2007－SE－157 (6)
2007／9／27

プログラム依存グラフを用いたアスペクトの干渉検出ツールの実装

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は，関心事を分離するという点で有用である．しかし，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意図した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 を対象とし，同一実行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ツールを提案する．ツールは統合開発環境 eclipse のプラグインとして実装した．このツールを用いることによって，アスペクトの実行順序に起因するエラーの混入を防ぐことができる．

Interference Detection Tool 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 : Aspect-oriented programming has advantages 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 running implementation as a plug-in of eclipse, which automatically detects such interference existing in programs written in AspectJ. It first extracts several aspects conflicting at the same program point, and then examines programs in which the conflicting aspects are woven with the different order of their execution to determine if they have the same behavior. With this tool, developers can easily find possible errors resulting from the detected interference.

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である．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メカニズムを提供している．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になってしまうという問題がある．また，記述した複数のアスペクトが互いに影響を及ぼしあい，プログラムが開発者の意図した通りに動作しないことも起こりうる．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[5, 9, 10, 11, 12, 13]．

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，同一実行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ツールを提案する．このツールでは，まず，アスペクトを任意の順序で結合することで，複数のプログラムを得る．そして，結合した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．このツール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，アスペクトの実行順序に起因する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用いたプログラムの等価性判定手法を説明し，4章で，実装したツールで用いた干渉検出手法について説明する．5章では，実装方法について説明し，問題点に関する考察と，簡単な評価を行う．最後に，6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る．アスペクト指向言語の中で，現在最も幅広く利用されている．

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)と呼ぶ．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

プログラム内のすべてのジョインポイントから，ある条件によって切り出された集合をポイントカット(point cut)と呼ぶ．そして，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ジョインポイントの条件と，それに関連付けるアドバイスを記述する．

アドバイスを記述するさいには，ジョインポイントの処理に対して，アドバイスを実行するタイミングを，直前"before"，直後"after"，ジョインポイント本来の処理を置き換える"around"の中から指定する．aroundアドバイス内のみで使用可能な特別なメソッドとして，proceed()メソッドがある．このメソッドを使用することによって，aroundアドバイスによって置き換えられるジョインポイントの本来の処理を呼び出すことができる．

また，アドバイスを実際に処理が行われる時点に挿入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[7]．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，アドバイスが織り込まれた後の，実際のプログラム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織り込まれること．

干渉：衝突したアドバイスの実行順序の違いによって，プログラム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，Counterクラスの stopButton()メソッドで干渉が発生している．いま，このプログラムの開発者の意図した処理を得るには，UseTime → ResetTimeの

```
class Counter {
 private int time = 0;
 void setTime(int t){time = t;}
 void stopButton(){ //何らかの処理 }
}
aspect UseTime{
 after(Counter c) :
  execution(void Counter.stopButton())
  && this(c) {
   // Counter.time の値を使用
}}
aspect ResetTime{
 after(Counter c) :
  execution(void Counter.sotpButton())
  && this(c) {
   c.setTime(0);
}}
```

図1:干渉の例

順でアドバイス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，ただ単にコンパイルしただけではResetTime → UseTimeの順で実行されるため，開発者の望む処理は得られない．

AspectJで，明示的にアドバイスの実行順序を指定するには“declare precedence : 〈アスペクト名〉;”と記述し，アスペクトの優先度を設定しなければならない．例えば，図1のソースコードの場合には，どちらかのアスペクト内に“declare precedence : ResetTime, UseTime;”と記述することによって，ResetTimeアスペクトの優先度をUseTimeアスペクトよりも高めることができる．afterアドバイスの場合は，優先度の高いアドバイスが，より後に実行されるため，これでUseTime → ResetTimeの順でアドバイスを動作させることが可能になる．図1のように，アスペクトの優先度が記述されていない場合は，衝突したアドバイスの実行順序が，コンパイラによって勝手に決定されてしまう．衝突したアドバイスが，干渉状態にある場合は，ユーザの意図した処理が実行されない可能性がある．

本論文で定義した干渉には，以下の3つの特徴がある．

1つめは，プログラマの予期していないプログラム実行時点で，不意に発生する可能性を持つ点である．AOPを用いてプログラムを開発するさいに，クラス部分とアスペクト部分をそれぞれ別のプログラマが担当したとする．この場合，あるプログラマは，他のプログラマがどのようなコードを書いているのかを知らないため，思わぬジョインポイントでアドバイスが実行されてしまう可能性がある．そのさい，衝突や干渉が発生する可能性もある．また，衝突や干渉はプログラム開発中に発生するだけでなく，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，突然発生することも考えられる．

2つめは，プログラマの記述したソースコードが正しくても，干渉は発生するという点である．個々のアドバイスは正しいコードが記述されているにもかかわらず，コンパイラによって決定されるアドバイスの実行順序によって，プログラマの意図した動作が得られなくなってしまう．つまり，干渉はアドバイス内部を単体でテストしただけでは見つけ出すことはできない．

3つめは，干渉は，まれにしか発生しないと想定される点である．まれにしか発生しないエラーに対して，プログラマが常に注意を払うことは大きな労力を伴ってしまう．記述されたアドバイスの数が増加し，アドバイスが織り込まれるジョインポイントの数も増加した場合には，プログラマの手で干渉を見つけ出すのは困難である．

以上のように，干渉という問題は人の手だけで扱うには厄介な問題である．また，われわれは，干渉は発見しづらいバグになると考えている．よって，ツールによるサポートが必須である．

もしツールのサポートによって，プログラム開発時に干渉を自動的に検出できたとすると，開発者に干渉発生の原因となるアドバイスを容易に知らせることができ，干渉を起こしているアスペクト間の優先度を明示的に記述するよう促すことができる．

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[6]で提唱されている．文献[6]の手法では，プログラム中の任意の2つの命令間に存在する5種

類の関係を考えることによって，プログラムの等価性を証明する．それぞれの関係を，以下で説明する．

(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）である．
- 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される．

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 に値が代入されている．
- 命令 t で，w の値が参照されている．
- 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つ存在する．

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 に値を代入している．
- ある命令 u が存在して，DD(s, u)かつ DD(t, u)である．

2つのプログラムから，以上5つの関係（2つの制御依存関係，2つのデータ依存関係，定義順序関係）を考慮して，それぞれ作成した 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，それらの処理が等価であることは，文献[6]で立証されている．

4. PDG を用いた干渉検出システム

本手法を実現するシステムは，大きく分割して4つの部分から構成されている．以降，それぞれの部分について説明する．

（1）衝突発見部

本手法に対する入力は，解析の対象となるすべてのソースコードである．ソースコードは Java, AspectJ で記述された，コンパイル可能なものとする．

まず，入力されたソースコードからポイントカットの条件を調べ，複数のアドバイスが織り込まれるジョインポイント，つまり衝突を見つけ出す．

（2）コントロールフローグラフ作成・結合部

衝突しているアドバイスが見つかったら，それぞれのアドバイスのコントロールフローグラフ(Control Flow Graph：CFG)を作る．そして，それらの順序を変えて結合し，実際に実行される可能性のあるすべての処理を CFG で表現する．

CFG を結合するさいには，after, before アドバイスの場合と，around アドバイスの場合で，2つの異なる処理を行う必要がある．

1つめは，CFG を結合する場所についてである．after, before アドバイスの場合は，あるアドバイスが実行され，その処理がすべて終了した後に，次の優先度のアドバイスが実行される．そのため，アドバイスの CFG の終わりを表すノードに，次の優先度のアドバイスの CFG の始まりを表すノードを結合すればよい．しかし，around アドバイスの場合は，proceed()メソッドが呼び出される場所で，次の優先度のアドバイスが実行される．そのため，アドバイスの CFG の中から proceed()メソッドを呼ぶ場所を見つけ，そこに次の優先度のアドバイスの CFG を挿入する形で結合する必要がある．

2つめは，引数についてである．引数が参照型変数の場合には，after, before アドバイスも

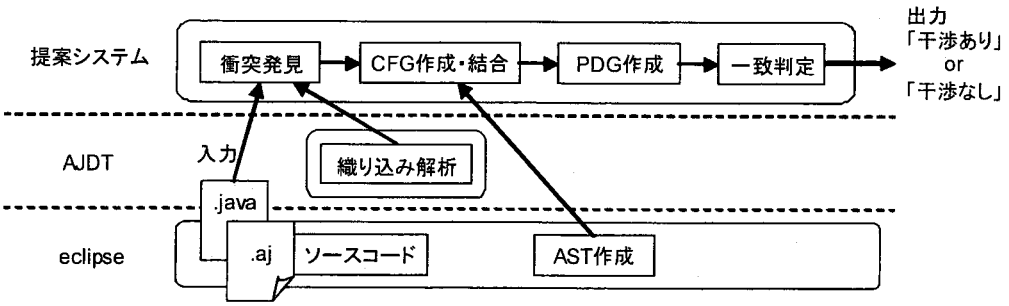

図２：アーキテクチャ

around アドバイスも，先に実行されたアドバイスの影響を，後のアドバイスが受ける．しかし，プリミティブ型変数の場合には，around アドバイスは先に実行されたアドバイスの影響を後のアドバイスが受けるが，after，before アドバイスの場合は，影響を受けないという違いがある．そのため，どの引数と，どの引数を結合後の CFG で同一のものとして扱うかを，引数の型とアドバイスの種類を考慮して判断しなければならない．

最後に，CFG を結合した後に，処理の一致を確かめるさいに不要なノード（個々のアドバイスの CFG の始まり，終わりを表すノードなど）を削除する．

（3）PDG 作成部

結合したそれぞれの CFG から PDG を作る．このとき，３章で説明した５つの依存関係を考慮する必要がある．

本手法は，いわば，アドバイスの実行順序が変わったさいに，プログラムの他の部分に与える影響が変化するかどうかを判断していると言える．そのため，アドバイスの中に出現したすべての変数はプログラムの他の部分で後々使用されると考え，結合後の CFG の終わりを表すノードですべての変数を使用すると仮定し，PDG を作成する．

（4）グラフ一致判定部

最後に，グラフどおしの一致を判定する．本手法の出力は，それぞれの衝突時点について，アドバイスの干渉が「ある」か「ない」かのどちらかである．それぞれの CFG から作成された PDG のうち，１つでも一致しない PDG があれば「干渉あり」となり，すべての PDG が一致すれば「干渉なし」となる．

５．実装・評価

5.1 実装

４章で述べた手法を，統合開発環境 eclipse[2] のプラグインとして実装した．このプラグインのアーキテクチャは，図２のようになっている．

図２のとおり，このプラグインは，eclipse 上で AspectJ プロジェクトを開発するための環境である AJDT(AspectJ Development Tools)[4] の機能と，eclipse の機能を利用する形で作られている．開発に使用した環境を以下に示す．

環境	version
J2SE	1.5.0_12
eclipse	3.2.1
AJDT	1.4.2

以降，４章で述べたそれぞれの部分の実装について説明する．

（1）衝突発見部

AJDT は，AspectJ プロジェクトのコンパイルに成功した時点で，どのアドバイスが，どのジョインポイントに織り込まれるかという情報を保持している．本ツールは，AJDT の内部にある，その情報を解析して衝突を発見する．

（2）コントロールフローグラフ作成・結合部

まず，アドバイス内部の CFG を作成する部分について述べる．この部分は eclipse の AST (Abstract Syntax Tree)作成機能を利用する．１つのノードを作るさいに，ノードを一意に識別できるように ID を付与しておく．

また，現状ではアドバイス内部のみを解析範囲としている．つまり，解析範囲にメソッド呼び出

```
public class Sample{
 boolean isEqual(Sample target){
  if(this == target){
   return true;
  }
  return false;
}}
aspect A {
 before(Sample self) :
 execution(boolean Sample.isEqual(Sample))
  && this(self){
   // 何らかの処理
}}
aspect B {
 before(Sample target) :
 execution(boolean Sample.isEqual(Sample))
  && args(target){
  // 何らかの処理
}}
```

図３：引数の意味が異なる例

しがある場合は，メソッドの呼び出し先までは解析していない．メソッド呼び出しの実引数がプリミティブ型なら変数が使用されるとみなし，参照型なら使用，定義されるとみなす．また，クラスメソッドが呼ばれている場合も同様である．そして，そのオブジェクト自体も使用，定義されるとみなす．

次に，CFG を結合する部分について述べる．４（２）で述べた処理において，アドバイスの引数の意味は，ポイントカットの条件によって決定される．そのため，どの引数と，どの引数が結合後の CFG で意味的に同一の変数になるのかは，引数部分とポイントカット部分の２つを総合して判断する必要がある．現状の実装では，この部分に問題を抱えている．この点については，5.2.1 節で詳説する．

（３）PDG 作成部

（２）で作成したすべての CFG に対して，４（３）で述べた処理を行う．

CFG のノードを作成するさいに付与したノード ID は，対応する PDG のノードにも同じ番号を付与する．

（４）一致判定部

ここでは，PDG の一致を検査する．

まず，すべての PDG の中から順番に２つを選び出す．PDG 内の，各々の PDG のノードに付与されているノード ID は，（２）で CFG を作成した段階から引き継がれている．よって，ノード ID を用いることによって，比較する２つの PDG の中から，もともと同一の CFG のノードだった，一組のノードを探し出すことは容易である．対応するノードの，すべての依存関係が一致した場合は，そのノードが一致するとみなす．すべてのノードが一致すれば，２つの PDG は一致しているとみなす．

これをすべての PDG に対して行い，すべての PDG が一致していると判定されれば，「干渉なし」となり，１つでも一致しない PDG があれば「干渉あり」と出力する．

5.2 問題点

ここでは，本論文で提案したツールの，現状での問題点を述べる．

5.2.1 AJDT に起因するもの

当初，CFG 作成部分は AJDT 内の AspectJ ソースコードの AST を作成する部分を使用して実装する予定だった．しかしながら，現在の実装では，eclipse 内の Java ソースコード用の AST を作る部分を替わりに用いている．その影響で，AST を作った段階で，ポイントカットに関する情報が消えてしまう．よって，アドバイスの引数に関する解析が不完全なものになっており，衝突したアドバイスの引数の並び方，意味に違いが有る場合には，干渉の有無の判断を誤る可能性がある．

例えば，図３では，A 内のアドバイスも B 内のアドバイスも，両方とも引数は Sample 型のものを１つ取る．しかし，A 側の引数は，execution ポイントカットの実行主体を意味しているが，B 側の引数は，Sample.isEqual(Sample) メソッドの引数を意味しており，その意味は全く異なっている．ポイントカットの情報が消えてしまうため，この２つの引数が別のものを意味していることを判断することができない．

5.2.2 干渉検出手法に起因するもの

本手法を用いると図４の例は「干渉あり」と判定される．干渉の定義によれば，アドバイスが内

```
public class Base{
 Result calcResult(Input in){
  // 時間のかかる処理
}}
public aspect Cache {
 HashMap<Input, Result> holder;
 Result around(Input in) :
  execution(Result Base.calcResult(Input))
  && args(in){
   if(holder.containsKey(in)){
    return holder.get(in);
   }
   else{
    Result res = proceed(in);
    holder.put(in, res);
    return res;
}}}
public aspect ExtendBase{
 Result around(Input in) :
  execution(Result Base.calcResult(Input))
  && args(in){
   Result res = proceed(in);
   // ret を使用・定義
   return res;
}}
```

図4：判定を誤る例

部的に「干渉あり」という判定は正しいが，実行順序が入れ替わっても外部的な動作に影響を与えない．これらの違いは，アドバイス内部のみで使用される変数に関するデータ依存関係が異なっていることに起因する．以下に例を用いて説明する．

図4は，Base クラスの calcResult メソッドの処理に時間がかかるため，Cache アスペクトによってキャッシュ機能を追加した場面を想定している．その Cache アスペクトと，もともと Base クラスの機能を拡張していた ExtendBase アスペクトのアドバイスが衝突している．

このケースは，Cache アスペクトがデータの加工を行わないため，プログラム全体で考えたときの，アドバイスの実行順序による外部的動作の違いはない．しかし，内部のデータ依存関係は異なっているため，「干渉あり」という出力になる．これは，ツールのユーザにとっての直感に合致しない結果かもしれない．

この点を改善するには，「外部に影響を与える干渉（ユーザが望むもの）」と，「外部的動作に影響を与えない干渉（ユーザの望まないもの）」を区別する必要がある．

なお，この図4のようなケースの逆で，実際には「干渉あり」なものを「干渉なし」と判定してしまうケースは，3章で述べた方法を利用してプログラムの等価性を判定しているため，発生しない．

表1：実験環境

環境	情報
OS	WindowsXP
CPU	Pentium 4, 3.2GHz
JavaVM メモリ量	1024MB

表2：実験内容

ケース	内容
a	2個のアドバイスの衝突が1箇所
b	2個のアドバイスの衝突が10箇所
c	6個のアドバイスの衝突が1箇所
d	6個のアドバイスの衝突が10箇所

表3：実験結果（複雑度低）

ケース	処理時間（単位：ミリ秒）
a	125
b	953
c	2344
d	23547

表4：実験結果（複雑度高）

ケース	処理時間（単位：ミリ秒）
a	203
b	1657
c	約21分
d	約215分

5.3 評価

ここでは，実装したツールの実行可能性についての評価を行う．評価に使用した環境を表1に，実験の内容を表2にそれぞれ示す．

本手法では，around アドバイスが衝突している，かつ，「干渉なし」と判定されるケースが，最も処理時間とメモリリソースを要するため，実験はすべてそのケースを対象に行った．また，アドバイス内部の CFG，PDG の複雑度による影響を示す

ために，複雑度を変えて実験を行った．それぞれのケースで，本ツールを実行した結果は表３，表４の通りである．複雑度低は，アドバイスの処理が３行程度，複雑度高は25行程度である．

本手法では，起こりうるすべてのアドバイスの組み合わせに対して，処理の等価性を判定するため，$O(n!)$の計算量が必要である．

実験結果を見ると，２～３個のアドバイスが衝突するジョインポイントのみが数多く存在するので場合であれば，それほど大きな待ち時間がなく干渉の有無を判定することができる．また，６個のアドバイスが衝突しているケースですら，１つ21分程度で判定を終えることがわかる．実際には，６個を超えるアドバイスが衝突することは，極めてまれであるため，実用性に問題はないと考えている．

６．おわりに

本論文では，同一のジョインポイントに複数のアドバイスが関連付けられている状態を衝突と定義し，それらのアドバイスの実行順序によって，実行結果に違いが生じる場合を干渉と定義した．そして，任意の順序でアドバイスの処理を結合して得られるプログラムを静的に解析し，プログラム依存グラフの等価性を判定することによって干渉の有無を自動的に判定するツールを提案した．このツールを用いることによって，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．

今後の課題として，文献[10]の手法との比較を考えている．本手法との大きな違いとしては，本手法がアドバイスの織り込まれるすべての順序を考慮しているのに対して，文献[10]の手法は，１つの順序しか考慮しないことによって，処理速度を向上している点である．

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] AJDT, http://www.eclipse.org/ajdt/

[5] 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.

[6] 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.

[7] 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.

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

[9] G.Kniesel, U.Bardey, “An Analysis of the Correctness and Completeness of Aspect Weaving”, Proceedings of Working Conference on Reverse Engineering (WCRE 2006), pp.324-333.

[10] M.Stoerzer, R.Sterr, F.Forster, “Detecting Precedence-Related Advice Interference”, Automated Software Engineering (ASE 2006), 2006.

[11] A.C.D’Ursi, L.Cavallaro, M.Monga, “On bytecode slicing and AspectJ interferences”, International Workshop on Foundations of Aspect Oriented Language (FOAL 2007), March 13, 2007.

[12] C.Koppen, M.Stoerzer, “PCDiff : Attacking the Fragile Pointcut Problem”, 1st European Interactive Workshop on Aspects in Software (EIWAS ’04), 2004.

[13] 平井孝，丸山勝久，“プログラム依存グラフを用いたアスペクトの干渉検出”，情報処理学会第 153 回ソフトウェア工学研究会，SIGSE vol.153, pp7-14.