修士学位論文

題目

Java プログラムの変更作業を支援する影響波及解析システム

指導教官
井上 克郎 教授

報告者
近藤 和弘

平成 14 年 2 月 13 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 13 年度 修士学位論文

Java プログラムの変更作業を支援する影響波及解析システム

近藤 和弘

内容梗概

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，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に用いられる．回帰テストでは，影響波及解析に基づいて変更の影響を受ける部分を識別することにより，テスト対象を限定し，適用テストケースを必要最小限に抑えることができる．なお，既存の影響波及解析手法は，回帰テストでの利用が前提となっている．

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．特に，近年多く利用されている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存在するため，影響波及解析に基づくプログラム変更の支援は有効であると考えられる．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．また，実利用を考慮した影響波及解析システムは未だ存在していない．

本論文では，Java を対象言語として，クラスのメンバ間の関係を表現する 2 つのグラフ，メンバオーバーライドグラフ，メンバアクセスグラフを利用した影響波及解析手法の提案を行う．本手法により，オブジェクト指向言語である Java の特徴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，抽出を行うことができる．また，提案手法を Java 影響波及解析システムとして実装し，その有効性を検証した．

主な用語

影響波及解析 (Change Impact Analysis)
オブジェクト指向プログラム (Object-Oriented Program)
Java

目 次

1　まえがき

ソフトウェア保守工程において，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 [14]．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．

ソフトウェアに加えられた変更による影響を受ける部分（被影響部分と呼ぶ）を識別するための手法として，影響波及解析が提案されている．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*）[13] への利用が挙げられる．回帰テストは，

Step 1: 被影響部分の識別

Step 2: 修正コンポーネントの再テスト方法の決定

Step 3: 再テストによる補償範囲の認識

Step 4: テストケースの選択，再利用，修正，新規作成

という過程をたどるが，**Step 1:** 被影響部分の識別 においては，影響波及解析に基づいてテスト対象を限定し，適用テストケースを必要最小限に抑えることができる．既存の影響波及解析手法は，この回帰テストへの利用を前提としたものになっている．

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し，従来の手続き型プログラムに比べ，変更行った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．さらには，プログラム中のあるモジュールの修正が，他のモジュールでのバグの原因となることもある．そのため，これらを考慮した解析手法が望まれる．

本論文では，Java[7] を対象言語として，クラスのメンバ間の関係を表現する 2 つのグラフ，メンバオーバーライドグラフ (Member Override Graph，MOG)，メンバアクセスグラフ (Member Access Graph，MAG) を利用した影響波及解析手法の提案を行う．本手法により，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，抽出を行うことができる．また，提案手法を，JAVA 影響波及解析システムとして実装し，手法の有効性を検証した．

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 JAVA 影響波及解析システムについて述べ，5 で評価する．最後に，6. でまとめと今後の課題について述べる．

2　Java プログラムに対する影響波及解析

本節では，既存の影響波及解析手法と，本研究の対象となるオブジェクト指向言語 JAVA に対する影響波及解析について説明する．さらに，既存手法の問題点について考察する．

2.1　影響波及解析

影響波及解析（*Change Impact Analysis*）とは，プログラム変更による影響を受ける部分（**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, 5, 10, 15, 16] されている．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 [15]．数百クラスにもおよぶような大規模ソフトウェアに対して変更を行う際は，変更の影響を受けるクラスも多く存在すると考えられるが，クラス単位の解析を行うことにより，それらを容易に把握することができる．

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 [10, 16]．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, 5]．**プログラムスライス**（*Program Slice*）[11] 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影響を受けているかを特定できないため，影響の予測としては効果が薄く，回帰テストにおいて再テストの必要がないメンバも解析結果に含み得ることになる [8]．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といったメンバに関する変更による被影響部分の抽出を目的としているため，メンバ単位での解析に着目し，メンバを被影響部分の単位とする．

2.2 Java プログラムに対する影響波及解析

本研究では，JAVA を対象言語とした影響波及解析手法を提案する．以降，JAVA プログラムについて簡単に説明し，例を用いて JAVA プログラムに対する影響波及解析の説明を行う．

2.2.1 オブジェクト指向言語 Java の特性とそれによる影響

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もある．

オブジェクト指向言語の一つである JAVA で記述されたプログラムに対する変更は，メソッドのオーバーライド，フィールドの隠蔽といった概念により様々な影響が引き起こされる [4, 12]．

以下では，メソッドのオーバーライド，フィールドの隠蔽について説明する．

メソッドのオーバーライド

クラスがインスタンスメソッドを宣言している場合，そのメソッド宣言は，そのクラスのスーパクラス及びスーパインタフェース内で同じシグネチャをもつ全てのメソッドを**オーバーライド**（*Override*）すると呼ぶ。オーバーライドされなければ，そのスーパクラス及びスーパインタフェースのメソッドは，クラス内のコードからアクセス可能である。

図 1 は，メソッドのオーバーライドの例と，プログラムの実行結果を示している．クラス `PaintedPoint` は，クラス `Point` を継承しているため，`Point` のメソッド `setPosition()` を継承し，`Tester::test()` 中のメソッド呼び出し式 `p.setPosition(10, 20)` は，このメソッドを呼び出す．

ところが，`Point` のメソッド `toString()` は，`PaintedPoint` の同シグネチャのメソッドによってオーバーライドされるため，`Tester::test()` 中のメソッド呼び出し式 `p.toString()` は，`Point::toString()` ではなく，`PaintedPoint::toString()` を呼び出すことになる．

また，クラスで宣言されたメソッドが `abstract` ではなく，そのクラスのスーパクラス及びスーパインタフェースで同じシグネチャをもつ `abstract` メソッドが存在する場合，そのメソッド宣言は，`abstract` メソッドを**実装**（*Implement*）すると呼ぶ。

図 1 では，`abstract` メソッド `Paintable::paint()` が，`PaintedPoint::paint()` によって実装されている．

```
class Point {
  protected int _x, _y;

  public void setPosition(int x, int y){
    _x = x; _y = y;
  }
  public String toString(){
    return "(" + _x + ", " + _y + ")";
  }
}
```

```
interface Paintable {
  public static final int BLACK = 0;
  public static final int WHITE = 1;

  public abstract void paint(int color);
}
```

```
class PaintedPoint extends Point
                    implements Paintable {
  protected int _color;

  public void paint(int color){
    _color = color;
  }
  public String toString(){
    return "(" + _x + ", " + _y + ")"
                + " color:" + _color;
  }
}
```

```
class Tester {
  void test(){
    PaintedPoint p = new PaintedPoint();
    p.setPosition(10, 20);
    p.paint(Paintable.WHITE);
    System.out.println(p.toString());
  }

  public static void main(String args[]) {
    Tester t = new Tester();
    t.test();
  }
}
```

実行結果

```
% java Tester
(10, 20) color:1
```

図 1: メソッドのオーバーライド

フィールドの隠蔽

クラスがある特定の名前をもつフィールドを宣言した場合，そのクラスのスーパクラス及びスーパインタフェースの中の，それと同じ名前をもつ全てのアクセス可能なフィールドの宣言を**隠蔽**（*Hide*）すると呼ぶ。また，あるフィールド宣言が他のフィールド宣言を隠ぺいする場合，二つのフィールドが同じ型をもつ必要はない。

図2は，フィールドの隠蔽の例と，プログラムの実行結果を示している．クラス `RealPoint` は，クラス `Point` を継承しているため，`Point` のフィールド `_name` を継承し，`RealPoint` の宣言内では，単純名 `_name` は，`Point` 内で宣言されたフィールドを参照する．

ところが，`Point` の `int` 型のフィールド `_x`，`_y` は，`RealPoint` の同名フィールドによって隠蔽されるため，`RealPoint` の宣言内では，単純名 `_x`，`_y` は，`RealPoint` 内で宣言された `double` 型のフィールドを参照する。

```
class Point {
  protected String _name;
  protected int _x, _y;

  public Point(String name){
    _name = name;
  }
  public void setPosition(int x, int y){
    _x = x; _y = y;
  }
}
```

```
class RealPoint extends Point {
  protected double _x, _y;

  public RealPoint(String name){
    super(name);
  }
  public void setPosition(double x, double y){
    _x = x; _y = y;
  }
  public String toString(){
    return _name + "(" + _x + ", " + _y + ")";
  }
}
```

```
class Tester {
  void test(){
    RealPoint p = new RealPoint("P");
    p.setPosition(1.0, 2.0);

    System.out.println(p.toString());
  }

  public static void main(String args[]) {
    Tester t = new Tester();
    t.test();
  }
}
```

実行結果

```
% java Tester
P(1.0, 2.0)
```

図 2: フィールドの隠蔽

2.2.2 影響波及解析の適用範囲とその例

ここでは，例を用いて JAVA プログラムに対する影響波及解析の説明を行う．図 3 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．

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 [6]，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．

```
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 3: メソッドの追加による影響（下線部：被影響メソッド）

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, 16] 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．

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，`abstract` メソッドの実装，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 4(b) は，図 4(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 4(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 4(c) は，図 4(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 5 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．

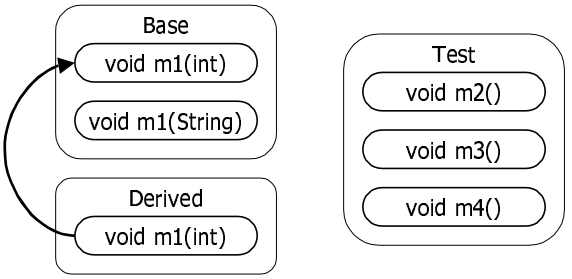
MAG は，MOG と同様の解析による MAG 節点の抽出，およびメソッド呼び出し式，フィールドアクセス式の解析による MAG 辺の抽出により構築される．

```
public class Base {
   public void m1(int x) {
      //do something
   }
   public void m1(String x) {
      //do something
   }
}
```

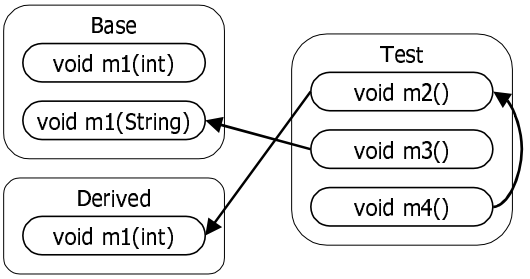
```
public class Derived
                extends Base {
   public void m1(int x) {
      //do something
   }
}
```

```
class Test {
   Derived obj;
   public void m2() {
      obj = new Derived();
      obj.m1(10);
   }
   public void m3() {
      obj = new Derived();
      obj.m1("ten");
   }
   public void m4() {
      this.m2();
   }
}
```

(a) プログラム

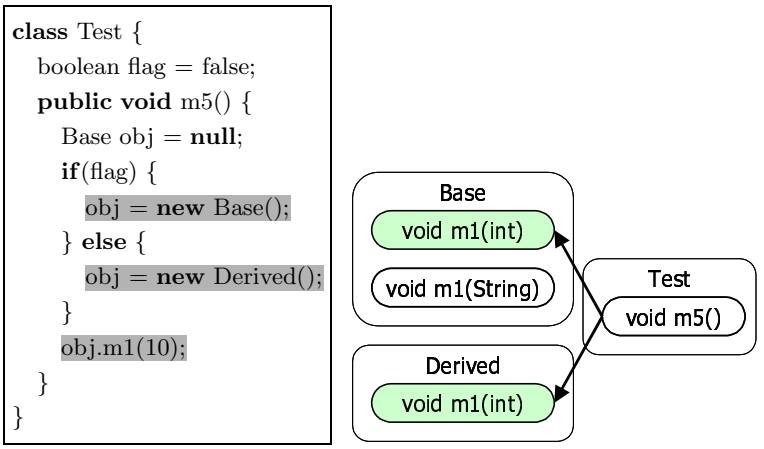


(b) MOG



(c) MAG

図 4: 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）



(a) プログラム　　(b) MAG

図 5: インスタンスの型が特定できない例

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 6 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．直感的には，変更によって新たにオーバーライドされるメソッド (M7)，変更によって新たな呼び出し先が発生し，実行結果が変化し得るメソッド (M3) などが直接被影響節点となる．

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 7 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．直感的には，変更メンバがオーバーライドするメンバを修正した場合に, オーバーライド関係が変化し得るメソッド (M10)，実行結果が変化し得るメソッドを呼び出していることで, 推移的に実行結果が変化し得るメソッド (M1) などが間接被影響節点となる．

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 6 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 6 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 6 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 6 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 6 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 7 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 7 の MOG 節点 {M10} および MAG 節点 {M1，M3，M4}）

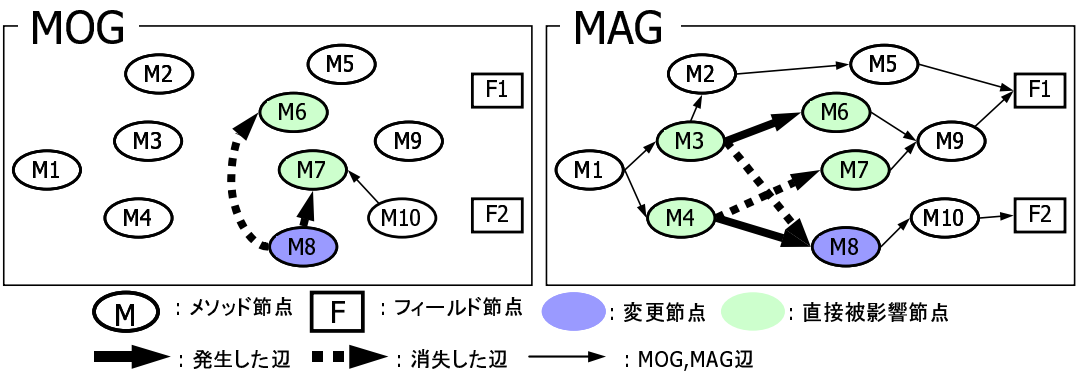


図 6: 直接被影響節点の例（表 1 参照）

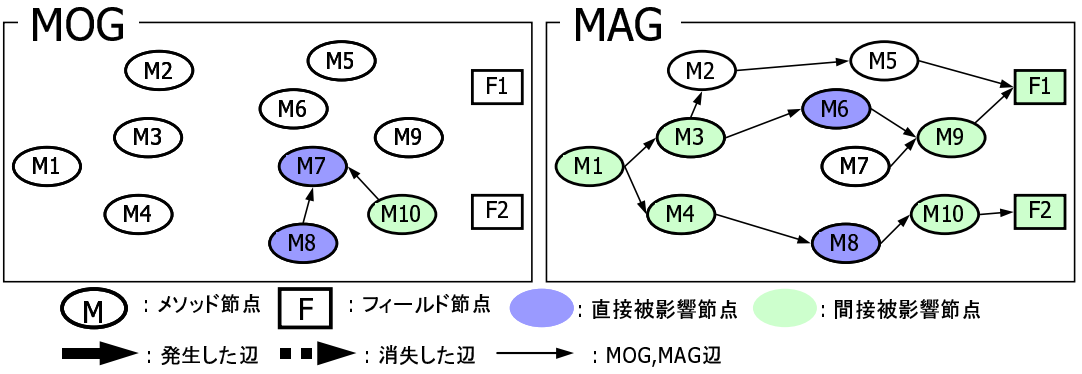


図 7: 間接被影響節点の例（表 2 参照）

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}

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: アクセス変化メンバ抽出

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 8 に，M8 が変更メンバである場合の抽出例を示す．

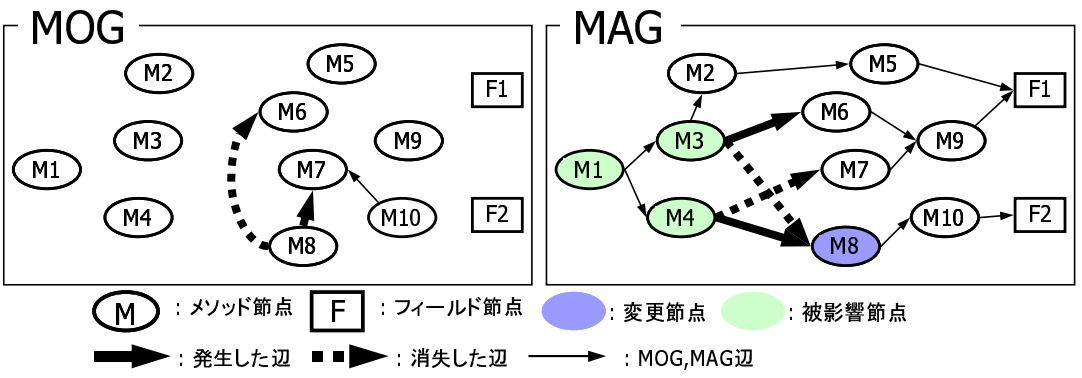


図 8: R1: アクセス変化メンバ抽出

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 9 に，M8 が変更メンバである場合の抽出例を示す．

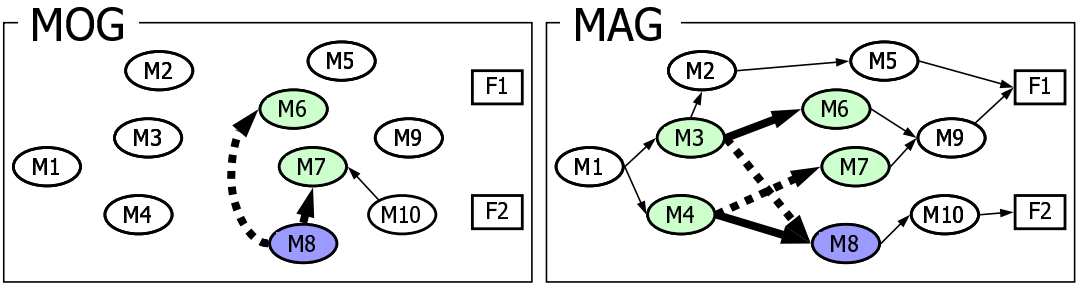


図 9: R2: 関係変化メンバ抽出

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 10 に，M8 が変更メンバである場合の抽出例を示す．

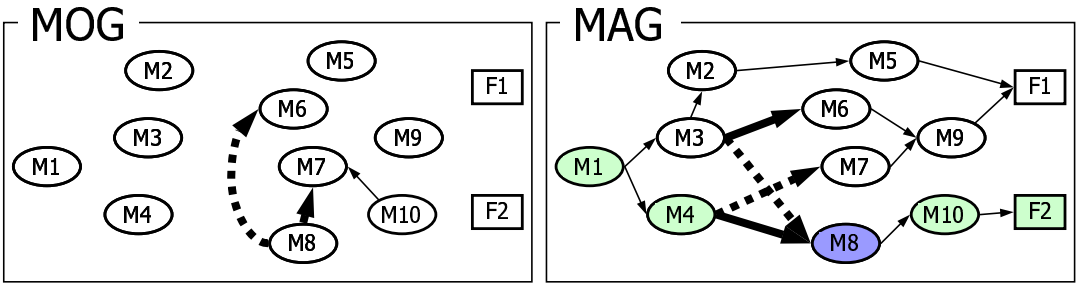


図 10: R3: 間接アクセスメンバ抽出

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 3 の `Student` への `toString()` メソッドの追加に対して，前節で定義した「アクセス変化メンバ抽出」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 11 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．

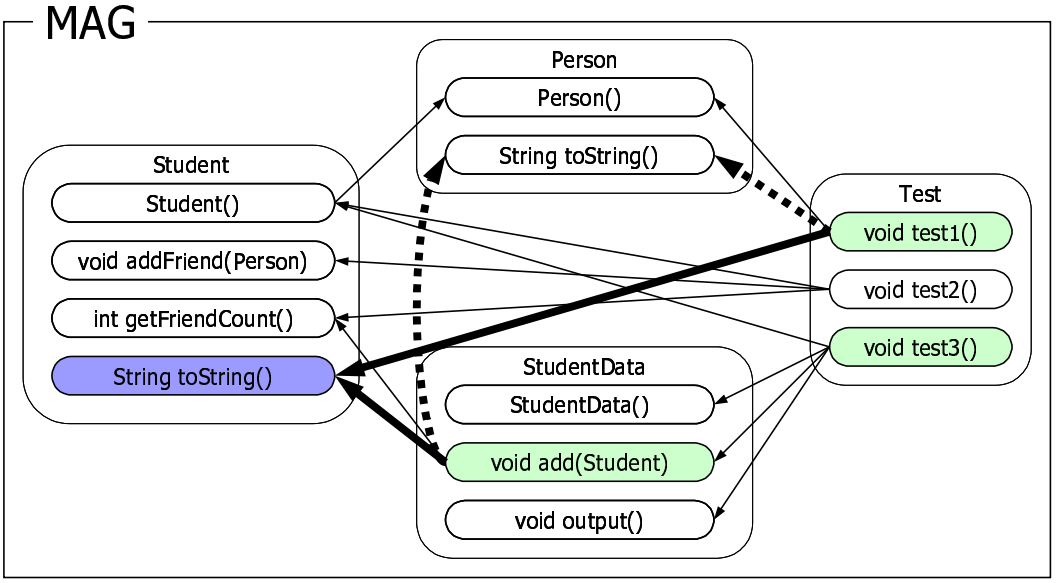


図 11: 「アクセス変化メンバ抽出」適用例

4　Java 影響波及解析システム

本節では，提案手法の実装である JAVA 影響波及解析システムの構成，およびその機能について述べる．

4.1　概要

本システムは，JAVA プログラムにおける，メンバ単位の変更で生じる影響を解析，表示するシステムである．本システムの利用の流れは次のようになる（図 12 参照）．

Step1: ユーザは，システムに対して変更前の状態を表すグラフ構築を要求する．

Step2: システムは，指定されたソースファイルを解析し，グラフを構築する．

Step3: ユーザは，ソースファイルに対して変更を行う．

Step4: ユーザは，システムに対して影響波及解析の実行を要求する．

Step5: システムは，ソースファイルを解析し，グラフを再構築する．

Step6: システムは，変更前後のグラフの差分から，被影響メンバを抽出する．

Step7: システムは，抽出された被影響メンバを表示する．　→ **Step3 へ**

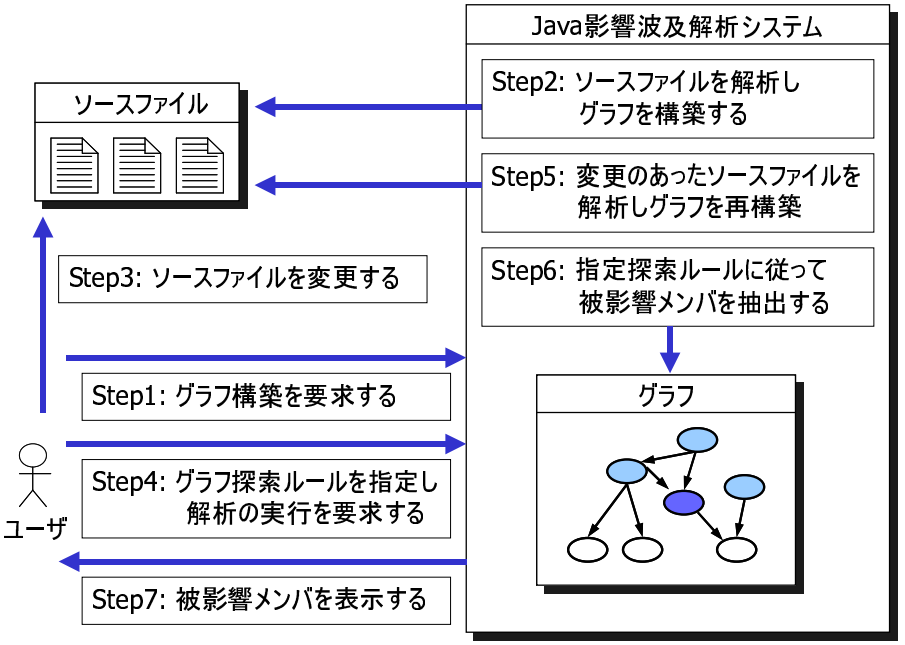


図 12: システム利用の流れ

4.2 利用ツール

本システムでは，提案手法による影響波及解析に必要となる，JAVA プログラムの字句解析，構文解析，意味解析を行うためのツールとして javac[17] を利用し，システムの GUI 部を構成するテキストエディタとして jEdit[19] を利用する．

4.2.1 javac

javac は，JDK[18] 付属の JAVA コンパイラであり，JAVA で記述されたクラスとインタフェースの定義を読み取り，**Java バーチャルマシン**（*Java Virtual Machine*）で実行可能な**バイトコード**（*Byte Code*）にコンパイルする．

影響波及解析はプログラムに変更を行うことが前提となるため，その変更によって構文的，意味的な正しさが失われてしまう可能性もある．したがって，MOG，MAG を構築する際，同時にこれらのチェックを行うことも求められる．

本システムでは，バイトコードの生成機能を削除し，JAVA プログラムの字句解析，構文解析，意味解析を行うためのツールとして，javac を用いた．具体的には javac のソースコード中に，必要情報を取得するためのコードを埋め込むことで実現した．その際，javac のソースコードに対する変更は最小限となるよう心がけたため，実際に埋め込んだコードは 100 行未満となり，JDK のバージョンアップ等への対応も容易である．

4.2.2 jEdit

jEdit は, Pestov らによって開発されたプログラマ向けのテキストエディタであり，次のような特徴を持っている．

- JAVA で記述されているため，動作環境を選ばない．
- プラグイン [20] に対応しており，拡張性が高い．
- ユーザが独自のプラグインを作成するための API が用意されている．
- オープンソースソフトウェアである．

本システムでは，ソースコードの変更を行うテキストエディタとしてこの jEdit を用いた．また，解析結果の表示を行う機能は，jEdit のプラグインとして実装した．これにより，jEdit のテキストエディタとしての高度な機能を利用しながら，提案手法による影響波及解析の実行，解析結果の表示を行うことができる．

4.3 システム構成

本システムは，実際に提案手法による影響波及を行う解析部と，ユーザに対するインタフェースとなるGUI部で構成されている．図13はシステムの各構成要素の関連を示したものである．また，以下の説明では，MOG，MAGを単にグラフと呼ぶ．

4.3.1 解析部

解析部は，次の3つの部分から構成されている．

JCIA

グラフ管理部，変更管理部，グラフ走査部に指示を出すことにより，Javaプログラムに対して提案手法による影響波及解析を行う．

グラフ管理部

指定されたJavaプログラムに対応するグラフの構築を行う．

変更管理部

変更前後のプログラムに対応するグラフを比較し，節点, 辺の発生, 消失といったグラフの変化を検出する．

グラフ走査部

与えられた被影響メンバ探索ルールに従ってグラフを走査し，被影響メンバの抽出を行う．

javac'

Javaプログラムのソースコードに対し，字句解析，構文解析，意味解析を行うと共に，グラフ構築に必要な情報を収集する．

4.3.2 GUI部

GUI部は，次の2つの部分から構成されている．

JCIB

jEditのプラグインとして動作し，ユーザからの解析要求を解析部に伝え，解析結果をユーザに表示する．

jEdit

実際にJavaプログラムの変更を行う際のエディタとなり，解析結果の表示においては，JCIBと連携動作することにより効果的な表示を実現する．

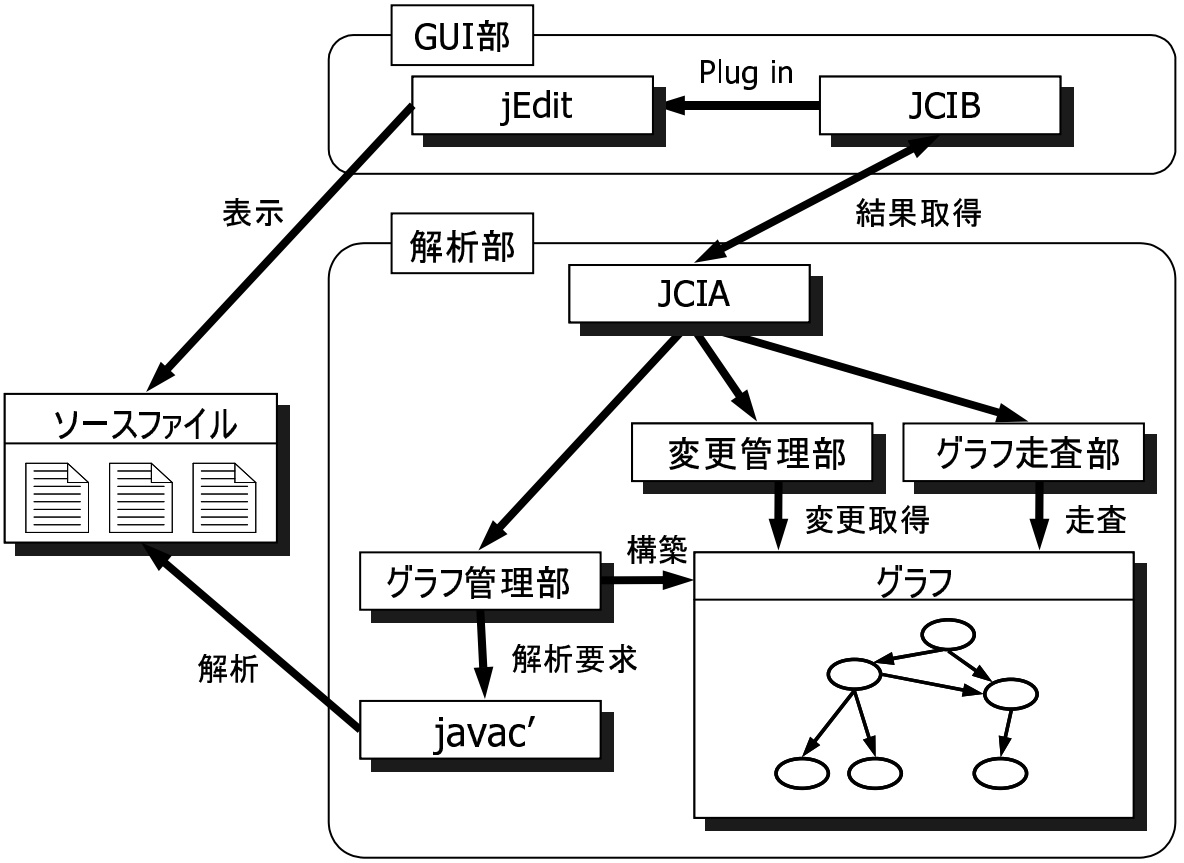

GUI部
Plug in
jEdit
JCIB
表示
結果取得
解析部
JCIA
ソースファイル
変更管理部
グラフ走査部
変更取得
走査
構築
グラフ管理部
グラフ
解析要求
解析
javac'

図 13: システム構成

4.4 実装機能

以下ではシステムに実装した機能を順に説明する．

ユーザが影響波及解析を行う場合は，システムを起動し，変更前の状態のグラフを構築しておく必要がある．システム起動時の jEdit と JCIB の画面写真をそれぞれ図 14，図 15 に示す．jEdit の機能についての説明は省略するが，プログラムの変更を行うために必要な機能を備えているだけでなく，コーディング用のエディタとしても優れたものになっている．JCIB は，jEdit のプラグインに実装可能な一体化機能を備えているため，ユーザの要求によっては，図 15 のように別ウインドウとして表示させるだけでなく，図 16 のように jEdit ウインドウ内に一体化して表示させることも可能である．

JCIB は，**被影響メンバ表示ツリー**(図 15 左部) と，**メンバ情報表示ペイン**(図 15 右部) と，各種命令ボタンで構成されている．被影響メンバ表示ツリーは，解析結果として得られた被影響メンバをツリー構造で表示し，メンバ情報表示ペインは，指定されたメンバの呼び出し関係などの情報を表示する．詳細はそれぞれ 4.4.3，4.4.4 で述べる．

4.4.1 グラフの構築

JCIB 起動時は，グラフが構築されていない状態であるため，変更前のソースコードをシステムにチェックインし，グラフ構築を行う必要がある．この状態では，メンバ情報表示ペインに，「no checkin」というメッセージが表示されている．グラフの構築は，JCIB の「Checkin」ボタンを押すだけで行うことができるが，解析対象となるファイルが未指定である場合は，「Config」ボタンを押し，**環境設定ダイアログ**(図 17) を表示させ，その指定を行う必要がある．指定項目は，解析対象ソースファイルへのパスと，ライブラリ等を使用するプログラムである場合は，それらへのクラスパスである．JCIB 起動時に自動的にチェックインを行い，グラフを構築するようすることも，このダイアログ上で設定可能である．チェックインを行った後，ユーザは jEdit 上で任意のソースコードに対して変更を行うことになる．

4.4.2 探索ルールの指定

適用する探索ルールの指定は，**探索ルール指定ウインドウ**(図 18) で行う．3.4.2 で挙げた 3 つの探索ルールから適用するルールを選択することができ，システムで定義されているもの以外の探索ルールを作成する場合は，チェックボックスで抽出する被影響節点を指定すればよい．間接被影響節点を抽出する場合は，チェックボックス横の入力欄に数値を入力することで，メソッド呼び出しの深さを指定することができる．また，「Preview」を押すことにより，現在の解析結果に対して様々な探索ルールを一時的に適用することもできる．

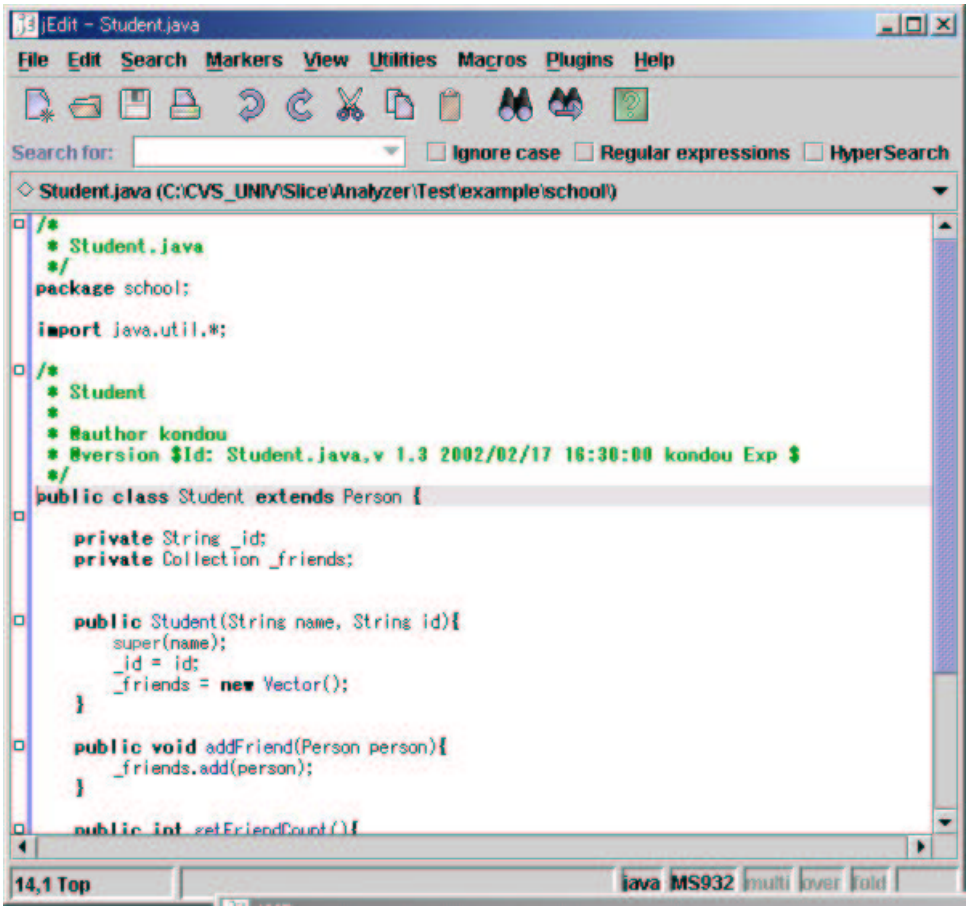


図 14: jEdit

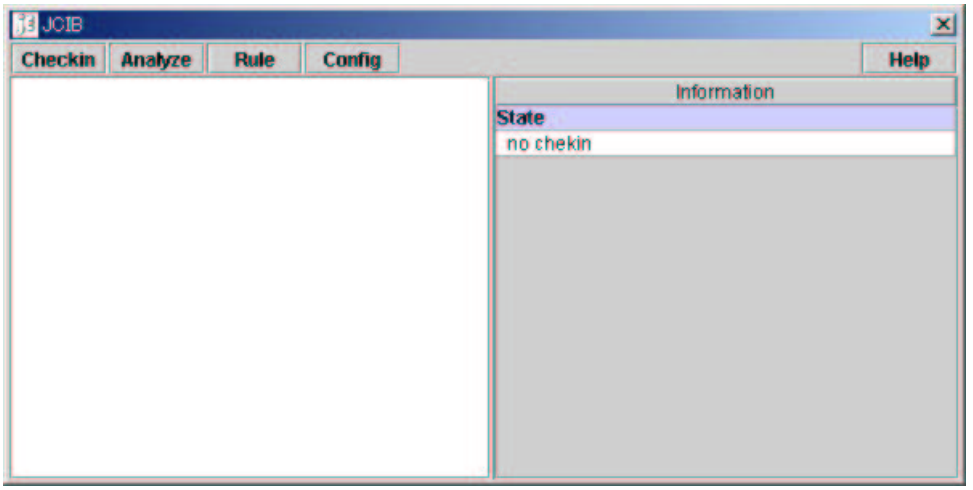


図 15: JCIB

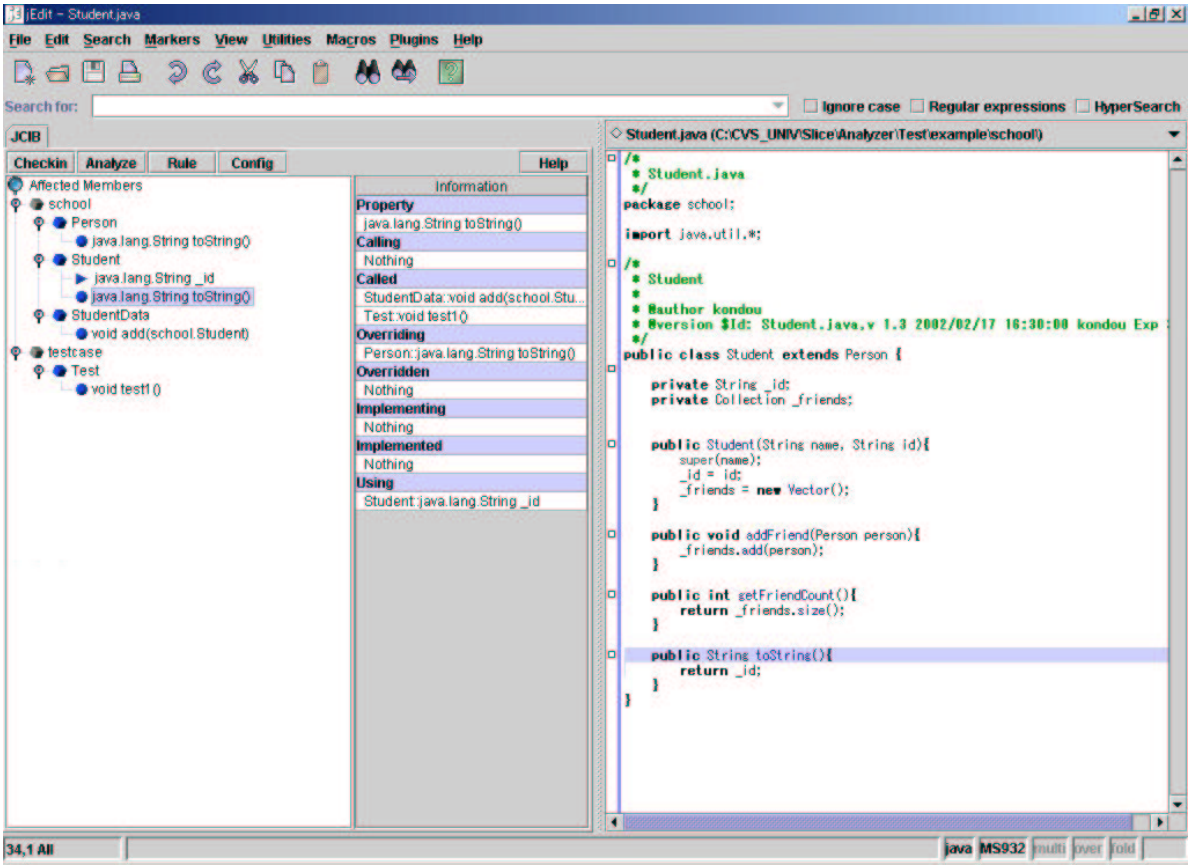


図 16: JCIB と jEdit の一体化

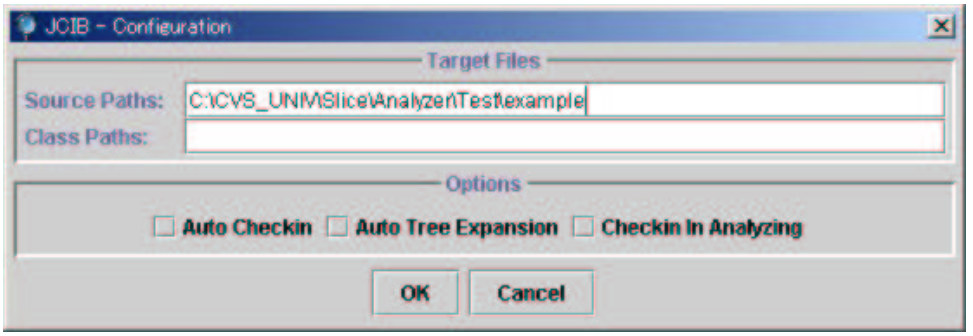


図 17: 環境設定ダイアログ

4.4.3 被影響メンバの抽出，表示

プログラムに変更を加えた後，その変更による被影響メンバの抽出および表示を行うには，JCIB の「Analyze」ボタンを押せばよい．それにより，指定された探索ルールに基づく解析結果を被影響メンバ表示ツリー上に表示する (図 19)．

被影響メンバ表示ツリー

被影響メンバ表示ツリーは，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ものである．ツリーを構成するのノードには次の 6 種類がある．

- メッセージノード
  被影響メンバ表示ツリーのルートとなるノード．現在の状態に応じて様々なメッセージが表示される．
- パッケージノード
  被影響メンバを持つクラスおよびインタフェースが属するパッケージを表すノード．パッケージ中にパッケージが存在する（サブパッケージを持つ）場合，各パッケージごとに 1 つのパッケージノードが表示される．したがって，メッセージノードまたはパッケージノードの子ノードとなる．
- クラスノード，インタフェースノード
  被影響メンバを持つクラスおよびインタフェースを表すノード．メッセージノードまたはパッケージノードの子ノードとなる．
- メソッドノード，フィールドノード
  変更の影響を受けるメソッドおよびフィールドを表すノード．クラスノードまたはインタフェースノードの子ノードとなる．

図 19 では，パッケージノード `school` の子ノードとして，クラスノード `Person` が存在し，その子ノードとして，メソッドノード `toString()` が存在している．これは，`school.Person::toString()` が被影響メンバであることを表している．

JCIB と jEdit の連携

被影響メンバ表示ツリー上で，メソッドノードまたはフィールドノードが選択された場合 (図 20(a))，そのメンバに関する情報をメンバ情報表示ペインに表示する (4.4.4 参照) と同時に，jEdit 上に表示されたソースコードにおける対応個所にカーソルを移動させ，背景を薄青色にすることで強調表示を行う (図 20(b))．選択メンバが宣言されているファイルがオープンされていない場合は，自動的にファイルをオープンし，同様の処理を行う．

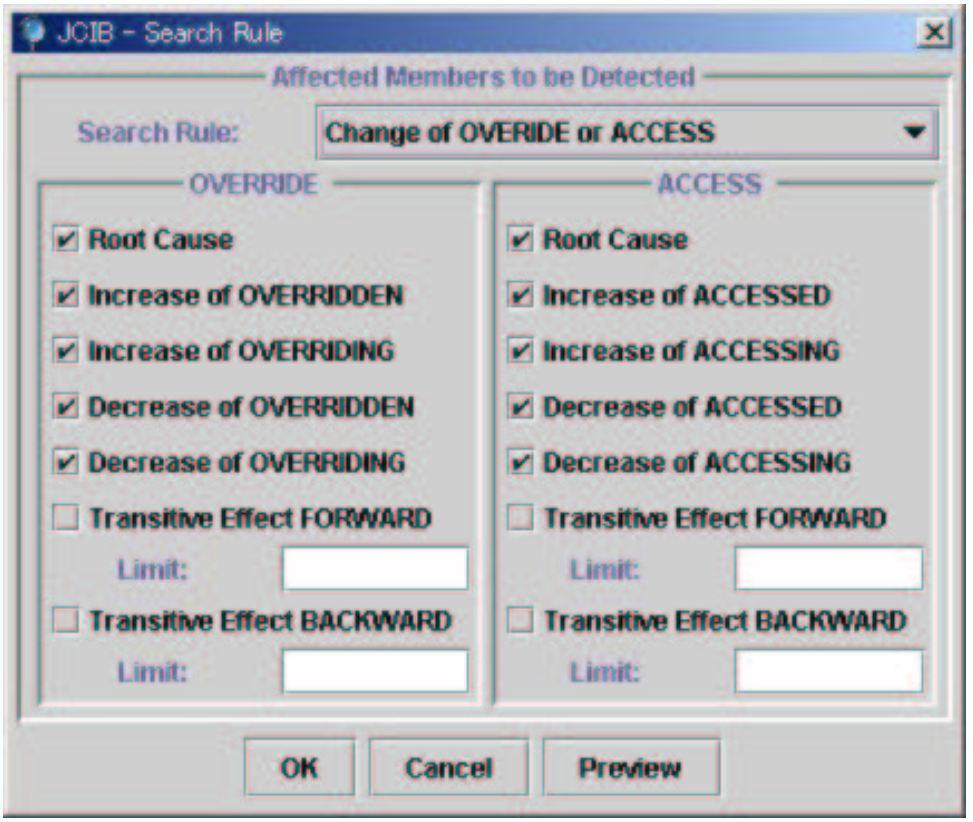


図 18: 探索ルール指定ウインドウ

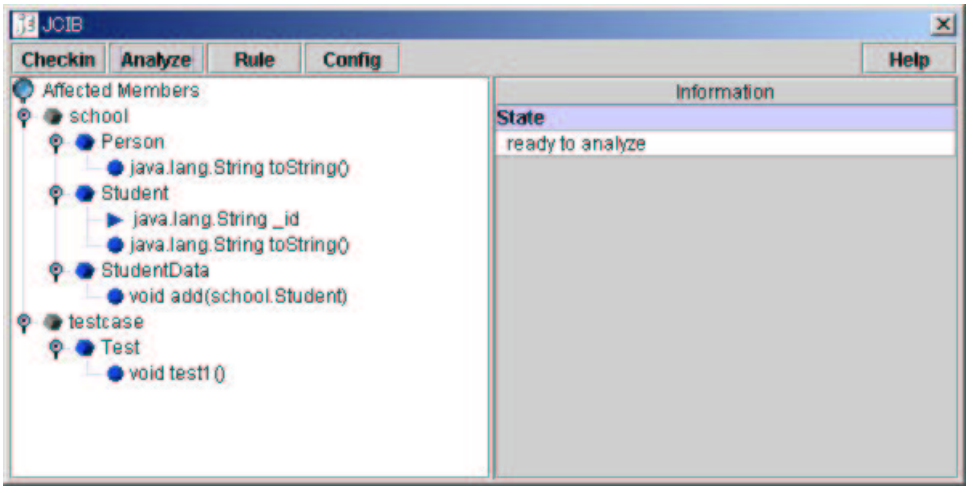
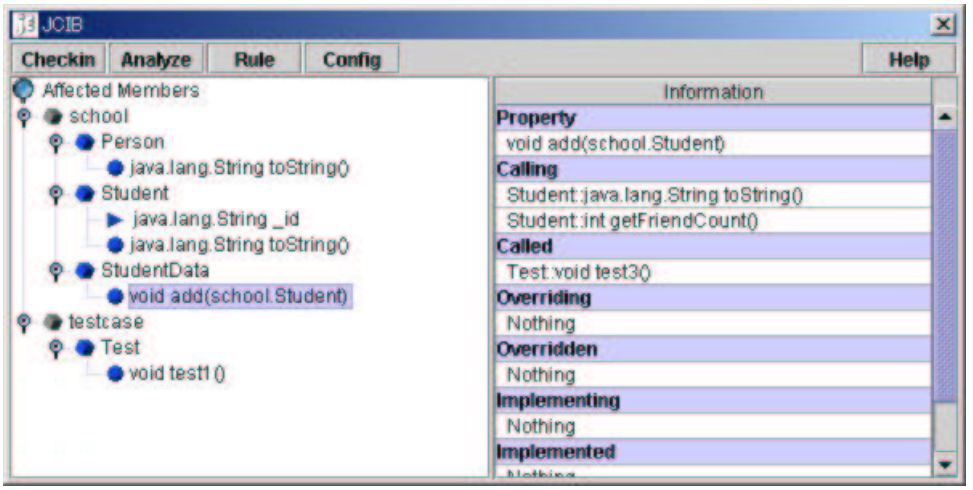
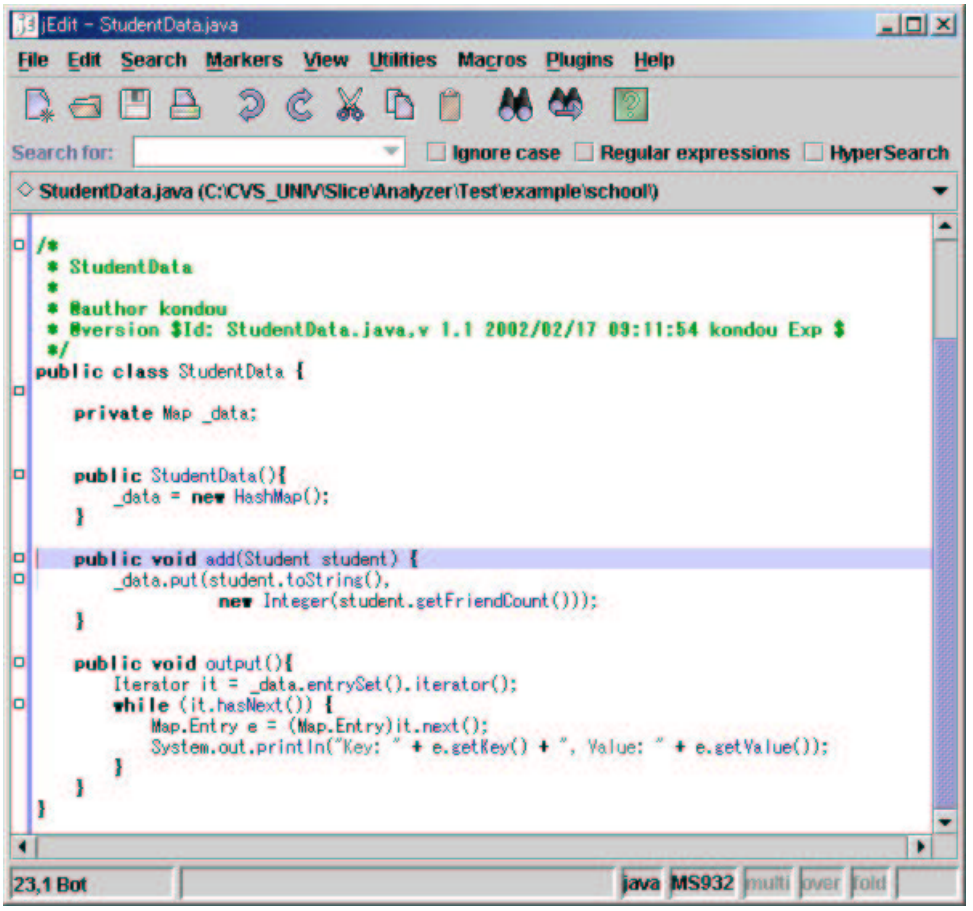


図 19: 被影響メンバの表示



(a) JCIB



(b) jEdit

図 20: jEdit と JCIB の連携

4.4.4 メンバ情報の表示

被影響メンバ表示ツリーにおいて，メソッドノードまたはフィールドノードが選択された場合，そのメンバに関する情報を，メンバ情報表示ペイン上に表示する (図 21).

メンバ情報表示ペイン

メンバ情報表示ペインは，指定メンバに関する情報を表示するものである．ユーザによって指定されたメンバに関し，次のような情報を表示する．

- 指定メソッドが呼び出すメソッド
- 指定メソッドを呼び出すメソッド
- 指定メソッドがオーバーライドするメソッド
- 指定メソッドをオーバーライドするメソッド
- 指定メソッドが実装する `abstract` メソッド
- 指定 `abstract` メソッドを実装するメソッド
- 指定メソッドが参照するフィールド
- 指定フィールドが隠蔽するフィールド
- 指定フィールドを隠蔽するフィールド

図 21 は，被影響メンバ表示ツリーでメソッド `Student::toString()` を表すメソッドノードを選択した場合のメンバ情報表示ペインを示している．表示された情報により，`toString()` が，`StudentData::add()`，`Test::test1()` から呼び出されていること，および `Person::toString()` をオーバーライドしていることを把握できる．

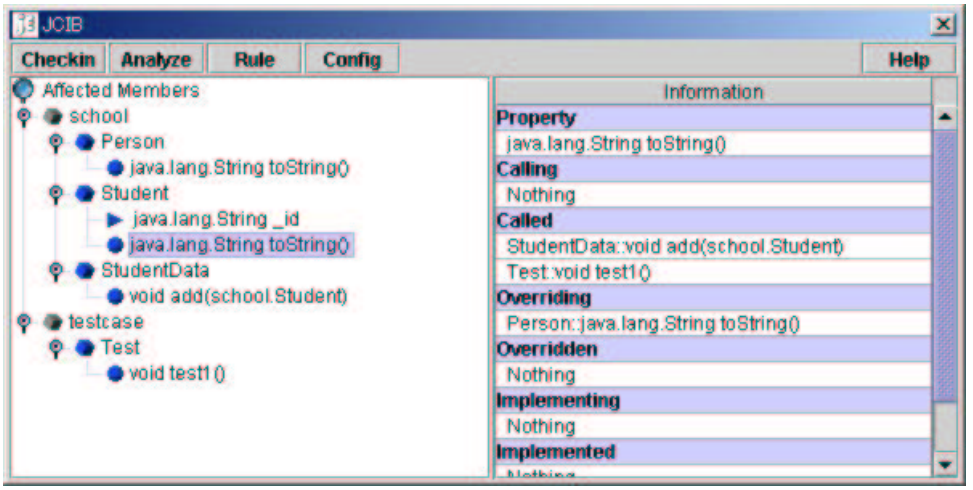


図 21: メンバ情報表示ペイン

4.5 開発

JAVA 影響波及解析システムの構成要素である，JCIA，JCIB は共に JAVA で記述を行った．コードはシステム全体で 1,1000 行程度となり，それらの内訳は次のようになる．

- JCIA：約 5,000 行
- JCIB：約 6,000 行

5　システムの適用例と有効性

本システムの有効性の評価方法として，実際のソフトウェア開発者を被験者とし，システムを利用することでソフトウェアに対する変更の効率化が図れるかどうかを調べる実験を行うことが挙げられる．本論文では，そうした評価実験を行っていないため，ある基準バージョンのソフトウェアに対してシステムの適用実験を行い，システムによって特定された被影響メンバと，変更後のバージョンにおける変更個所とを比較し，その結果を考察することで有効性を検証する．

適用対象ソフトウェア

本システムの適用対象ソフトウェアとして，Java ベースのビルドツールである Ant を選択した．Ant は Jakarta Project で開発されているオープンソースソフトウェアであり，Ant v1.1 は，96 個のクラス (約 20,000 行) から構成されている．

適用実験

ここでは，Ant の開発過程において実際に行われた変更に対してシステムの適用を試みる．Ant は v1.1 から v1.2 において，様々な機能追加および修正が行われているが，その中の一つであるメソッド `Property::init()` の削除に着目し，このメソッドの削除による被影響メンバをシステムを用いて特定する．

まずは，`Property::init()` の削除によって生じた全ての直接的な影響を調べるために「関係変化メンバ抽出」を探索ルールとして指定し，解析を行った．その結果，多くの被影響メンバが抽出された (図 22(a)) ため，抽出する被影響メンバの種類を限定して解析を行うと，次のようなことを把握できる．

- オーバーライド関係が変化したもののみを抽出すると，`Property` の親クラス `Task` のメソッド `init()` がオーバーライドされなくなることを把握できる (図 22(b))．
- オーバーライド関係の変化により，メソッドのアクセス先が変化したメソッドを抽出すると，`TaskHandler::init()`，`Ant::execute()` 中のメソッド呼び出し式において，その呼び出し先が変化することを把握できる (図 22(c))．
- アクセス先が変化したメソッドを呼び出すことで，推移的に実行結果が変化し得るメソッドを抽出すると，動作テストを行うメソッドを限定することができる (図 22(d))．

実際の更新作業との比較に基づく考察

適用実験の結果をまとめると，次のようになる．

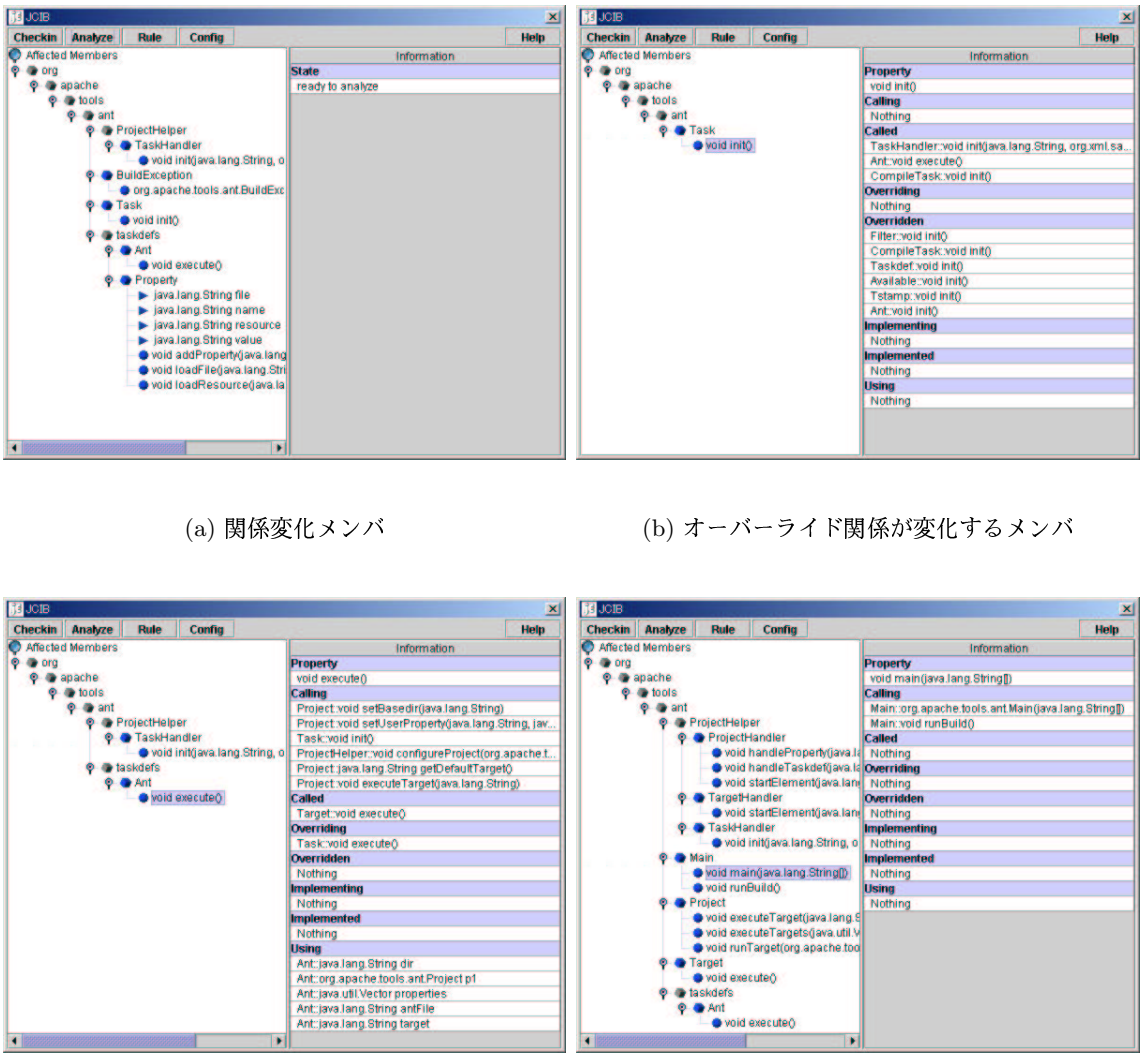
- Task::init() の，Property::init() によるオーバーライドが解除された．
- TaskHandler::init()，Ant::execute() において，Property::init() への呼び出しが Task::init() への呼び出しに変化した．

削除されたメソッド Property::init() の機能について考えると，その機能が不要な場合は削除され，必要な場合はプログラムの他の部分に移譲されるはずである．その場合，Property::init() の削除により影響を受けるメンバにおいて，何らかの対応を行う必要があると考えられる．そこで，上記の 3 つの被影響メンバ Task::init()，TaskHandler::init()，Ant::execute() が，v1.1 から v1.2 においてどのように修正されたかを調査した．

- Task::init()
  v1.1 から v1.2 において全く変更が行われておらず，Property::init() 中に存在していた機能は，オーバーライドの対象であった Task::init() に移譲されたわけではないことがわかった．
- TaskHandler::init()
  v1.1 から v1.2 においてメソッドに変更が加えられており，各バージョンにおけるソースコードは図 23 のようになっていた．Property::init() の削除とは関係のない，機能追加と考えられる部分を除くと，図 23 の網掛部が削除の影響により変更された部分であると思われる．v1.2 では，Task 型の参照変数 task が指し得るオブジェクト (Task クラスの子クラスである Property クラスのオブジェクトなど) に対して，v1.1 と同様に init() の呼び出しを行うだけでなく，場合によっては execute() の呼び出しも行うよう変更されていることがわかる．
- Ant::execute()
  v1.1 から v1.2 においてメソッドに変更が加えられており，各バージョンにおけるソースコードは図 24 のようになっていた．これにより，Property::init() の呼び出し式が，Property::execute() の呼び出し式に置換されていることがわかる．

上記の変更点より，Property::init() の機能は，Property::execute() に移譲されたと推測できる．実際に，v1.1 における Property::init() と v1.2 における Property::execute() の間には機能の一致が確認できた．

以上より，システムにより抽出された被影響メンバは，実際の変更作業において確かに修正が行われていたことがわかり，ソフトウェアに対して変更を行う際に本システムによって修正個所の特定を行うことは有効であると考えられる．今後は，本システムを用いることで変更作業に要する時間が短縮できることを，実際のソフトウェア開発者を被験者とした評価実験を行うことによって検証することが必要である．



(a) 関係変化メンバ　　(b) オーバーライド関係が変化するメンバ

(c) アクセス先が変化するメンバ　　(d) 実行結果が変化し得るメンバ

図 22: メソッド Property::init() 削除による被影響メンバ

```
private Task task;
  ...
public void init(String tag, AttributeList attrs) {
  ...
  task.setLocation(···);
  task.init();
  if (target != null) {
    task.setOwningTarget(target);
    target.addTask(task);
  }
}
```

(a) v1.1

```
private Task task;
  ...
public void init(String tag, AttributeList attrs) {
  ...
  task.setLocation(···);
  configureId(task, attrs);
  if (target != null) {
    task.setOwningTarget(target);
    target.addTask(task);
    task.init();
    wrapper = task.getRuntime
                          ConfigurableWrapper();
    wrapper.setAttributes(attrs);
  } else {
    task.init();
    configure(task, attrs, project);
    task.execute();
  }
}
```

(b) v1.2

図 23: v1.1 と v1.2 における TaskHandler::init()

```
public void execute() {
  ...
  Enumeration e = properties.elements();
  while (e.hasMoreElements()) {
    Property p = (Property) e.nextElement();
    p.init();
  }
  ...
}
```

(a) v1.1

```
public void execute() {
  ...
  Enumeration e = properties.elements();
  while (e.hasMoreElements()) {
    Property p = (Property) e.nextElement();
    p.execute();
  }
  ...
}
```

(b) v1.2

図 24: v1.1 と v1.2 における Ant::execute()

6　まとめと今後の課題

本論文では，オブジェクト指向言語である JAVA の特性を考慮した影響波及解析手法の提案を行った．また，提案手法を JAVA 影響波及解析システムとして実装し，適用例を挙げることで有効性を検証した．

既存の影響波及解析手法は，回帰テストへの利用を前提としたものになっており，影響の定義は，回帰テストを行うユーザを対象としたものであった．しかし，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．

本手法では，クラスのメンバ間の関係を表現する 2 つのグラフ，メンバオーバーライドグラフ，メンバアクセスグラフを定義し，プログラム変更による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．さらに，グラフの変化の種類によって変更の影響を受けるメンバを分類し，それらの探索ルールを選択して適用できる枠組を実現した．これにより，ユーザの目的に応じた影響の定義，抽出を行うことが出来る．

今後の課題としては，

- 解析精度の向上
  - Rapid Type Analysis[3] の利用による参照変数の指すオブジェクトの型の限定
  - プログラムスライスの併用による文単位での影響波及解析の実現
- 例外に対する変更など，対応可能な変更の種類の拡張
- システムを利用した評価実験

などが挙げられる．

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 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 ソフトウェア科学分野 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 ソフトウェア科学分野 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 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 大畑 文明 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] 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.

[3] D. F. Bacon, “Fast and Effective Optimization of Statically Typed Object−Oriented Languages,” Ph.D. Thesis, Computer Science Division, University of California, Berkeley, December, UCB/CSD-98-1017, 1997.

[4] 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.

[5] 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.

[6] 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.

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

[8] K. Rangarajan, P. Eswar, and T. Ashok, “Retesting C++ Classes,” in Proceedings of the Ninth International Software Quality Week, San Francisco, USA, May 1996.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] `http://java.sun.com/j2se/1.3/docs/tooldocs/win32/javac.html`, "javac - Java programming language compiler."

[18] `http://java.sun.com/j2se/1.3/`, "The JavaTM 2 Platform, Standard Edition."

[19] `http://www.jedit.org/`, "jEdit - Open Source programmer's text editor."

[20] `http://plugins.jedit.org/`, "jEdit - Plugin Central."