアクセスグラフ（MAG）

ここでは，MAG の構成要素である MAG 節点および MAG 辺の定義，MAG の構築方法について述べる．図 2(c)

```
class Test {
  boolean flag = false;
  public void m5() {
    Base obj = null;
    if(flag) {
      obj = new Base();
    } else {
      obj = new Derived();
    }
    obj.m1(10);
  }
}
```



(a) プログラム　　(b) MAG

図 3: インスタンスの型が特定できない例

は，図 2(a) のプログラムから構築される MAG である．

MAG 節点（*MAG Node*）は，MOG 節点と同様に，クラス内のメンバに対応する二種類の節点，MAG メソッド節点（*MAG Method Node*），**MAG フィールド節点**（*MAG Field Node*）から成る．MAG 節点は MOG 節点と一対一で対応する．静的初期化子およびコンストラクタはメンバではないが，メンバと同様のアクセス関係を持つことから，これらも MAG 節点として扱う．

MAG 辺（*MAG Edge*）は，2 つの MAG 節点間のアクセス関係を有向辺で表現したものであり，メソッド呼び出しを表す **MOG call 辺**（*MOG call Node*），フィールドの参照を表す **MAG use 辺**（*MAG use Node*）の二種類から成る．

なお，参照変数が指すインスタンスの型が特定できないことにより，アクセスされるメンバが一意に決まらない場合がある．その際には，その参照変数の型から派生したクラスに存在する，同一シグニチャを持つすべてのメンバに対して辺を引く．例えば，図 3 では，`flag` の値が静的に決まらない場合，`obj.m1(10)` によって呼び出されるメソッドは，`void Base.m1(int)`，`void Derived.m1(int)` のどちらであるのかを特定できない．そのため，MAG 節点 `void Test.m5()` から両者に対して MAG call 辺が引かれている．

MAG は，MOG と同様の解析による MAG 節点の抽出，およびメソッド呼び出し式，フィールドアクセス式の解析による MAG 辺の抽出により構築される．

3.4 MOG，MAG による影響波及解析

変更によって何らかの変化が生じた MOG 節点，MAG 節点を検出し，その節点から MOG 辺，MAG 辺をたどることで，被影響部分の抽出を行う．さらに，探索ルールを変更可能とすることにより，ユーザの様々な目的に対応することができる．

3.4.1 被影響部分の分類

提案する影響波及解析により抽出される被影響部分の単位は，プログラム上ではメンバ，MOG，MAG 上では節点となる．本論文では，これらをそれぞれ**被影響メンバ**（*Affected Member*），**被影響節点**（*Affected Node*）と呼ぶ．

ユーザの目的に応じた被影響メンバの抽出を行うためには，グラフ探索の対象となる節点の分類が必要である．具体的には，グラフ節点を，直接的な影響を受けるもの，間接的な影響を受けるものに分け，前者を**直接被影響節点**（*Direct Affected Node*），後者を**間接被影響節点**（*Indirect Affected Node*）と呼ぶ．

直接被影響節点

直接被影響節点は変更が行われた節点に基づいて決定されるもので，表 1 に挙げる 5 種類がある．図 4 は，変更が行われた節点が M8 であるときの直接被影響節点の例を示している．

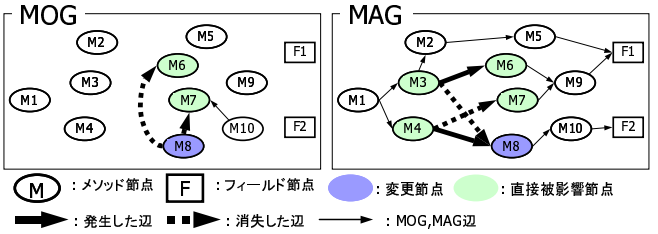


図 4: 直接被影響節点の例（表 1 参照）

間接被影響節点

間接被影響節点とは，前述の直接被影響節点から辺をたどることで到達可能な節点のことで，表 2 に挙げる 2 種類がある．図 5 は，MOG の直接被影響節点が M7，M8，MAG の直接被影響節点が M6，M8 であるときの間接被影響節点の例をそれぞれ示している．

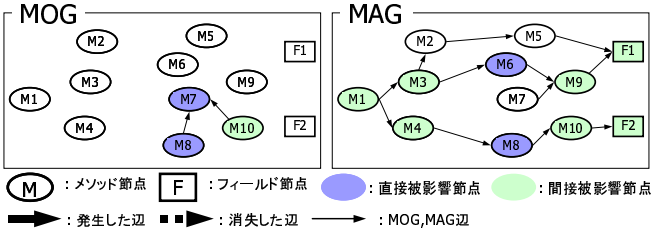


図 5: 間接被影響節点の例（表 2 参照）

3.4.2 被影響部分の抽出

提案手法では，前節で定義した各被影響節点の抽出の有無を組み合わせることにより，ユーザの目的に応じた被影響メンバの抽出が可能となる．組み合わせは数多く存

表 1: 直接被影響節点

直接被影響節点	概要
D-E1: 影響の発生元の節点	プログラム変更に対応する節点（発生，消失した節点もこれに該当する）．（図 4 の MOG 節点 {M8} および MAG の M8}）
D-E2: 辺の発生先の節点	プログラム変更により発生した辺の終節点．(図 4 の MOG 節点 {M7} および MAG 節点 {M6，M8})
D-E3: 辺の発生元の節点	プログラム変更により発生した辺の始節点．(図 4 の MOG 節点 {M8} および MAG 節点 {M3，M4})
D-E4: 辺の消失先の節点	プログラム変更により消失した辺の終節点．(図 4 の MOG 節点 {M6} および MAG 節点 {M7，M8})
D-E5: 辺の消失元の節点	プログラム変更により消失した辺の始節点．(図 4 の MOG 節点 {M8} および MAG 節点 {M3，M4})

表 2: 間接被影響節点

間接被影響節点	概要
I-E1: 順方向の推移的な影響波及のある節点	直接被影響節点から有向辺に従い到達可能な節点．（図 5 の MAG 節点 {M9，M10，F1, F22}）
I-E2: 逆方向の推移的な影響波及のある節点	直接被影響節点から有向辺の逆向きに従い到達可能な節点．（図 5 の MOG 節点 {M10} および MAG 節点 {M1，M3，M4}）

表 3: 探索ルールの例

探索ルール	探索対象となる被影響節点	
	MOG	MAG
R1: アクセス変化メンバ抽出	ϕ	{D-E1, D-E3, D-E5, I-E2}
R2: 関係変化メンバ抽出	{D-E1, D-E2, D-E3, D-E4, D-E5}	{D-E1, D-E2, D-E3, D-E4, D-E5}
R3: 間接アクセスメンバ抽出	ϕ	{D-E1, I-E1, I-E2}

在するが，ここでは代表的な 3 つの探索ルールについて述べる．なお，各ルールに対応する被影響節点の組み合わせの一覧を表 3 に示す．

R1: アクセス発生メンバ抽出

変更により発生した新たな実行経路上に存在する（新たにプログラムの実行結果に影響を及ぼす）部分を抽出する．既存の影響波及解析手法が対象としている，回帰テストでの利用を考慮したものである．2.2 のようなテストドライバ選択に用いる場合は，この被影響節点の中で，ドライバに対応するものだけを選択すればよい．

図 6 に，M8 が変更メンバである場合の抽出例を示す．

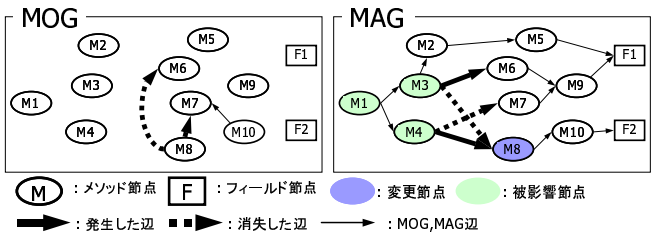


図 6: R1: アクセス発生メンバ抽出

R2: 関係変化メンバ抽出

オーバーライド関係の変化，およびそれに伴うアクセス関係の変化が発生するメンバをすべて抽出する．プログラム変更によるメンバ間の関係の変化を把握し，変更に対応するべき修正個所を識別するために有効である．

図 7 に，M8 が変更メンバである場合の抽出例を示す．

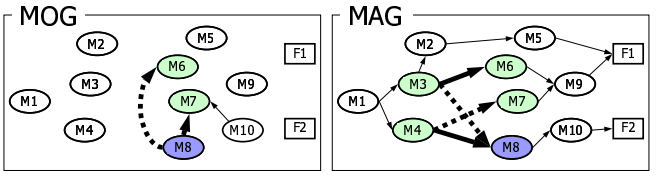


図 7: R2: 関係変化メンバ抽出

R3: 間接アクセスメンバ抽出

変更メンバに直接的および間接的にアクセスする可能性のあるメンバ，変更メンバが直接的および間接的にアクセスする可能性のあるメンバをすべて抽出する．メソッド本体やフィールドの初期値などを変更する場合，アクセス関係に変化は無いが，それらを使用するメンバの実行結果などが変化する可能性があるため，このルールによる被影響メンバの把握が有効となる．

図 8 に，M8 が変更メンバである場合の抽出例を示す．

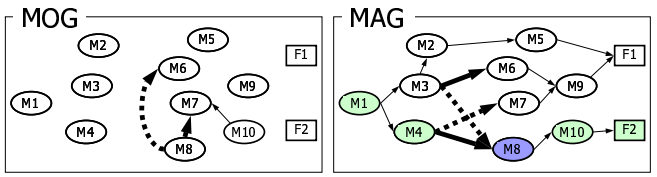


図 8: R3: 間接アクセスメンバ抽出

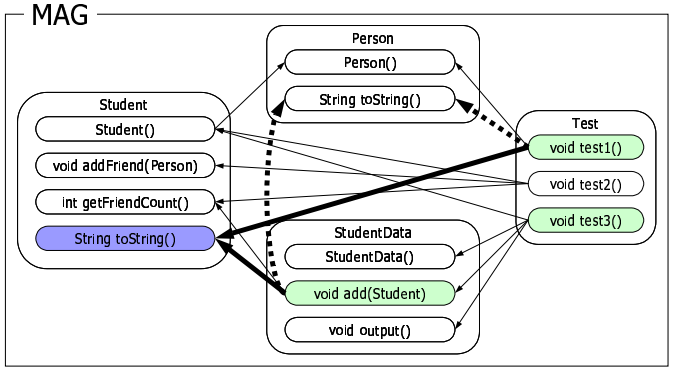


図 9: 「アクセス発生メンバ抽出」適用例

3.5 適用例

ここで，実際に被影響部分の抽出を例を用いて説明する．具体的には，2.2 で挙げた図 1 の `Student` への `toString()` メソッドの追加に対して，前節で定義した「アクセス発生メンバ抽出」の適用を試みる．

(1) 変更前の MOG，MAG を作成する．

(2) 変更後の MOG，MAG を作成する．

(3) 変更前と変更後の MOG，MAG をそれぞれ比較し，探索ルールに基づく被影響節点の抽出を行う．

  (a) 変更前後の MOG には何も適用しない

  (b) 変更前後の MAG に {D-E1, D-E3, D-E5, I-E2} を適用

これにより図 9 の網掛部が抽出メンバとなる．`Test` クラスに着目すると，再実行が必要となるドライバは 2.2 で挙げたものと同じ `Test.test1()`, `Test.test3()` となっており，正しく抽出されていることがわかる．

4 Java 影響波及解析ツール

本節では，提案手法の実装である JAVA 影響波及解析ツールについて説明する．

4.1 ツール構成

ツール構成を図 10 に示す．ツールは解析部（JAVA 影響波及解析ライブラリ）とユーザインタフェース部（以降，UI 部と略す）で構成されている．

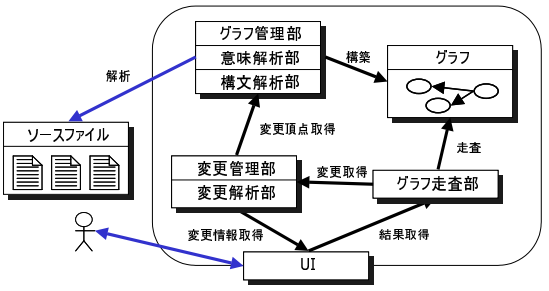


図 10: ツール構成

解析部（Java 影響波及解析ライブラリ）

解析部は JAVA で記述されており，次の 3 つの部分から構成されている．

グラフ管理部: JAVA プログラムに対し構文解析および意味解析を行い，MOG，MAG の構築を行う．

変更管理部: UI 部より与えられたプログラムの変更情報と，MOG，MAG の各節点との対応付けを行う．

グラフ走査部: 適用する被影響メンバ探索ルールを UI 部より取得し，それに従い MOG，MAG を用いた被影響メンバの抽出を行う．

ユーザインタフェース（UI）部

UI 部は JAVA で記述されており，下記の機能を有する．また，現在は CUI の実装のみとなっているが，GUI による実装も行う予定である．

変更の指定: ユーザが行う変更を解析部に与える．

探索ルールの指定: ユーザの選択した探索ルールを解析部に与える．

解析結果の表示: 解析部より取得した被影響メンバの抽出結果を表示する．

4.2 ツールの機能

ここではツールの有する機能を簡単に説明する．

被影響メンバの抽出，表示: 指定された探索ルールを適用し，被影響メンバの抽出および表示を行う．なお，ツールで定義されているもの以外の探索ルールを作成，適用することもできる．

抽出結果のフィルタリング: 複数の探索ルールの組み合わせによる被影響メンバの抽出を行う場合，ルールごとにフィルタリングして抽出結果の表示を行う．

アクセス関係の表示: ユーザが指定したメンバに関し，アクセスされる可能性のあるメンバ，およびアクセスする可能性のあるメンバを表示する．直接的なアクセスだけでなく，間接的なアクセスの表示も行い，間接的なアクセスに関しては，その深さを指定することができる．

5 まとめと今後の課題

本研究では，JAVA を対象言語とした影響波及解析手法の提案を行った．提案手法では，グラフに基づいて変更の検出および被影響部分の探索を行うことで，ユーザの目的に応じた様々な解析結果を導出できる．また，提案手法の実装である JAVA 影響波及解析ツールについて紹介した．

今後の課題としては，解析結果を視覚的に表示するための GUI の実装，ツールの有効性評価などがある．

参考文献

[1] A. Krishnaswamy, "Program Slicing: An Application of Object-Oriented Program Dependency Graphs," Technical Report TR94-108, Department of Computer Science, Clemson University, 1994.

[2] B. G. Ryder and F. Tip, "Change Impact Analysis for Object-oriented Programs," in Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001), pp.46-53, Snowbird, USA, June, 2001.

Technical Report TR94-108, Department of Computer Science, Clemson University, 1994.

[3] D. Kung, J. Gao, P. Hsia, and F. Wen, "Change Impact Identification in Object Oriented Software Maintenance," in Proceedings of the International Conference on Software Maintenance, pp.202-211, Victoria, Canada, September 1994.

[4] G. Rothermel and M. J. Harrold, "Selecting Regression Tests for Object-Oriented Software," in Proceedings of the International Conference on Software Maintenance, pp.14-25, Victoria, Canada, September 1994.

[5] G. Rothermel and M. J. Harrold, "A Safe, Efficient Regression Test Selection Technique," ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 2, pp.173-210, April 1997.

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

[7] 林崎, "ソースプログラムの変更と波及解析に基づく開発状況の把握法に関する研究," 北陸先端科学技術大学院大学 情報科学研究科 修士論文, 2000.

[8] L. Li, and A. J. Offutt, "Algorithmic Analysis of the Impact of Changes on Object-Oriented Software," International Conference on Software Maintenance (ICSM '96), pp.171-184, Monterey, USA, November 1996.

[9] M. Weiser, "Program slicing," in Proceedings of the 5th International Conference on Software Engineering, pp.439-449, San Diego, USA, March 1981.

[10] S. Eisenbach, and C. Sadler, "Changing Java Programs," in Proceedings of the International Conference on Software Maintenance (ICSM 2001), pp.479-487, Florence, Italy, November 2001.

[11] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An Empirical Study of Regression Test Selection Techniques," in Proceedings of the 20th International Conference on Software Engineering, pp.188-197, Kyoto, Japan, April 1998.

[12] W. Hetzel, "The Complete Guide to Software Testing," QED Information Sciences, Wellsley, Mass., 1984.

[13] X. Chen, W. T. Tsai, and H. Huang, "Omega - an Integrated Environment for C++ Program Maintenance," in Proceedings of the International Conference on Software Maintenance, pp.114-123, Monterey, USA, November 1996.

[14] Y. K. Jang, H. S. Chae, Y. R. Kwon, and D. H. Bae, "Change Impact Analysis for A Class Hierarchy," in Proceedings of the Asia Pacific Software Engineering Conference (APSEC'98), pp.304-311, Taipei, Taiwan, December 1998.