社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2006－SE－153 (2)
2006／7／10

# プログラム依存グラフを用いたアスペクトの干渉検出

平井孝† 丸山勝久†

†立命館大学 理工学研究科
〒525-8577 滋賀県草津市野路東 1-1-1

あらまし：アスペクト指向プログラミングでは，特定の処理とその処理を実行させる時点の条件を，クラスとは独立してアスペクトに記述する．これは関心事を分離するという点で有用だが，処理の流れが明示的でないため，実際の動作が把握しづらくプログラムが予想通りに動作しないことが起こる．本論文では，アスペクト指向言語 AspectJ に焦点をあて，同一時点に関連付けられたアスペクトに対して，実行順序の違いによって発生する干渉の検出を自動化する手法を提案する．この手法では，まず，アスペクトを任意の順序で織り込むことで得られる複数のソースコードを静的に解析し，プログラム依存グラフを作成する．次に，干渉が発生する可能性を，プログラム依存グラフによるプログラムの等価性判定を用いて検査する．提案手法を用いることによって，エラーの混入を防ぐことができる．

# Interference Detection of Aspects by using Program Dependence Graphs

Takashi Hirai† Katsuhisa Maruyama†
†Graduate School of Science and Engineering, Ritsumeikan University
1-1-1 Nojihigasi, Kusatsu, Shiga 525-8577, Japan

**Abstract** : In aspect-oriented programming, developers can describe some operations and their execution conditions by using aspects which are defined in isolation from classes. This mechanism has an advantage of separating cross-cutting concerns. However, it is hard to detect interference between aspects since the woven program contains several implicit method calls for the aspects. This paper proposes a new method of automatically detecting such interference existing in programs written in AspectJ. The proposed method finds aspects conflicting at the same program point, and then generates all programs containing several aspects that are concatenated in any order. Next it creates a program dependence graph for each of the programs, and judges if they have the same behavior based on isomorphism of their graphs. With our method, the developers can easily see possible errors resulting from the detected interference at compile time.

## 1. はじめに

プログラムの保守性・再利用性を高めるためには，なるべく関連の深いコードを近い場所に集め，他とは分離して記述するべきだという考え方がある．これを，一般に関心事の分離(Separation of Concerns)と呼ぶ．オブジェクト指向プログラミング(Object Oriented Programming：OOP)は，データとそれに深く関連する手続きを１箇所（クラス）にまとめ，関心事の分離を行う考え方である．しかしながら，OOP では，関心事の分離を完全には達成できないことが知られている．例えば，ロギングや同期処理などは，複数のクラスに処理がまたがってしまう．このような処理を横断的関心事(Cross-cutting Concern)と呼ぶ．アスペクト指向プログラミング(Aspect Oriented Programming：AOP)では，アスペクトと呼ばれる新しいモジュール単位を導入し，横断的関心事

を分離する．このような観点から，AOPは，プログラムの理解性や保守性の向上に貢献し，OOPを補う技術として注目を集めている[1]．

基本的にアスペクト内には，特定の処理とその処理を実行する時点の条件を記述し，処理の明示的な呼び出しは記述しない．このため，プログラム全体の動作の把握が困難だという問題がある．また，記述した複数のアスペクトが互いに干渉し，プログラムが開発者の予想通り動作しないことが起こりうる．このような状況を受けて，アスペクトに関連したエラーの発見・解消・予防に対しての研究成果やツールは既に数多く発表されている[4,9,13,14]．また，近年では，実装の前段階でアスペクトの発見を行うEarly Aspectsの研究[5]が進んでおり，この段階においてアスペクトの衝突の予見や優先度の決定を行う研究[6,15]も存在する．

本研究では，対象言語をJavaにAOPの概念を付け加えたAspectJ[3]に絞り，同一時点で衝突したアスペクトの動作順序の違いによって，プログラム全体としての実行結果に違いが発生する場合を干渉と定義する．

本論文では，アスペクト間の干渉検出を自動化する手法を提案する．提案手法では，まず，アスペクトを任意の順序で織り合わせることで得られる複数のプログラムに対して，それらのソースコードを静的に解析することでプログラム依存グラフを作成する．次に，アスペクト間で干渉が発生する可能性を，プログラム依存グラフによるプログラムの等価性判定を用いて検査する．提案手法を用いることによって，開発者が予期していなかった干渉を発見することができ，エラーの混入を防ぐことができる．

以降，２章ではアスペクト指向プログラミングに関して簡単に説明し，本論文で扱う問題点を述べる．３章で，プログラム依存グラフを説明し，４章で，プログラム依存グラフを用いたプログラムの等価性判定によるアスペクトの干渉検出手法を提案する．最後に，５章でまとめを述べる．

## 2. アスペクト指向プログラミング

### 2.1 AspectJ

AspectJは，オブジェクト指向言語であるJavaに，AOPを実現するための言語仕様を追加した言語であり，現在もっとも幅広く利用されているアスペクト指向言語である．

AspectJでは，プログラム実行における特定の時点をジョインポイント（join point），ある条件によって切り出したジョインポイントの部分集合をポイントカット（point cut），ポイントカットに関連付ける処理のことをアドバイス（advice）と呼ぶ．通常，アスペクト内には，ポイントカットとして切り出す部分の条件とそれに関連付けるアドバイスを記述する．

ジョインポイントの例としては，以下のようなものがある．

- メソッドの呼び出しや実行
- フィールドへの代入や参照

これらのジョインポイントに対して，アドバイスを実行するタイミングを，直前(before)，直後(after)，本来の処理を置き換える(around)の中から指定する．また，実際にアスペクトとクラスを結合する処理を織り込み(weaving)と呼ぶ．

### 2.2 問題点

AOPで関心事の分離を行う際には，「クラス側もしくはアスペクト側のどちらかのソースコードを見ただけでは，実際の動作を把握することができない」ことが良いとされている[8]．しかし，これをAOPの問題点と捉える人も多く，これによってアスペクトに関連した多くのエラーが発生する可能性がある．本研究では，実際の動作を容易に把握できないことを前提としつつ，アスペクトの織り込みにより発生するエラーを未然に防ぐことを目的とする．このために，以下に示す２つの特性を定義する．

衝突：同一のジョインポイントに複数のアドバイスが存在すること

干渉：衝突したアドバイスの実行順序の違いによって，全体としての実行結果に違いが生じること

例えば，図１のソースコードでは，`Base`クラスの`z(int)`メソッドで干渉が発生している．いま，このプログラムの記述者が望む挙動を得るには，`Mult`→`Puls`の順でアスペクトの処理が実行される必要がある．しかしながら，現行のAspectJの仕様では`Plus`→`Mult`の順で実行されるため，

```
public class Base{
  private int f;
  public int getF(){
    return this.f;
  }
  public int z(int x){
    return x * x;
  }
}
public aspect Plus{
  int around(int p, Base baseP) :
   execution(* Base.z(int))
   && args(p) && this(baseP) {
    int f = baseP.getF();
    int puls = p + f;
    return proceed(plus, baseP);
  }
}
public aspect Mult{
  int around (int m, Base baseM) :
   execution(* Base.z(int))
   && args(m) && this(baseM) {
    int f = baseM.getF();
    int mult = m * f;
    return proceed(mult, baseM);
  }
}
```

図1：干渉の例

開発者の望む処理は得られない．なお，図1内のproceed()メソッドは，aroundアドバイスによって置き換えられる本来の処理を実行する特殊なメソッド呼び出しである．

このようなアスペクトの干渉は，プログラム開発中に発生する可能性があるだけでなく，ポイントカットの記述方法（ワイルドカードの使用等）によっては，リファクタリングの際に不意に発生することもある．また，将来，アスペクト指向が広く使用されるようになると，このような衝突・干渉が発生する可能性は高くなる．

ここで，AspectJでは"declare precedence"という記述によって，アスペクトの優先度を変更することができる．例えば，図1の場合は，どちらかのアスペクト内に"declare precedence : Mult, Plus;"と記述することによって，Multアスペクトの優先度をPlusアスペクトよりも高めることができる．もし，プログラム開発時に干渉が自動的に検出できたとすると，開発者がアスペクト間の干渉発生の時点を容易に知ることができ，その時点において優先度を明示的に記述することを促すことが可能である．

## 3. プログラムの等価性判定

プログラム依存グラフ(Program Dependence Graph：PDG)は，デバッグやプログラムの保守などさまざまな用途に用いられている．PDGを用いて2つのプログラムの等価性を判定する手法は，文献[7]で提唱されている．PDGを用いて，プログラムの等価性を判定するためには3種類の関係を考慮する必要がある．

**(a) 制御依存関係**

制御依存関係CD(s, t)が存在するとは，以下の2つの条件が成立することを指す．

- 命令sが条件節（分岐命令 or ループ命令）
- 命令tが実行されるかどうかが，命令sの判定結果によって決定

等価性を判定するために，制御依存関係をさらに2つに分類する．

- True制御依存関係： 命令tが命令sのthen節内に含まれる
- False制御依存関係： 命令tが命令sのelse節内に含まれる

**(b) データ依存関係**

データ依存関係DD(s, t)が存在するとは，次の3つの条件が成立することを指す．

- 命令sで，ある変数wに値を代入
- 命令tで，wの値を参照
- 命令sから命令tに到達可能な制御フローがあり，途中で変数wへの代入文が無いような経路が少なくとも1つ存在

制御依存関係と同様に，等価性判定のために，データ依存関係をさらに2つに分類する．

- ループ経由データ依存関係： 命令s, tが同一のループL内に存在しており，sからtへの制御フローにLのループ命令を含む

- ループ独立データ依存関係： 命令s, tが同一のループ内に無い．また，同一のループL内にあっても，sからtへの制御フローにLのループ命令を含まない

(c) 定義順序関係

定義順序関係 DefOrd(s, t)が存在するとは，次に示す3つの条件が成立することを指す．

- プログラム上で，命令がs→tの順で並ぶ
- ある変数wが存在しており，s, tが両方ともwに値を代入
- ある命令uが存在して，DD(s, u)かつDD(t, u)である

2つの制御依存関係，2つのデータ依存関係，1つの定義順序関係という全部で5つの関係を考慮して作成したPDGが一致すれば，みかけ上のソースコードは異なっていても，それらの挙動が等価であることは，文献[7]で確認されている．

## 4. PDGを用いた干渉検出手法

### 4.1 干渉検出処理

提案手法は5つのステップで実現されている．ここで，本手法では，対象プログラムがコンパイル可能なことを前提にしている．以下，5つのステップについて説明する．

（1）衝突検出部

Javaで記述されたクラスと，AspectJで記述されたアスペクトのソースコードから，複数のアドバイスが関連付けられているジョインポイントを見つける．本手法は，静的解析に基づくので，プリミティブポイントカット”cflow”のような，ジョインポイントが動的に決定されるものについては考慮しない．

（2）手法適用可能性判定部

ステップ（1）で発見したアドバイスに対して本手法を適用する際，開発中のプログラムにおいてPDGの作成が困難な部分が無いかどうかを探す．例えば，例外処理や動的束縛(dynamic binding)が入ると，現段階の手法ではPDG作成のコストが大きくなりすぎるので適用外とする．また，リフレクションを含むプログラムは，PDGが作成できないため，対象外とする．

（3）組み合わせ生成部

衝突した複数のアドバイスを，優先度を変えて結合し，実行される可能性のあるソースコードを擬似的に作成する．

まず初めに，変数名を一意に特定できるように，”変数名#アドバイス・メソッドの通し番号#クラス・アスペクト名(完全限定名： Fully Qualified Name)”と変更する．

次に，アドバイスを任意の優先度で結合する．その際，アドバイスの引数について注意が必要である．ここでは，基本型と参照型に分けて説明する．

引数で基本型のものは，アドバイス同士が結合される部分に，実引数をアドバイス内の仮引数に代入する文を追加することによって解決する．しかし，この文はPDGの等価性判定を阻む要因になるため，後で削除する必要がある．そのため，この文が引数解決のために追加したものであることが後でわかるようにマーキングしておく．また，aroundアドバイスのproceed()メソッドの戻り値についても，引数のときと同様の方法をとる．このとき，”return#アドバイス・メソッドの通し番号#クラス・アスペクト名(完全限定名, Fully Qualified Name)”という名前の特殊な変数を使用する．

引数で参照型のものは，どのオブジェクトが同一のインスタンスを参照しているかを考慮しなければならない．そのため，参照型の引数は，あるアドバイスとその次に実行されるアドバイスの間で，どのオブジェクト同士が別名（alias）の関係にあるのか，という情報を残しておく．そして，同一のインスタンスを参照しているオブジェクトは，同一のオブジェクト名で置き換える．置き換えによる名前の統一には表1の規則を用いる．

また，proceed()メソッドの戻り値については，表1のaroundのargsの項の「引数」を「戻り値」に換えた規則を用いる．

（4） PDG作成部

開発中のプログラム外のアスペクトやクラス(JDKやLibraryなど)のAspect-oriented System Dependence Graph (ASDG) [10], Class Dependence Graph (ClDG) [11]をあらかじめ作成し，アスペクトやクラス内の5つの関係を要約しておく．開発中のプログラムについては，干渉検出のたびに関係の要約を作成する．そして，組み合わ

表１：オブジェクト名の統一規則

| 引数＼タイプ | before / after | around |
|---|---|---|
| this / target | thisは"this##クラス・アスペクト名（完全限定名）", targetは"target##クラス・アスペクト名（完全限定名）"で統一する. | |
| args | 織り込み対象の引数名で統一する. | 1.別名関係の中に本来のメソッドの引数があれば，その名前で統一する.<br>2.他のaroundアドバイスではなく、本来のメソッドの処理を呼ぶproceed()の引数があれば，その名前で統一する.<br>3.上の1,2を含むグループに属さない別名があれば、この段階で「干渉有り」と判断可能なので，名前の統一は行わない. |

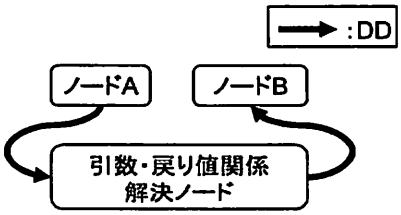


図２：AB間にエッジが引かれる関係

せ生成部で作成したソースコードに対してPDGを作成する．そのとき，それぞれの命令コードをラベルとして各ノードに保持させておくものとする．ソースコード内にメソッド呼び出しがある場合は，ASDG，CIDG内の該当するノードへエッジを引き，メソッド内部の処理をさらに展開して解析することは行わない．

ここで，ステップ（3）で追加した，基本型の引数・戻り値関係解決用のノードを削除する．そのとき，任意の３つのノード間に図２のような関係がある場合は，ノードＡからノードＢにデータ依存関係（DD）のエッジを引き，引数・戻り値関係解決ノードを削除する．

定義順序関係については，少し特殊な判定を行う．定義順序関係が成立するための条件は３章（c）で述べた．本手法では，プログラムを部分的にしか解析しないため，３章（c）での変数ｗがフィールドにあたる場合は，解析範囲内に命令ｕが存在していない場合でも，範囲外で定義順序関係が成立する場合が考えられる．よって，解析範囲の終了地点で，フィールドは必ず使用されると考え，解析範囲内に命令ｕが存在していない場合でも，定義順序関係を成立させる．

(5) PDG一致判定部

PDG作成部で作成したPDGの一致を判定する．グラフの一致判定には，各ノードのラベルの一致から判定し，その後グラフ構造の一致を判定する．

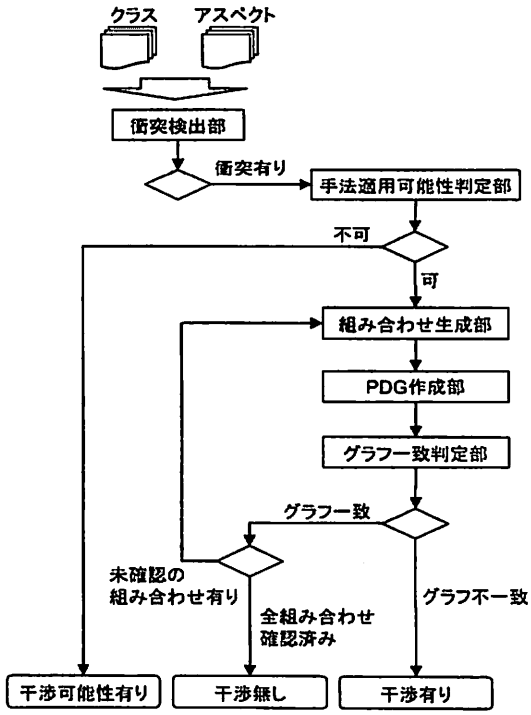


図３：干渉検出のフローチャート

干渉検出のフローチャートは図３のようになる．なお，衝突検出部で衝突無と判定された場合は，干渉は決して発生しない．

ここで，例として，本手法を図１のプログラムに対して適用する．まず，衝突検出部によって"Base.z(int)"に２つの around アドバイスが関連付けられていることが検出される．また，解析対象となる部分には手法が適用できない部分は無い．いま，優位性を変えてアドバイスの結合を行うとPlusの優先度を高めた図４と，Multの優

```
T1  p#1#Plus = x#2#Base;
N1  int f#1#Plus = this##Base.getF();
N2  int plus#1#Plus =
                p#1#Plus + f#1#Plus;
T2  m#1#Mult = plus#1#Plus;
N3  int f#1#Mult = this##Base.getF();
N4  int mult#1#Mult =
                m#1#Mult * f#1#Mult;
T3  return#1#Mult = z(mult#1#Mult);
T4  return#1#Plus = return#1#Mult;
```

図4：Plus の優先度を高めた擬似コード

```
T1  m#1#Mult = x#2#Base;
N3  int f#1#Mult = this##Base.getF();
N4  int mult#1#Mult =
                m#1#Mult * f#1#Mult;
T2  p#1#Plus = mult#1#Mult;
N1  int f#1#Plus = this##Base.getF();
N2  int plus#1#Plus =
                p#1#Plus + f#1#Plus;
T3  return#1#Plus = z(plus#1#Plus);
T4  return#1#Mult = return#1#Plus;
```

図5：Mult の優先度を高めた擬似コード

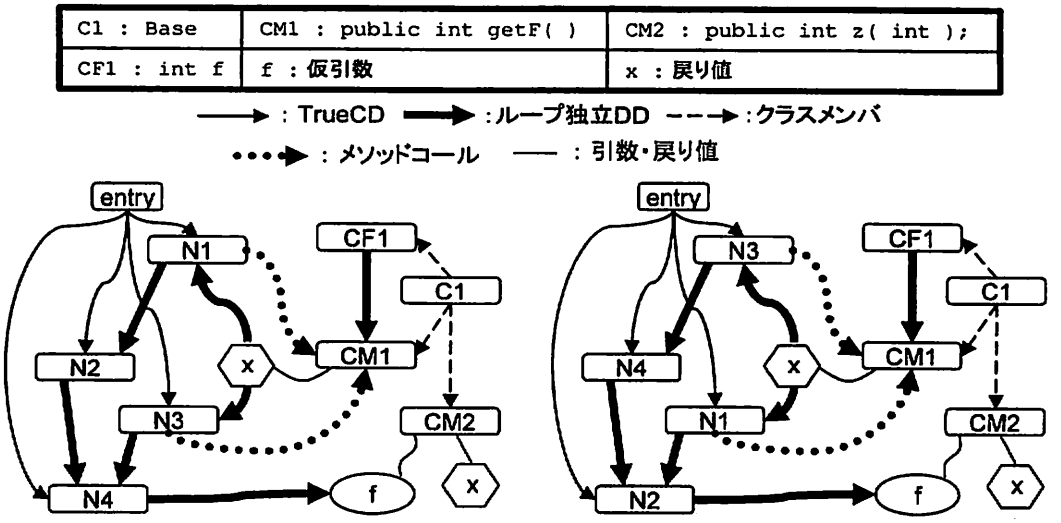


図6：図4のPDG

図7：図5のPDG

```
Advice1{A.field = 1;}
Advice2{A.field = 2;}
Advice3{System.out.println(A.field);}
Advice4{
        B.field = 3;
        System.out.println(B.field);
}
```

図8：不必要なアドバイスを含む例

先度を高めた図5の2種類の擬似コードが生成される．擬似コードの結合の際に使用した引数の別名関係は，this##Base = baseP#1#Pluse = baseM#1#Mult となり，図4と図5では，同一のインスタンスを参照するオブジェクトは，表1の規則に基づいて“this##Base”に統一されている．次に，PDG 作成部に移り，図4と図5のソースコードの PDG を作成する．そして，引数・戻り値解決ノードを削除する．その際，図2の関係になるノードはデータ依存関係のエッジで結線する．それを行うと図6と図7のようになる．ここで，図6と図7におけるノードの識別子は，図4と図5のコードの左側に記してある．識別子が T で始まるものは，引数・戻り値関係解決ノードである．

図6と7の PDG に対して，グラフの一致判定を行う．ここでは，図6に示す PDG の{DD(N2, N4), DD(N4, CM2(f))}と，図7に示す PDG の{DD(N4, N2), DD(N2, CM2(f))}が異なっている．

そのため，この例の場合は，干渉有りと判断される．

ここで，本手法において解決すべき問題点を述べておく．いま，図8に記述した4つのアドバイスが衝突している場合，現段階ではジョインポイント単位でしか干渉の検出を試みていないため，干渉の原因となっているのはAdvice1〜3だけであるにもかかわらず，Advice4 も干渉に関係あると判定される．この問題に対しては，擬似コードから作成した PDG に対してプログラムスライス[12]を抽出し，複数 PDG 間に現れる同一の名前の変数のスライスを比較することで解決を試みている．どの変数が干渉を引き起こす要因となっているかを判断することで，Advice4 は干渉に関与していないことがわかる．

### 4.2 考察

本章では，提案手法の計算量と関連研究について考察する．

#### 4.2.1 計算量

まず，提案手法の計算量について考察する．

**(1) 組み合わせ作成部における計算**

実行される可能性のあるアドバイスの実行順序の組み合わせの数は，衝突しているアドバイスの数の階乗個となる．ただし，いまのところ衝突するアドバイスの数が5個を超えることは考えにくく，組み合わせの数はそれほど多くはならない．

**(2) 一致判定部における計算**

一般には，グラフ一致の計算量に関して，多項式時間内に解決できないことが知られている．本手法では，PDG の各ノードには，それぞれの命令コードがラベルとして保持されている．したがって，グラフ構造の一致を試みる前に，ノード間のラベルの比較を行うことで実質的な計算を大きく抑えることができる．ラベルの比較に関しては，単なる線形探索の繰り返しとなるため，多項式時間で実行可能である．

#### 4.2.2 関連研究

文献[4]では，織り込みの際にアドバイスによってメソッドが受ける影響の危険性を分類している．彼らは，同一のフィールドに対してメソッド・アドバイスが read/write するかどうかという点から，

表2：文献[4]での分類

| | メソッド | アドバイス |
|---|---|---|
| Orthogonal | なし | |
| Independent | 両方 read，又は，片方のみ write | |
| Observation | write | read |
| Actuation | read | write |
| Interference | write | write |

アスペクトとメソッドの関係を表2のように分類している．これらの分類はどれもメソッドとアドバイスの間の関係だけを表現しており，同一時点に織り込まれるアドバイス間の干渉について述べたものではない．

これに対して，提案手法はメソッドとアドバイス間の関係でなく，複数のアドバイス間での干渉を検出する．例えば，提案手法では，図8においてAdvice1 とAdvice2 の実行順序の違いにより，Advice3 が常に影響を受ける（つまり，干渉がある）かどうかを検出できる．一方，文献[4]の手法では，アドバイスが織り込まれるメソッドにフィールドへのアクセスが無い（つまり，read/write が無い）場合，Advice1 および Advice2 とメソッドの関係は Independent になる．

また，文献[9]では，既存のアスペクト指向を用いて構築されたシステムに，新たにアスペクトを追加するとき，そのアスペクトが既存システムのどの部分に影響を与えるか，ということを織り込み後の Java バイトコードのプログラムスライスを取得することによって実現している．これは，干渉の発生を検出していると考えられる．本手法で検出できる干渉との違いを明確にし，その精度について早急に比較する必要がある．

## 5. まとめ

本論文では，同一のジョインポイントに複数のアスペクトが関連付けられており，そのアスペクトの動作順序によって実行結果に違いが発生する場合を，アスペクトの干渉と定義した．そして，任意の順序でアスペクトを織り込むことで得られるプログラムに対して，そのソースコードを静的に解析し作成したプログラム依存グラフの等価性判定を行い，干渉の有無を自動的に検出する手法を提案した．本手法を用いることによって，アス

ペクトの干渉によって発生する予期せぬエラーを防止することが可能である.

現在，本手法を実現するツールを統合開発環境Eclipse[2]のプラグインとして実装中である．また，今後の課題として，検出の精度に関する評価を考えている.

## 謝辞

本論文を作成するにあたり，有益なご意見を頂きました立命館大学情報理工学部 山本哲男講師，ソフトウェア基礎技術研究室，大森隆行氏，戸子田健祐氏に感謝します.

## 参考文献

[1] G.Kiczales, J.Lamping, A.Mendhekar, C.Maeda, C.V.Lopes, J.-M.Loingtier and J.Irwin , "Aspect-Oriented Programming ", In Proc. European Conference on Object-Oriented Programming (ECOOP '97), LNCS 1241, pp.220-242, 1997

[2] Eclipse Project, http://www.eclipse.org/

[3] AspectJ, http://www.eclipse.org/aspectj/

[4] M.Rinard, A.Salcianu, S.Bugrara, "A Classification System and Analysis for Aspect-Oriented Programs", In Proc. 12th International Symposium on Foundations of Software Engineering (FSE 2004), pp.147-158, 2004

[5] J.Bakker, B.Tekinerdogan, M.Aksit, "Characterization of Early Aspects Approaches", Proc. Early Aspects Workshop at AOSD, 2005

[6] A.Rashid, A.Moreira, J.Araujo, "Modularisation and Composition of Aspectual Requirements", Proc. 2nd international conference on Aspect-Oriented Software Development (AOSD '03), ACM Press pp.11-20, 2003

[7] S.Horwitz, J.Prins, T.Reps, "On the adequacy of program dependence graphs for representing programs", Proc. 15th ACM Symposium on Principles of Programming Languages (POPL '88), pp.146-157, 1988

[8] Rovert E. Filman and Daniel P. Friedman, "Aspect-Oriented Programming Is Quantification and Obliviousness", In Proc. OOPSLA 2000 workshop on Advanced Separation of Concerns, 2000

[9] D.Balzarotti, M.Monga, "Using Program Slicing to Analyze Aspect Oriented Composition", In Proc. Foundations of Aspect-Oriented Languages 2004 (FOAL 2004), 2004

[10] J.Zhao, "Slicing Aspect-Oriented Software", Proc. International Workshop in Program Comprehension (IWPC 2002), 2002

[11] G.Rothermel, M.J.Harrold, "Selecting Regression Tests for Object-Oriented Software", Proc. International Conference on Software Maintenance, IEEE Computer Society Press, pp.14-25, 1994

[12] M.Weiser, "Program Slicing", IEEE Transactions on Software Engineering, Vol.SE-10, No.4, pp.352-357, 1984

[13] 篠塚卓, 鵜林尚靖, 四野見秀明, 玉井哲雄, "契約によるクラスとアスペクト間の影響解析", 日本ソフトウェア科学会 第 22 回大会, 2005

[14] 石尾隆, 楠本真二, 井上克郎, "アスペクト指向プログラムに対するプログラムスライシング", 電子情報通信学会技術研究報告 2002-SS-39, Vol.102, No.617, pp.13-18, 2003

[15] F.Tessier, M.Badri, L.Badri, "A Model-Based Detection of Conflicts Between Crosscutting Concerns : Towards a Formal Approach", Workshop on Aspect-Oriented Software Development (AOSD 2004), 2004