情報処理学会論文誌　Vol. 50　No. 12　3108–3126 (Dec. 2009)

プログラム依存グラフの等価性に基づくアスペクトの干渉検出

丸山勝久[†1]　平井　孝[†2,*1]

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．しかしながら，複数のアスペクトが互いに干渉することで，開発者が期待していた実行結果が得られないことがある．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を自動的に検出する手法とその実装ツールを提案する．本手法では，実行順序を変えて結合したアスペクト内部のアドバイスに対して，それらのプログラム依存グラフの同形比較を行う．グラフに差異が存在する場合を干渉と判定し，それを開発者に通知する．さらに，本論文では，提案ツールの実行時間に関する実験結果と実プロジェクトに適用した実験結果を示すことにより，ツールの実用性を議論する．

Detecting Aspect Interferences based on the Equivalence of Program Dependence Graphs

Katsuhisa Maruyama[†1] and Takashi Hirai[†2,*1]

Aspect-oriented programming supports the idea of separation of concerns. However, woven aspects sometimes cause unexpected side-effects since they might interfere with each other. The mechanism described here is capable of detecting interferences between aspects that will be executed at the same point in an AspectJ program. This mechanism generates programs derived from any combination of such conflicting aspects. If some of the dependence graphs of the generated programs are not isomorphic, interferences could be detected. An automated tool based on the proposed mechanism can provide a programmer with information on the detected interference. The paper also shows experimental results with source code in actual software development projects, and demonstrates its resulting observations on the effectiveness of the tool.

1. はじめに

アスペクト指向プログラミング（Aspect Oriented Programming: AOP）では，関心事の分離を実現するために，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[1)]．アスペクトを導入することで，オブジェクト指向プログラミングにおいてクラス間にまたがる関心事を，クラスから分離してモジュール内に閉じ込めることが可能である．その反面，アスペクトの定義を見なければ，クラスの振舞いを正確に把握することはできないという性質（obliviousness[2)]）により，互いに影響を及ぼしあう複数のアスペクトを，開発者が無意識に記述してしまうことがある[3)]．

本論文では，文献 3) で提唱されたアスペクトの相互作用問題（aspect interaction problem）において，同一実行時点に織り込まれる複数のアスペクトの実行順序により，プログラム全体の実行結果に違いが発生する場面を干渉と呼ぶ．そのうえで，アスペクト指向言語 AspectJ[4),5)] を対象とし，アスペクトの干渉を自動的に検出する手法とその手法に基づくツールを提案する．提案手法は，まず，アスペクトを任意の順序で結合することで複数のコードを生成する．次に，それぞれのコードから作成したプログラム依存グラフ（Program Dependence Graph: PDG）[6)] に対して同形比較を行い，それらの差異を検査する．2 つの PDG が同形の場合，同一の入力に対して出力（実行結果）が等価であると近似的に判定できる[7)] ので，任意の PDG の組合せに関して差異が存在しない場合にアスペクトは非干渉であると判定する．比較した PDG に差異が存在する場合，実行結果がアスペクトの実行順序に依存して変化する．この場合，アスペクト間に干渉が存在することを開発者（ツール利用者）に警告する．

さらに，本論文では，提案ツールの実行時間に関する評価実験の結果と実プロジェクトに適用した際の検出結果を示すことにより，ツールの実用性を議論する．今回の実験で，4 個までのアドバイスが衝突する場面において，干渉検出が最大でも 2.5 秒程度で完了することを確認した．さらに，2 つの実プロジェクトのソースコードに対して提案ツールを適用した結果，2 秒程度で正しく干渉を検出できることを確認した．これらの結果より，提案ツール

†1 立命館大学情報理工学部
Department of Computer Science, Ritsumeikan University
†2 立命館大学大学院理工学研究科
Graduate School of Science and Engineering, Ritsumeikan University
*1 現在，株式会社野村総合研究所
Presently with Nomura Research Institute, Ltd.

を実際の開発に導入できる可能性は高い．これにより，開発者が予期していなかった干渉を容易に発見することができ，アスペクトの実行順序に起因する副作用によるエラーの混入を防ぐことが期待できる．

以降，2 章では，本論文で扱うアスペクトの干渉について述べる．3 章では，プログラムの振舞いの等価性に基づく干渉検出手法とその手法を実現した干渉検出ツールの概要について説明する．4 章では，干渉検出ツールの実装について詳細を述べる．5 章では実装ツールを用いて行った評価実験の結果を示し，ツールの実用性に関して議論する．6 章では，アスペクトの干渉を扱う関連研究を紹介し，提案手法と比較する．最後に，7 章でまとめと今後の課題を述べる．

2. アスペクトの干渉

AOP において，アスペクトは織り込み（weaving）により既存のクラスと合成される．AspectJ では，織り込みの手段として，プログラムの静的な構造に作用するインタータイプ定義とプログラムの動的な振舞いに作用するポイントカット・アドバイス定義を用意している[5]．静的な構造への作用とは，既存のクラス，インタフェース，アスペクトに対して，新たにメンバ（メソッド，コンストラクタ，フィールド）を追加したり，継承関係や例外検査を緩和したりすることを指す．動的な振舞いへの作用とは，プログラムの実行時に発生するメソッド呼び出し，フィールドアクセス，インスタンス生成などに対して，新たな処理を割り込ませることを指す．

本論文で提案する手法では，静的な構造への作用を適用した後のソースコードに関して，動的な振舞いへの作用による干渉を検出する．ここで，本論文で定義する干渉を理解するため，動的な織り込みの仕組みを簡単に説明する．AspectJ では，プログラム実行における特定の位置をジョインポイント（join point），特定のジョインポイントの集合を選択する条件をポイントカット（pointcut）と呼ぶ．また，ポイントカットに関連付けられる処理をアドバイス（advice）と呼ぶ．アドバイスを記述する際には，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（before，after，after returning，after throwing，around）を指定する．さらに，around アドバイス内のみで呼び出し可能な特別なメソッドとして，`proceed()` が用意されている．このメソッドを呼び出すことにより，around アドバイスに置換された元のメソッドの処理や未実行のアドバイスの処理を実行することができる．

いま，同一の実行時点（同一ジョインポイント，かつ，同一アドバイス実行タイミング）に複数のアドバイスが織り込まれる場合，アドバイスの実行順序はそれらの優先順位により決定される．before アドバイスの場合，優先順位の高い方から順番にアドバイスの処理が実行され，最後にジョインポイントの処理が実行される．反対に，after，after returning，after throwing アドバイス（以降，これらをまとめて after 系アドバイスと呼ぶ）の場合，最初にジョインポイントの処理が実行され，優先順位の低い方から順番にアドバイスの処理が実行される．around アドバイスの場合，まず最も優先順位の高いアドバイスの処理が開始される．アドバイスの本体内部に `proceed()` 呼び出しが記述されていると，その場所で実行中のアドバイスに対して次の優先順位を持つアドバイスの実行が開始される．最も優先順位の低い around アドバイス内部に `proceed()` 呼び出しが記述されていた場合，置換されたジョインポイントの処理が実行される．優先順位の高い around アドバイスに `proceed()` 呼び出しが記述されていなければ，その優先度より低いアドバイスやジョインポイントの処理は実行されない．

ここで，AspectJ では，declare precedence 句を用いることで，異なるアスペクトに記述されたアドバイスに対して優先順位を指定することができる．逆に，優先順位を指定しないことも可能である．優先順位が一意に決定できず，さらに，開発者が優先順位を明示的に指定していないアドバイスの実行順序は織り込みを処理するコンパイラに依存し，その場合アドバイスの実行順序は開発者から見て不定となる[8]．

本論文では，アドバイスの実行順序に起因する副作用に関して，衝突（conflict）と干渉（interference）を次のように定義する[9]．

衝突　同一実行時点（同一ジョインポイント，かつ，同一アドバイス実行タイミング）に対して，複数のアドバイスが記述されていること

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

ここで，AspectJ では，1 つのアスペクトに複数のアドバイスを記述することが可能である．よって，上記の衝突および干渉は異なるアスペクト間だけでなく，同一のアスペクト内部で発生することもある．そこで，本論文では，より正確な表現として，アスペクトの衝突および干渉を，アドバイスの衝突および干渉と呼ぶことにする．**図 1** に本論文で扱うアドバイスの衝突と干渉の概念を示す．

次に，アドバイスが干渉する例を示す．**図 2** のソースコードは，商品購入時のレシートの各品目（`calcTotal` メソッドの引数 `List l` の要素）の金額を合計した値（`total`）に消費税（5%）を乗じたものを，画面に表示するプログラムである．このプログラムでは，`Account` クラス内の `calcTotal()` メソッドの戻り時点において，`Print` アスペクト内部の

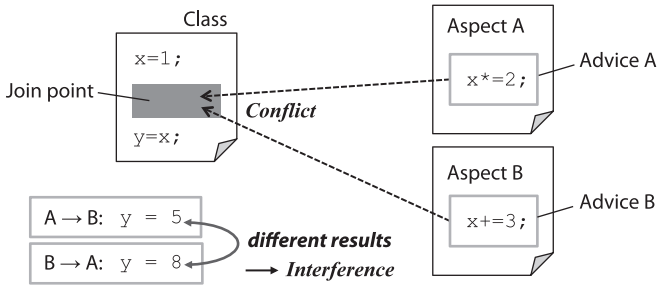


図 1　アドバイスの衝突と干渉
Fig. 1　Conflict and interference between advices.

```
public class Account {
  private int total;
  public void setTotal(int t) { total = t; }
  public int getTotal() { return total; }
  public void calcTotal(List l) {
    total = 0;
    for (Iterator it = l.iterator(); it.hasNext(); ) {
      total = total + (Integer)it.next();
    }
}
public aspect Tax {
  after(Account ac) :
    execution(public void Account.calcTotal(List)) && this(ac) {
     double t = ac.getTotal() * 1.05;
     ac.setTotal((int)t);
   }
}
public aspect Print {
  // declare precedence: Print, Tax;
  after(Account acc) :
   execution(public void Account.calc*(..)) && this(acc) {
    System.out.println(acc.getTotal());
  }
}
```

図 2　アドバイスが干渉する例
Fig. 2　Example of an interference between advices.

アドバイスと Tax アスペクト内部のアドバイスで干渉が発生している．いま，アドバイスが織り込まれたプログラムが開発者の期待する結果を出力するためには，まず Tax に記述されたアドバイスが実行され，その後 Print に記述されたアドバイスが実行される必要がある．しかしながら，コンパイラによっては，Print のアドバイスが Tax のアドバイスより先に実行されることがあり[⋆1]，その場合には開発者の期待どおりの実行結果は得られない．

もし仮にアドバイスの実行順序が変わったとしても，実行結果に違いがなければ，開発者は衝突を意識する必要はない．つまり，衝突するアドバイスに対して，それらが干渉するかどうかを容易に知ることができれば，アドバイスの実行順序に起因する副作用を開発者が意識する必要はなくなり，さらに，予期していなかった副作用によるエラー発生時に開発者が検査しなければならないジョインポイントを減少させることができる．特に，規模の大きなプログラムにおいて，複数の開発者がクラスやアスペクトを別々に記述した場合，それぞれの開発者が想定しなかったジョインポイントでアドバイスが衝突する可能性がある．また，ポイントカットの記述（ワイルドカードの使用）によっては，リファクタリングの際にアドバイスの衝突が突然発生することもある．このような衝突に対して，干渉の可能性をつねに意識してクラスやアスペクトを記述することは開発者にとって非常に面倒である．さらに，実際に干渉の可能性を検査するためには，クラスやアスペクトに記述されたコードの詳細を調査する必要があり，開発者にとって，干渉の検査を手動で行う負担は大きい[8]．

本論文で提案する手法およびツールを用いることでアドバイスの干渉が検出できた場合，干渉するアドバイスを含むアスペクトに優先順位を明示的に指定することで，コンパイラに依存せずに，開発者の期待する実行結果を得るようにアドバイスの実行順序を決定することが可能となる[⋆2]．たとえば，以下の記述

```
declare precedence:  Print, Tax;
```

を Print アスペクト内に追加することで，Print の優先順位は Tax の優先順位よりも高くなる．この結果，Print 内部のアドバイスの実行は Tax 内部のアドバイスの実行よりも必ず後ろになり，コンパイラに依存したエラーが発生することはない．

3. アドバイスの干渉検出手法

本論文では，プログラムの振舞いの等価性に基づき，2 章で定義したアドバイスの干渉を検出する手法を提案する．まず 3.1 節で提案手法の概要を示す．次に，3.2 節で提案手法を実装した干渉検出ツールの概要を述べ，その後 3.3 節でツールの動作を説明する．

⋆1 我々の調査では，複数の AspectJ コンパイラ（ajc 1.5.4，ajc 1.0.6，abc 1.2.1[10]）において，衝突する Print と Tax のアドバイスの実行順序が一意にならないことを確認した．
⋆2 AspectJ では，同一のアスペクトに記述されたアドバイスに対して優先順位を指定できない．この場合，それぞれのアドバイスを別々のアスペクトに分離することで優先順位を指定することが可能となる．

3.1　振舞いの等価性に基づく干渉の検出

いま，プログラムに複数のアドバイスが織り込まれる状況を考える．アドバイスが織り込まれたプログラム全体の振舞いが，織り込まれたアドバイスの実行順序に依存せず等価である場合，プログラムの実行結果はアドバイスの衝突の有無に関係なく必ず等価になる．つまり，干渉は決して起こらない．反対に，もしアドバイスの実行順序を変えて織り込みを適用した複数のプログラムに対して，それらの振舞いが異なる場合，それぞれのプログラムの実行結果が等価にならないことがある．この場合，織り込まれたアドバイスは干渉しているといえる．

ここで，プログラムの振舞いが等価であるとは，同一の入力データ集合に対して，プログラムの実行後に同一の出力データ集合が得られることを指す．このように定義した振舞いの等価性は，プログラムのソースコードから作成したプログラム依存グラフ（以下，PDG と呼ぶ）の等価性（一致あるいは不一致）で判定可能なことが保証されている[7),11)]．特に，文献 11) では，複数の変更を同一のプログラムに対して適用した（たとえば，UNIX diff & patch による差分合成の）際に，プログラムの振舞いが変更の適用順序に依存する（それぞれの差分を独立して合成できない）場合を干渉（interference），依存しない場合（それぞれの差分を独立して合成できる）場合を非干渉（non-interference）と定義している．本論文で提案する干渉検出手法では，アスペクトの織り込み（動的な振舞いへの作用）をプログラム変更ととらえ，アドバイスの干渉問題を文献 11) におけるプログラム変更の干渉問題で再定義したものである．**図 3** に，本論文で提案する干渉検出手法の概要を示す．図 3 において，CFG とは制御フローグラフ（Control Flow Graph）[12)]（以下，CFG と呼ぶ）を指す．

PDG の等価性に基づく干渉検出に関して注意しなければならない点として，PDG が一致しなくても等価な振舞いを持つ複数のプログラムが存在することがある．つまり，PDG が異なるからといって，必ず干渉が発生するわけではない．よって，提案手法は，アドバイスが非干渉であることを検出可能であるが，アドバイス間で干渉が必ず発生することを検出することはできない．

また，本提案手法では，文献 7) における干渉の定義に基づき，対象とするプログラムのデータ依存関係と制御依存関係をすべて正しく表現した PDG が構築可能であることを前提としている．しかしながら，本論文で対象としている Java 言語（AspectJ 言語）には，参照オブジェクト（ポインタ）のエイリアスや多相的なメソッド呼び出しが存在し，実際にはソースコードの静的解析により正確な PDG が構築できるとは限らない．特に，本手法では，プログラム解析における実装上の制約や解析コスト減少のための設計判断により，PDG を

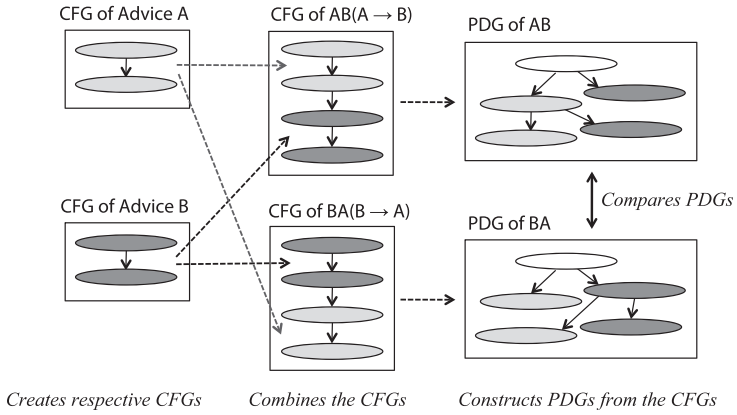


図 3　PDG の等価性に基づくアドバイスの干渉検出
Fig. 3　Advice interference detection based on the equivalence of PDGs.

保守的に構築する．このため，PDG が不正確になることがある．よって，PDG が等価であるにもかかわらず，振舞いの異なるプログラムが存在する．つまり，PDG が等価であるからといって，必ず非干渉といえるわけではない．さらに，複数のスレッドが同時に実行されている場合を考えると，それらの実行順序は静的に作成した PDG では表現できない．リフレクションが存在する場合も同様である．このような場合，PDG の等価性から干渉の有無を正確に判断することはできない．

以上を簡単にまとめると，以下のプログラムに対して，本手法では干渉の誤検出や検出漏れを起こす可能性がある．

- 参照オブジェクトのエイリアスが存在する．
- JDK のクラスを使用している．
- メソッドのオーバライド関係を含む．
- 複数のスレッドで実行される．
- リフレクションを含む．

開発者の負担軽減という観点からは，衝突するアドバイスに対して，それらの間の干渉の可能性を（従来手法に比べて）できるだけ高精度で（誤検出を減らして）検出することを目標としている（検出した干渉の妥当性については，5.4.2 項で議論する）．これにより，開発者は，予期していなかった干渉を発見できたり，干渉の可能性の高い衝突から優先的にエ

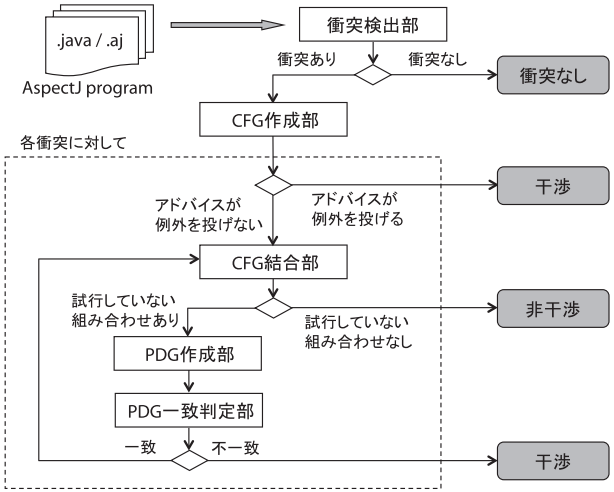


図 4　提案ツールにおける干渉検出手順
Fig. 4　Procedure of interference detection in the proposed tool.

ラーの原因を調査したりすることが可能となる．

3.2　干渉検出ツール

我々は，2 章で述べたアドバイスの干渉を自動的に検出するツールを，統合開発環境 Eclipse [13] のプラグインとして実装した[⋆1]．このツールは，(1) 衝突発見部，(2) CFG 作成部，(3) CFG 結合部，(4) PDG 作成部，(5) PDG 一致判定部，の 5 つのモジュールで構成されている．ツールの入力は，Java および AspectJ で記述されたコンパイル可能なソースコードである．各モジュールを用いた干渉検出手順は，**図 4** のようになる．

まず最初に衝突するアドバイスの発見を試みる．本論文で定義したアドバイスの干渉は，衝突の特別な状況であり，衝突していないアドバイスにおいて干渉が発生することはない．よって，アドバイスの衝突が発見されなかった場合には衝突なしと判定し，検出処理全体を終了する．

衝突するアドバイスが発見された場合，各衝突に対して，衝突アドバイスの CFG の結合処理に移る．ここで，実際に CFG の結合を行う前に，結合するアドバイス内部での例外の扱いを検査する．本ツールでは，コンパイル時にその発生の可能性（アドバイスから投げられる例外が throws 節で明示的に宣言されているかどうか）が検査可能なチェック例外（checked exception）を検査対象とする．コンパイラが検査しない非チェック例外（クラス RuntimeException とクラス Error，および，それらの子孫で定義される例外）は検査対象としない．

いま，アドバイス内部で例外が発生し，かつ，アドバイス内部にその例外を捕捉する文が存在しない場合，アドバイスが例外を外部に投げることになる．このような例外を投げるアドバイスでは，例外の発生により，以降の実行が中止され，プログラムの実行時点がアドバイス外部に移る．よって，プログラム全体の振舞いはアドバイスの実行順序に依存する可能性が高い．本ツールでは，例外処理による制御依存関係の変化を厳密に解析することはせず，文献 8) のアドバイス制御依存の考え方を採用し，アドバイスが例外を投げる場合を干渉と見なすことにする．

結合する CFG のアドバイスが例外を投げない場合は，アドバイスの実行順序を変えて実際に織り込みを適用し，PDG の等価性を検査する．アドバイスの実行順序のすべての組合せに対して PDG が一致する場合，プログラムの実行結果はアドバイスの実行順序に依存しないと考え，非干渉と判定する．不一致の PDG が存在した場合は，実行結果が異なる可能性があるので，干渉と判定する．

ここで，衝突アドバイス本体に，干渉する別のアドバイス群が織り込まれる場合や，衝突アドバイス本体で干渉するアドバイス群が織り込まれたメソッドを利用している場合を考える．このような場合，干渉アドバイス群の影響が干渉検出対象の衝突アドバイスに伝搬する．このような干渉を正確に検出するためには，アドバイスの織り込みやメソッド呼び出しに対する深い影響解析と，影響のあるすべてのアドバイス群の実行順序の組合せに対する振舞いの等価性の検査が必要である．筆者らは，現時点において，このような影響解析と等価性検査を実用ツールに組み込むことは現実的でないと考え，本ツールでは衝突がネストする場合を扱わないこととする．

このような前提に基づき，本ツールにおいて，干渉の検出をアドバイスの衝突ごとに行う理由を示す．本論文で定義するアドバイスの衝突は同一実行時点でしか発生しない．つまり，ジョインポイントが異なるアドバイス間や実行タイミングが異なるアドバイス間には決して衝突が発生しない．よって，干渉する可能性があるすべてのアドバイスは同時に単一の

⋆1 Eclipse 3.2.1 と AJDT 1.5.3（AspectJ Development Tools）[14] を利用している．

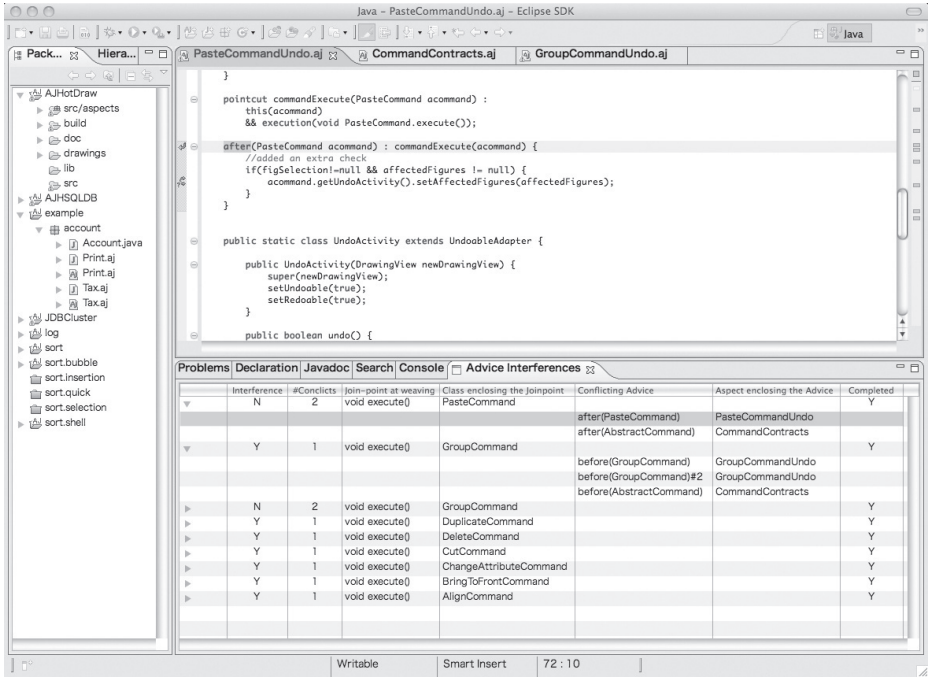


図 5　干渉検出ツールにより出力された判定結果
Fig. 5　Results produced by the interference detection tool.

衝突に割り当てられる*1．さらに，異なる衝突に割り当てられたアドバイスが互いに干渉することはない．このように，干渉は同一の衝突に関連するアドバイス間だけで発生するため，それぞれの衝突に対して独立に干渉を検出すればよい．すべての衝突に対して，各衝突に関連するアドバイスが非干渉と判定される場合，プログラム内部には振舞いが等価かつ独立した衝突だけが含まれることになり，プログラム全体で見ても非干渉といえる．

3.3　ツールの動作

図 5 に，干渉検出ツールの実行の様子を示す．アスペクトの干渉検出を開始するには，統合開発環境 Eclipse の Package Explorer に一覧表示される AspectJ のプロジェクトを右クリックして，メニューから項目 Detect Advice Interferences を選択する．それぞれの衝突に関する干渉の検出結果は，Eclipse のビュー（Advice Interferences ビュー）において，テーブル形式で出力される．

判定直後の出力画面では，テーブルの各行（以降，衝突表示行と呼ぶ）に 1 つの衝突に関する情報が表示される．衝突表示行は，左から 1 番目の欄にあるタブをクリックすることで展開可能となっている．たとえば，図 5 では 9 個の衝突が表示されており，上から 2 つの衝突に関して展開が行われている．衝突が存在しない場合は，1 行目に “No Conflict” とだけ表示される．

衝突表示行の Interference 欄（左から 2 番目の欄）は，各衝突の判定結果が干渉あるいは非干渉（Y/N）であることを指している．干渉かつ衝突アドバイスに優先順位が付与されている場合は，Y (P) と表示される．また，#Conflicts 欄（左から 3 番目の欄）は衝突が発生する実行時点の数を表す．ポイントカット記述にワイルドカードが使用されている場合，同じアドバイスの組が異なる実行時点で衝突することがある．このように，同じアドバイス群が異なる実行時点で衝突している場合には，この数が 1 より大きくなる．さらに，Join-point at weaving 欄と Class enclosing the Joinpoint 欄（それぞれ左から 4 番目と 5 番目の欄）は，衝突が発生しているジョインポイントと，そのジョインポイントが所属するクラスをそれぞれ指している．たとえば，図 5 の最上段の衝突は，`PasteCommand` クラスの `execute()` メソッドへの織り込み時点（after アドバイスのため，2 実行時点）で衝突が発生していることを表している．

衝突表示行を展開することで，各衝突の原因となっているアドバイスに関する情報が表示される．Conflicting Advice 欄と Aspect enclosing the Advice 欄（左から 6 番目と 7 番目の欄）は，衝突の原因となっているアドバイス群と，それらのアドバイスが所属するアスペクトをそれぞれ指している．たとえば，図 5 の最上段の衝突では，`PasteCommandUndo` アスペクトの after アドバイスと `CommandContracts` アスペクトの after アドバイスが衝突している．

Completed 欄（左から 8 番目の欄）は，干渉の判定が完全に終了したかどうかを表している．本ツールでは，CFG の組合せを生成する手段として，以下のモードが選択できるようになっている．

(a) **完全モード**：すべての組合せを生成する．つまり，n 個のアドバイスが衝突している場合，$n!$ 通りの組合せを生成する．

*1 本ツールでは，1 つのアドバイスが複数の実行時点で衝突する場合，そのアドバイスのコピーをそれぞれの衝突に割り当てる．

(b) **検出時間優先モード**：プログラム開発途中では，検出の精度を犠牲にしても，できるだけ迅速に検出結果を知りたい場合がある．このような要求に応えるために，各衝突に対する結合 CFG を 1 秒間でできるだけ多く生成する．ただし，1 秒を超えても，最低 1 組の結合 CFG は生成する．

検出時間優先モードを選択した場合，すべての組合せにおいて結合 CFG が生成されている保証はない．すべての組合せで判定が完了した場合に Completed 欄に Y が，完了していない場合に N が出力される．

干渉判定結果ビューのテーブル中の行をダブルクリックすると，その行に表示されているプログラム要素に対応するエディタ上のコードに，フォーカス（カーソル）を移動させることができる．たとえば，図 5 において，1 行目をダブルクリックすると，衝突が発生しているジョインポイント（衝突の実行時点数が 1 より大きいので，ジョインポイントの 1 つ）にフォーカスが移動する．また，2 行目あるいは 3 行目をダブルクリックすると，衝突の原因となっているアドバイスにフォーカスが移動する．図 5 では，2 行目をダブルクリックしたことにより，`PasteCommandUndo` アスペクトの before アドバイスにフォーカスが移動している．

4. 実　　装

本章では，干渉検出ツールの実装において，各モジュールの詳細を説明する．

4.1 衝突発見部

衝突の発見には，AJDT [14] から提供される情報を利用する．まず，入力されたソースコードの中からアドバイスを見つける．次に，それぞれのアドバイスに対して，それらのポイントカット記述を調べ，織り込まれるジョインポイントと実行タイミングを取得し，記録する．複数のアドバイスが同一実行時点に織り込まれる場合，それらのアドバイスを衝突と見なし，CFG 作成部に処理を移す．

ここで，after アドバイスは，after returning アドバイスと after throwing アドバイスで指定された実行ポイントを合わせたものであり，それ独自の実行タイミングを有しているわけではない．そこで，本ツールでは，1 つの after アドバイスが衝突している場合，after returing と after throwing アドバイスの実行タイミングの 2 つを同時に記録する．

4.2 CFG 作成部

衝突するアドバイスが見つかった後，AJDT が提供する抽象構文木（Abstract Syntax Tree: AST）を利用して，それぞれのアドバイスから CFG を作成する．CFG とは，プログラム中の各命令を 1 つの節点とし，それぞれの命令間の制御フロー（制御の流れ）を矢印（有向辺）で表現したグラフである．本ツールでは，メソッドやアドバイスの開始と終了，メソッドやアドバイスの仮引数，変数宣言文，代入式，メソッド呼び出し式，条件文（**if-else**，**switch**，**while**，**do**，**for**-statement）の条件式，条件分岐の合流，**return** 文，**break** 文，**continue** 文，**throw** 文，**try** 文，**catch** 文が CFG の節点となる．また，これらの文や式が組み合わされている場合，それらを分解して，それぞれ独立した節点とする．たとえば，代入式の右辺にメソッド呼び出し式が含まれる場合，メソッド呼び出し式に対応する節点と代入式に対応する節点がそれぞれ作成される．

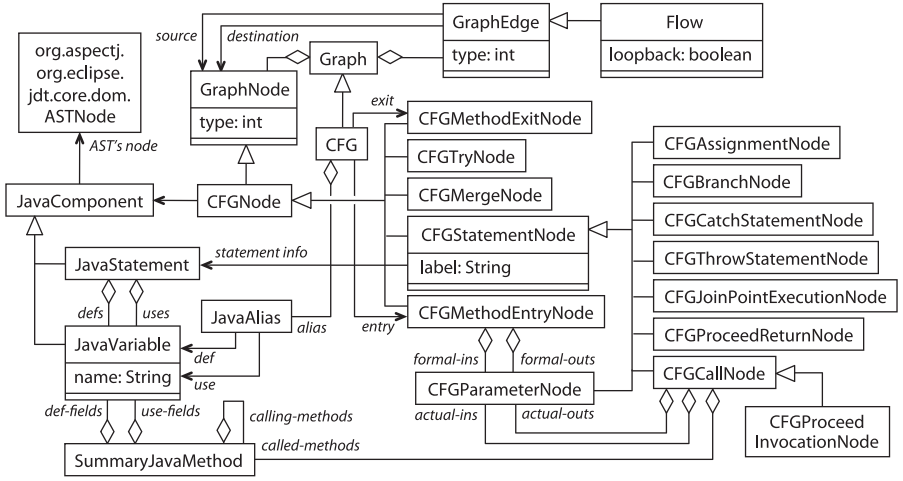


図 6　CFG を実装するクラス群とそれらの関係（一部省略）
Fig. 6　Classes implementing CFGs and their relationships.

4.2.1 CFG の作成手順

図 6 に，本ツールで作成する CFG を実装するクラス群を示す．CFG，CFGNode，Flow は，それぞれ CFG，CFG の節点，CFG の制御フローを実現するクラスである．CFGNode およびその子孫クラスが，それぞれの節点に関する情報を保持する．通常の文や式は，CFGStatementNode あるいはその子孫クラスで表現され，文や式の内容を表現したラベル（文や式に現れる変数名を含む）が付与される．さらに，JavaStatement には，CFG の節点に対応する文や式において値が定義される変数（JavaVariable）の集合（定義変数集合と呼ぶ）と，値が使用される変数の集合（使用変数集合と呼ぶ）が格納される．JavaStatement は

CFGStatementNode から参照可能である．また，ASTNode は，AJDT が提供する AST の各要素を表現するクラスであり，JavaComponent を介して CFGNode から参照可能である．

CFG を作成する主な手順は次のようになる．

(1) インタータイプ定義による静的な構造への作用を適用し，対象となるクラスにメソッドやフィールドを追加する．

(2) 衝突する n 個のアドバイスの CFG A_i（$1 \leq i \leq n$）を作成する．

(2-a) 開始節点（CFGMethodEntryNode）を作成し，A_i に追加する．

(2-b) アドバイスの仮引数節点（CFGParametrNode のインスタンス）と，その節点に関連する制御フロー（Flow のインスタンス）を A_i に追加する．仮引数節点の作成については 4.2.2 項で説明する．

(2-c) アドバイスの本体を解析し，アドバイス内部の文や式に対応する節点（CFGNode およびその子孫のインスタンス）を作成し，A_i に追加する．さらに，作成した節点とその節点に関連する制御フローを A_i に追加する．制御が分岐する際には，条件の真偽（True/False）をそれぞれの制御フローに付与する（GraphEdge の type に格納される矢印の種類で区別する）．また，ループの戻りを担う制御フローに印を付ける（Flow の loopback の値を `true` に設定する）．

(2-d) 終了節点（CFGMethodExitNode）を作成し，その節点に関連する制御フローとともに A_i に追加する．

(2-e) A_i の各節点に対して別名（エイリアス）関係を見つけ，それを A_i が参照している JavaAlias のインスタンスに格納する．

(3) 衝突するアドバイスから呼び出される可能性のある（$A_1, A_2, \ldots, A_n$ に含まれるメソッド呼び出し節点から到達可能な）すべてのメソッド（m 個）を特定し，それらのメソッドの CFG M_j（$1 \leq j \leq m$）を，手順 (2-a)〜(2-e) と同様に作成する．

(4) $M_1, M_2, \ldots, M_m$ に対して，メソッドの要約情報を作成し，SummaryJavaMethod のインスタンスに格納する．同時に，メソッド呼び出し節点（CFGCallNode のインスタンス）と呼び出し先メソッドの要約情報との間にリンクを付与する．メソッドの要約情報については 4.2.3 項で説明する．

(5) $A_1, A_2, \ldots, A_n$ の各節点（正確には CFGStatementNode およびその子孫クラスのインスタンス）に対して，定義変数集合と使用変数集合を求める．これらの集合は，各節点が参照している JavaStatement のインスタンスに格納する．節点がメソッド呼び出しの場合は，呼び出しメソッドの要約情報から定義変数集合と使用変数集合を取得する．

本ツールで採用している別名解析は単純である．非プリミティブ型の変数の値を更新する代入文に対応する節点（CFGAssignmentNode のインスタンス）において，左辺（定義側）の変数名を右辺（使用側）の変数の別名と見なす．CFG の作成が完了した後に，それぞれの節点の定義変数集合および使用変数集合に含まれる変数を調査し，その変数が別名を持つ場合には，それに対応する別名の変数を定義変数集合あるいは使用変数集合に追加する．

また，本ツールでは，JDK 内部のクラスは解析範囲から除くこととし，それ以降のメソッド呼び出しの影響は無視する．これらのクラスのメソッドが呼び出される場合，メソッド呼び出しの実引数がプリミティブ型の変数なら，その命令で使用されていると見なす．もし参照型（非プリミティブ型）の変数であれば，その命令で定義かつ使用されていると見なす．解析範囲外のクラスを型とする仮引数を含むメソッドが呼び出される場合も同様に，その仮引数に対応する実引数に対して，この処理を適用する．

4.2.2　仮引数節点の作成

AspectJ におけるアドバイスの宣言の仮引数の意味は，通常のメソッド宣言の仮引数の記述とは異なり，そのアドバイスのポイントカット条件に記述された this，target，args（ジョインポイントの実行文脈の情報を取り出すポイントカット）の記述により決定される．CFG の作成手順 (2-b) において，これらのポイントカットの引数に現れる識別子と一致する名前を持つアドバイスの仮引数を探し，対応する仮引数節点（CFGParameterNode のインスタンス）の定義変数集合および使用変数集合に，ポイントカットの種類に対応する特殊変数（$this，$target，$args）を追加する．ここで，this はジョインポイント処理の主体側インスタンス，target はジョインポイント処理の対象側インスタンスを指す．また，args はジョインポイント処理の引数（メソッドの引数，フィールドに代入する値，扱われる例外など）を指す．args に関しては，複数の識別子が記述される可能性があるため，仮引数の順番を特殊変数の名前に付与することで区別する．

さらに，after returning アドバイスの場合はジョインポイントの戻り値，after throwing アドバイスの場合はジョインポイントから投げられた例外を捕捉することができる．このため，これらも引数の一種と見なし，対応する引数節点（CFGParameterNode のインスタンス）を作成し，CFG に追加する．

4.2.3　メソッド要約情報の作成

本ツールでは，CFG を作成する際に，衝突するアドバイス内部のコードだけでなく，そのアドバイスから呼び出されるメソッド内部のコードまで解析する．CFG の作成手順 (4) では，以下に示すメソッド要約情報を作成する．

- 要約情報を作成するメソッド M が呼び出すメソッド群に対する要約情報の集合（SummaryJavaMethod の calling-methods で保持）
- M が所属するクラスのフィールドのうち，M で定義されるフィールドの集合と使用されるフィールドの集合（SummaryJavaMethod の def-fields と use-fields で保持）

呼び出しメソッドの集合を正確に取得するには，メソッド呼び出しに関する動的束縛を考慮しなければならない．しかしながら，本ツールでは，参照に関するポインタ解析は行わず，参照の静的な型（宣言上の型，apparent type）と静的なクラス階層（継承関係）に基づき，メソッドの呼び出し先を保守的に決定している．つまり，呼び出し先メソッドとオーバライド関係を持つすべてのメソッドの要約情報を集合に追加する．

メソッド要約情報における定義フィールドと使用フィールドは，要約情報の作成対象メソッド M からメソッド呼び出しを順番にたどることで収集する．実際には，メソッド呼び出しを逆順に戻る際に（メソッド呼び出しに対する深さ優先探索を用いて），呼び出し先メソッドにおいて取得した定義フィールド集合と使用フィールド集合を呼び出し元メソッドの定義変数集合と使用変数集合に追加することで求める．再帰的なメソッド呼び出しが存在した（1 度到達したメソッドに再度到達した）場合は，メソッド呼び出しの探索を中止し，メソッド呼び出しを逆順に戻ることとする．

このようにして呼び出された側のメソッド要約情報を作成し，それを呼び出し側のメソッドの定義変数集合と使用変数集合に反映させることで，衝突するアドバイスから作成される CFG の規模（節点数と矢印数）が，メソッド呼び出しに依存して大きくなることを回避している．

4.3　CFG 結合部

CFG 作成部で作成した CFG を実際に実行順序を変えて結合する．ここで，CFG を結合する際には，衝突したアドバイスの種類によって異なる処理が必要である．**図 7** に CFG 結合の様子を示す．*Xentry* と *Xexit* は，それぞれ CFG の開始節点と終了節点を指す．また，*Xbody* と *Xbody'* は，アドバイス本体に対応する CFG を指す．

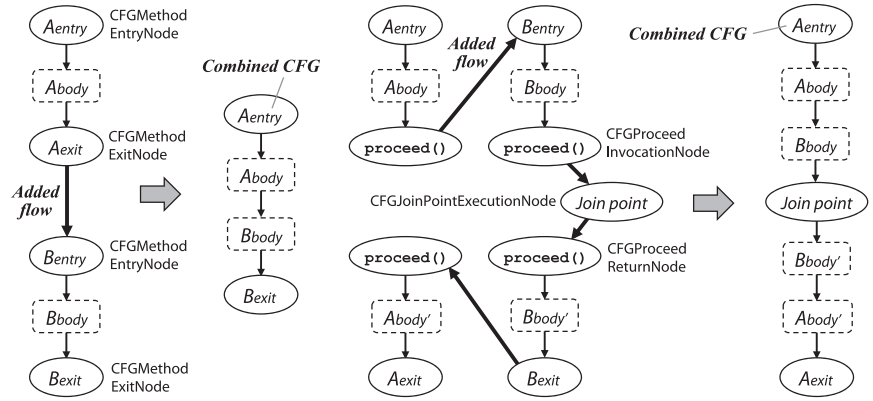
以下，before や after 系のアドバイス，around アドバイスに分けて，CFG を結合する方法を述べる．

4.3.1　before および after 系アドバイスの CFG の結合

before や after 系のアドバイスは戻り値を持たないので，これらのアドバイスがプログラムの他の部分に影響を与える要因は，以下の 2 点である．

(1) アドバイスの引数として渡された参照型インスタンスや，そのインスタンスを介して取得したインスタンスの内部状態の更新

(2) 静的フィールドの値の更新



(a) Combination of CFGs (before/after)　　(b) Combination of CFGs (around)

図 7　CFG の結合
Fig. 7　Combination of CFGs.

引数には，after returning アドバイスの戻り値と，after throwing アドバイスで捕捉された例外を含む．引数としてプリミティブ型の値が渡された場合，その値はもとの値のコピーであるため，アドバイス内でその値を更新してもプログラムの他の部分に影響を与えることはない．よって，before や after 系アドバイスの CFG を結合する際には，上記の 2 つの要因による影響を維持することに注意すればよい．

具体的には，結合する 2 つの CFG の終了節点（CFGMethodExitNode のインスタンス）と開始節点（CFGMethodEntryNode のインスタンス）を制御フロー（Flow のインスタンス）で結合すればよい（図 7 (a) の左側）．ただし，複数の CFG が同一の変数を受け渡す場合には，それらの変数を同じものと見なす必要がある．これを変数の統合と呼ぶ．

本ツールでは，アドバイスの引数を解析することで同一の変数群を見つけ，それらを特殊変数の名前（`$this`，`$target`，`$argsN`，`$returning`，`$throwing`）に置換することで，統合を実現する．`$this`，`$target`，`$argsN` は，それぞれの `this`，`target`，`args` ポイントカットに対応する．`$argsN` の `N` はジョインポイントの処理における引数の順番を指す．また，`$returning` と `$throwing` は，after returning アドバイスの戻り値と after throwing

アドバイスで捕捉された例外を指す．

CFG で受け渡される変数がプリミティブ型の場合は値渡しになるので，変数を統合してはいけない．また，before や after 系アドバイスの前後に実行されるもとのジョインポイントの処理は，衝突するアドバイスの実行順序に依存しない．よって，ジョインポイントの処理に関する CFG を結合する必要はない．

4.3.2　around アドバイスの CFG の結合

次に，around アドバイスの CFG の結合について述べる．around アドバイスも，before や after 系アドバイスと同じ影響をプログラムの他の部分に与える．加えて，before，after 系アドバイスと異なり，around アドバイスは戻り値を持つ．つまり，戻り値を介して，衝突アドバイスがプログラムの他の部分に影響を与えることができる．また，around アドバイスは，本体コード内の `proceed()` 呼び出しの位置でジョインポイントの処理が実行される．このため，アドバイスの引数としてプリミティブ型の値が渡された場合でも，`proceed()` 呼び出しより前に行った値の更新はジョインポイントの処理に影響を与えることになる．さらに，`proceed()` 呼び出しの実引数の意味は，呼び出される around アドバイスの仮引数とそのポイントカット条件によって決定される．このため，実引数の並びによって，引数として渡されるインスタンスの対応関係が変化する可能性がある．以上より，around アドバイスがプログラムの他の部分に影響を与える要因は，以下の 5 点になる．

(1) before や after 系の要因と同じ
(2) before や after 系の要因と同じ
(3) 戻り値の更新
(4) `proceed()` 呼び出し前までのプリミティブ型引数の値の更新
(5) ジョインポイントの処理が実行される際のインスタンスの対応関係

around アドバイスの CFG を結合する際には，上記の 5 つの要因による影響を維持することに注意しなければならない．

具体的には，CFG 作成部で生成する `proceed()` 呼び出しの節点（CFGProceedInvocationNode のインスタンス）と結合対象 CFG の開始節点（CFGMethodEntryNode のインスタンス），結合対象 CFG の終了節点（CFGMethodExitNode のインスタンス）と `proceed()` 呼び出しの戻り値を受け取る節点（CFGProceedReturnNode のインスタンス）を，それぞれ制御フロー（Flow のインスタンス）で結合すればよい（図 7 (b) の左側）．その際，`proceed()` 呼び出しにより最終的に呼び出されるジョインポイントの処理は衝突するアドバイスの実行順序に依存しない．よって，ジョインポイントは 1 つの特殊な節点（CFGJoinPointExecutionNode のインスタンス）で表現するだけで十分である．戻り値の更新については，`proceed()` 呼び出しの戻り値を格納する特殊変数を定義変数集合および使用変数集合に追加することで，呼び出し元のアドバイスに反映させる．また，参照型引数だけでなく，`proceed()` 呼び出し前までに更新されるプリミティブ型引数についても，変数の統合を行う．

ここで，複数の around アドバイスの CFG を結合する場合，`proceed()` 呼び出しの実引数の並びとそれらの意味に注意しなければならない．また，`proceed()` 呼び出しにより，元のジョインポイントの処理が実行される際には，インスタンスの対応関係が維持できているかどうかを後で（PDG の一致判定部で）検査できるように，対応関係を CFGJoinPointExecutionNode のインスタンスに格納しておく必要がある．

4.3.3　不要節点の削除

最後に，結合後の CFG から不要な節点を削除する処理について述べる．実際に CFG を結合する際には，CFG 作成部において，もとのアドバイスから作成した CFG のコピーを用いる．ここで，CFG のコピーはそれぞれ開始節点，終了節点，仮引数を表す節点を持つ．結合後の CFG が，これらの節点を重複して持つ必要はないので，結合後 CFG 全体に対する開始節点，終了節点，仮引数節点だけを残し，その他の開始節点，終了節点，仮引数節点（ただし，プリミティブ型の仮引数に対応する節点は除く）を削除する．この削除は，before，after 系，around アドバイスの結合において適用する．これに加えて，around アドバイスの CFG の結合においては，`proceed()` 呼び出しと戻りに対応する節点も削除する．不要な節点を削除した後の結合 CFG を，図 7 (a)，(b) の右側にそれぞれ示す．

4.4　PDG 作成部

それぞれの統合 CFG から PDG を作成する．PDG は，プログラム中の各命令を 1 つの節点とし，それぞれの命令間の依存関係を矢印（有向辺）で表現したグラフである[6]．**図 8** に，本ツールで作成する PDG を実装するクラス群を示す．PDG，PDGNode，Dependence は，それぞれ PDG，PDG の節点，PDG の依存関係を実現するクラスである．PDG の節点と CFG の節点は 1 対 1 に対応しており，CFG の節点情報（ラベル，定義変数集合，使用変数集合）は PDG の節点から参照可能である．また，CD と DD は，後述する制御依存関係（control dependence）とデータ依存関係（data dependence）を指す．

本ツールでは，プログラムの等価性を PDG の同形比較で判定する．このため，文献 7) に基づき，PDG の矢印（CD あるいは DD のインスタンス）として，次に示す 5 種類の依存関係を CFG から抽出する．

(a) True 制御依存関係：条件分岐における真方向の制御フローを介した制御依存関係

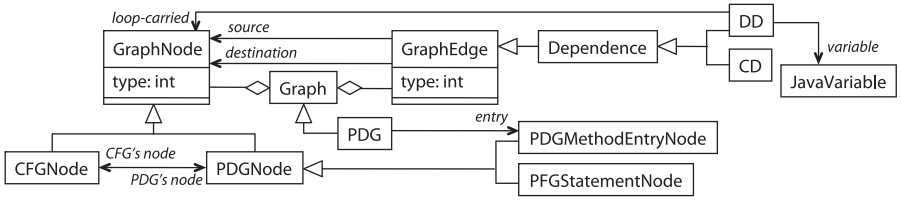


図 8　PDG を実装するクラス群とそれらの関係（一部省略）
Fig. 8　Classes implementing PDGs and their relationships.

(b) False 制御依存関係：条件分岐における偽方向の制御フローを介した制御依存関係
(c) ループ経由（loop carried）データ依存関係：変数の定義と使用が同一のループに含まれ，定義から使用への到達経路がループ命令を含むデータ依存関係．DD の loop-carried にループを担う節点（PDGNode のインスタンス）への参照が格納される．
(d) ループ独立（loop-independent）データ依存関係：変数の定義と使用が同一のループに含まれない，または，定義から使用への到達経路がループ命令を含まないデータ依存関係．DD の loop-carried が `null` になる．
(e) 定義順序（def-order）データ依存関係：同一の変数を定義し，かつ，それらの定義が同一の使用に到達するデータ依存関係

実際に PDG を作成する際には，定義順序依存関係を正確に抽出するために，結合した CFG 内部に現れる変数は，結合 CFG の最後ですべて使用されるものと考える．このため，結合 CFG の最終節点の直前に，その CFG 内のすべての変数を使用する最終使用節点を挿入する．

4.5　PDG 一致判定部

アドバイスの実行順序を変えて作成した複数の PDG に対して，ラベルを考慮した同形写像比較[15] を適用する．具体的には，比較する 2 つの PDG において，一方の PDG の節点と矢印を他方の PDG に写像することで，それらが同形であるかどうかを判定する．いま，PDG G_1 と G_2 が等価であるとは，次の 4 つの条件を満たすことを指す．

(1) G_1 と G_2 が同数の節点と同数の矢印を持つ．
(2) G_1 の節点から G_2 の節点に，1 対 1 かつ上への写像 φ が存在する．
(3) G_1 の各節点 p に対して，p のラベルと G_2 内の節点 $\varphi(p)$ のラベルが等しい．
(4) G_1 の各矢印 $p \to q$ に対して，矢印 $\varphi(p) \to \varphi(q)$ が G_2 に存在し，対応する矢印の依存関係の種類が等しい．

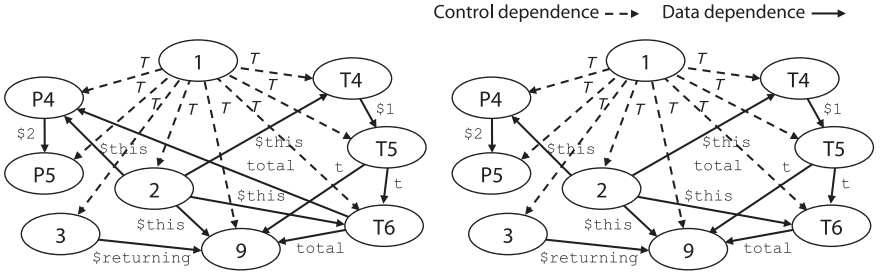
本ツールにおいて，比較する 2 つの PDG は，同一のアドバイスからそれぞれ作成した複数の CFG を順序を変えて結合した結合 CFG から作成したものである．よって，比較する 2 つの PDG の節点の数は等しく，ラベルが等しい節点の組が必ず存在する．

そこで，まず 1 つの PDG 内部にラベル（正確には，ラベルおよび入出力矢印の種類ごとの数）の等しい節点（同一ラベル節点と呼ぶ）が存在するかどうかを調べる．もし，すべての節点がそれぞれ異なるラベルを持つとすると，節点間の写像はすでに一意に決定されていることになる．この場合，各矢印に対して，接続元節点と接続先節点を写像に基づき検査することで，PDG の一致を判定できる．すべての矢印に対して接続元節点および接続先節点が等しい場合，2 つの PDG は一致すると判定される．逆に，接続元節点あるいは接続先節点が異なる矢印が存在する場合，2 つの PDG は一致しないと判定される．ただし，around アドバイスにおいて，その仮引数の並びは `proceed()` 呼び出しの実引数に影響を与えることに注意する．around アドバイスが衝突している場合は，CFG を結合する際に保持しておいた（`proceed()` 呼び出しによりジョインポイントの処理が実行される際の）インスタンスの対応関係に関しても，PDG の一致検査と同時に確認する必要がある．

1 つの PDG の内部に同一ラベル節点が複数存在する場合，同形比較において対応する同一ラベル節点が複数存在することは自明である．そこで，まずラベルに基づき節点間の写像の候補を複数生成する．その後，各写像に対して，上記と同様に矢印の検査を行う．2 つの PDG が一致する写像が 1 つでも存在する場合，それらは一致すると判定される．すべての写像の候補に対して矢印の検査を試したにもかかわらず，2 つの PDG が一致しない場合，それらは一致しないと判定される．

ここで，図 2 のソースコードを用いて，PDG の一致判定を説明する．`Tax` アスペクトのアドバイスと `Print` アスペクトのアドバイスは，`calcTotal()` メソッドの呼び出しの戻りにおける実行タイミング after で衝突している．これらのアドバイスはどちらも this ポイントカットを用いて `calcTotal()` メソッドを実行する主体側インスタンスを取得し，それぞれのアドバイスの引数 `ac` と `acc` に格納している．2 つのアドバイスを順番を変えて結合した CFG から作成した PDG を**図 9** に示す．節点 2 と 3 はアドバイスの引数を指す．また，節点 9 は最終使用節点を指す．引数 `ac` と `acc` の指すインスタンスは同一であるため，特殊変数 $this への置換によって統合されている．2 つの PDG を見ると，データ依存関係（節点 T6 → 節点 P4）に関する差異を含む．よって，2 つのアドバイス `Print` と `Tax` は干渉と正しく判定される．

Label	Defined variable	Used variable
1: (Entry)		
2: Param:$this	$this	
3: Param:$returning	$returning	
T4: $1=$this.getTotal()	$1	Account.total, $this
T5: t=$this.getTototal()*1.05	t	$1
T6: $this.setTotal((int)t)	Account.total	t, $this
P4: $2=$this.getTotal()	$2	Account.total, $this
P5: System.out.println($2)	$3, $2	$2
9: FinalUse		Account.total, t, $this, $returning



(a) Combined with Tax → Print (T4 → T5 → T6 → P4 → P5)　(b) Combined with Print → Tax (P4 → P5 → T4 → T5 → T6)

図 9　図 2 のアドバイスを結合した CFG から生成した PDG
Fig. 9　PDGs derived from CFGs of the advices shown in Fig. 2.

5. 評　　価

本章では，提案ツールの実用性を確認するために行った，以下の 2 点に関する評価実験とその結果ついて述べる．

- 干渉検出処理に要する時間
- 実プロジェクトへ適用した際の判定結果の妥当性

5.1　干渉検出に要する時間

干渉検出時間の測定は，Intel Core 2 Duo（3.06 GHz）を搭載した計算機の Mac OS X（10.5.6）において，Java VM に 1,024 MB のメモリ領域を割り当てて行った．本ツールでは，非干渉と判定される場合に最も検出時間を要するため，各実験において同一のアドバイスを同一の実行時点に複数個織り込む（衝突させる）非干渉のソースコードを用いて行った．また，検出時間に対するアドバイスの複雑度の影響を確認するために，アドバイス内のコード量や扱うデータ構造を変えた 5 種類のアドバイスを用意した．すべての実験において，メモリ容量が不足することはなかった．

衝突するアドバイスの数を 2〜6 個まで変化させた際の，それぞれのアドバイスに関する干渉検出時間を**表 1** に示す．表中に記された時間は，同一の干渉検出実験を 10 回行った際の検出時間の平均（ms）である．表 1 の 1 行目には，それぞれのコード行数（Lines of Code: LOC）を示す．また，#N と#E は，同形比較対象の PDG（結合後の CFG から作成した PDG）の節点数と矢印数をそれぞれ表す．

ここで，本実験で用意したアドバイスに関して，簡単に説明する．Log は，そのコード行数が極端に少ないアドバイス（3〜4 行）である．このようなアドバイスは特定の処理の前後でログを記録したり，メッセージを出力したりすることを目的として，実際のプログラムにおいて数多く記述されている．BubbleSort と ShellSort は，単一の機能を実現する標準的な規模のアドバイス（12〜22 行）として用意した．さらに，（規模の小さい）アドバイス（4〜5 行）が別のメソッド（25 行のコードで実現）を呼び出している例題として，QuickSort を用意した．アドバイスが扱うデータ構造として，配列と JDK のコレクションを用いたアドバイスを用意した（BubbleSort-Ar と ShellSort-Ar は配列を用いたアドバイス，ShellSort-Co と QuickSort-Co はコレクションを用いたアドバイスである）[*1]．これらのアドバイスが実際のソフトウェア開発でそのまま利用されるとは考えにくい．しかしながら，ソートのアルゴリズムは広く知られており，内部コードの複雑さをとらえやすい．このため，アドバイスの内部コードの複雑さと干渉検出時間との関係を示す例題として妥当であると，筆者らは考えた．

表 1 を見ると，before および after アドバイスに比べて，around アドバイスの検出に要する時間は少し長くなっているものの，アドバイスの種類に関する傾向に大きな違いはない．また，扱うデータ構造や別のメソッドを呼び出しているどうかという点が，干渉検出時間に大きな影響を与えているとはいえない．検出時間に大きな影響を与えているのは，衝突するアドバイスの数と規模（PDG の節点数と矢印数）であるようにみえる．

いま，衝突するアドバイスの数が 2〜4 個の部分に着目すると，たとえアドバイスの規模が大きくても，その検出時間は最大で約 2.5 秒（2,456 ms）である．この程度の時間であれば，実際の開発環境において，Java および AspectJ のソースコードのコンパイルと同時に干渉の検出を行っても支障はないといえる．

⋆1 実際には，3 種類のソート BubbleSort，ShellSort，QuickSort に対して，それぞれ配列版とコレクション版を用意して実験を行った．干渉検出時間全体の傾向は変わらなかったため，BubbleSort-Co と QuickSort-Ar の測定結果は省略した．

表 1　干渉検出時間の測定結果（Log，BubbleSort，ShellSort，QuickSort）
Table 1　Experimental results of measurements of detection time.

衝突数	before (LOC = 3)			after (LOC = 3)			around (LOC = 4)		
Log	検出時間	#N	#E	検出時間	#N	#E	検出時間	#N	#E
2	5	4	3	5	5	5	5	5	4
3	5	5	4	6	6	6	7	6	6
4	9	6	5	10	7	7	16	7	9
5	39	7	6	41	8	8	83	8	13
6	626	8	7	630	9	9	966	9	18

衝突数	before (LOC = 12)			after (LOC = 12)			around (LOC = 13)		
BubbleSort-Ar	検出時間	#N	#E	検出時間	#N	#E	検出時間	#N	#E
2	18	25	71	22	26	73	25	26	74
3	85	36	106	98	37	108	103	37	113
4	523	47	141	547	48	143	652	48	155
5	4,544	58	176	4,668	59	178	5,548	59	200
6	49,352	69	211	49,522	70	213	57,682	70	248

衝突数	before (LOC = 19)			after (LOC = 19)			around (LOC = 20)		
ShellSort-Ar	検出時間	#N	#E	検出時間	#N	#E	検出時間	#N	#E
2	34	33	177	35	34	179	37	34	180
3	195	48	265	202	49	267	228	49	273
4	1,420	63	353	1,444	64	355	1,638	64	370
5	12,282	78	441	12,486	79	443	14,083	79	471
6	123,773	93	529	125,284	94	531	134,288	94	576

衝突数	before (LOC = 21)			after (LOC = 21)			around (LOC = 22)		
ShellSort-Co	検出時間	#N	#E	検出時間	#N	#E	検出時間	#N	#E
2	39	45	188	41	46	190	46	46	191
3	258	66	280	249	67	282	302	67	290
4	2,113	87	372	2,122	88	374	2,456	88	395
5	18,989	108	464	19,128	109	466	21,790	109	506
6	198,129	129	568	201,258	130	558	224,828	130	623

衝突数	before (LOC = 4 + 25)			after (LOC = 4 + 25)			around (LOC = 5 + 25)		
QuickSort-Co	検出時間	#N	#E	検出時間	#N	#E	検出時間	#N	#E
2	10	11	23	9	12	25	11	12	27
3	14	15	33	14	16	35	20	16	43
4	39	19	43	40	20	45	73	20	63
5	162	23	53	298	24	55	678	24	87
6	4,375	27	63	4,392	28	65	7,128	28	115

表 2　DB Cluster Framework に対する判定結果
Table 2　Experimental results with DB Cluster Framework.

名前	#CA	#EP	衝突箇所	CA の種類	判定結果 1		判定結果 2	
A	2	39	PrivilegeCheckerImpl	after	非干渉	yes	非干渉	yes
B	2	5	ClusterFactory	after	非干渉	yes	非干渉	yes
C	2	4	TracingAspect	after	非干渉	yes	非干渉	yes
D	2	2	TracingAspect	before	非干渉	yes	非干渉	yes
E	2	2	CBicycle	around	干渉	yes	干渉	yes
F	2	1	CBicycle	around	干渉	yes	干渉	yes

衝突するアドバイスの数が 5 個および 6 個の部分を見ると，検出時間は大幅に増加する．特に，ShellSort-Co では最大 3 分 45 秒（224,828 ms）を要している（予備実験において，衝突するアドバイスの数を 7 とした場合，約 66 分（3,954,563 ms）を要した）．このように長い検出時間は，本ツールを実際の開発環境へ導入する際の障壁になる可能性がある．ただし，現時点で我々が調査した実際のソースコードにおいて，5 個以上のアドバイスが同一実行時点で衝突するものはない．

5.2　実プロジェクトに対する適用

ここでは，本干渉検出ツールを実プロジェクトに対して適用した結果を示す．評価実験に使用するプロジェクトの選択基準を以下に示す．

- Java と AspectJ のみで記述されており．その他の言語は未使用である．
- 100 個以上の Java ファイルと 10 個以上の AspectJ ファイルを含む．

規模の小さいプロジェクトではアドバイスの衝突が起こる可能性が低く，本ツールの評価に不適切であると判断し，比較的規模の大きなプロジェクトを評価実験の対象とした．

これらの基準を満たすソースコードを SourceForge [16] で探したところ，DB Cluster Framework（0.1.2 版）と AJHotDraw（0.4 版）の 2 つのプロジェクトが見つかった[⋆1]．

それぞれのプロジェクトに対して提案ツールを適用した判定結果を**表 2** と**表 3** に示す．説明のために，それぞれの衝突に名前（A〜O）を付与する．#CA は衝突するアドバイスの数を表す．また，#EP は衝突が発生する実行時点の数を指す．ポイントカット記述にワイルドカードが使用されている場合など，同じアドバイスの組が異なる実行時点で衝突する場合に#EP が 1 より大きくなる．衝突箇所は，衝突が発生するジョインポイントを含むク

⋆1 HSQLDB（1.8.0 版）（277 個の Java ファイルと 25 個の AspactJ ファイルで構成）も評価実験対象の候補として見つかったが，アドバイスの衝突が存在しなかった．

表 3　AJHotDraw に対する判定結果
Table 3　Experimental results with AJHotDraw.

名前	#CA	#EP	衝突箇所	CA の種類	判定結果 1		判定結果 2	
G	2	2	PasteCommand	after	非干渉	yes	非干渉	yes
H	3	1	GroupCommand	before	干渉	yes	非干渉	no
I	2	2	GroupCommand	after	非干渉	yes	非干渉	yes
J	3	1	DuplicateCommand	before	干渉	yes	非干渉	no
K	3	1	DeleteCommand	before	干渉	yes	干渉	yes
L	3	1	CutCommand	before	干渉	yes	干渉	yes
M	3	1	ChangeAttributeCommand	before	干渉	yes	干渉	yes
N	3	1	BringToFrontCommand	before	干渉	yes	干渉	yes
O	3	1	AlignCommand	before	干渉	yes	干渉	yes

ラスあるいはアスペクトを指す．CA の種類は衝突するアドバイスの種類を指す．

判定結果 1 は，例外の扱いを検査した場合に干渉検出ツールが出力した結果とその妥当性を表す．判定結果 2 は，例外の扱いを検査せず PDG の一致判定のみを適用することで干渉検出ツールが出力した結果とその妥当性を表す[⋆1]．“yes” は判定結果が正しかったこと，“no” は判定結果が正しくなかったことを指す．

5.2.1　DB Cluster Framework

DB Cluster Framework は，Ajax を用いたシステム開発のためのフレームワークである．128 個の Java ファイルと 28 個の AspectJ ファイルで構成されている．このソースコードに対して本ツールを適用した結果，53 カ所の実行時点で衝突が発見された．干渉検出に要した時間は，10 回の平均で 368 ms（例外を検査）と 392 ms（例外を無視）であった．さらに，判定結果の正しさは 6/6（判定結果 1 と 2）となった．以下，干渉と判定された衝突に関して，簡単に説明する．

衝突 E と F　`ClusterAttribute` アスペクトと `DomainCheck` アスペクトで記述されているアドバイスの衝突である．これらのアドバイスは，例外を投げる可能性があるため，判定結果 1 では干渉と判定された．また，2 つのアドバイスのポイントカット記述には，`target` ポイントカットが使用されており，ジョインポイント処理の対象側インスタンスを取得している．それぞれのアドバイスにおいて，この対象側インスタンスの扱いが異なるため，依存関係に違いが発生し，例外を検査しない判定結果 2 でも干渉と判定された．実際には，`DomainSetterUsage` アスペクト内に `precedence` 節が宣言されており，`DomainCheck` の優先順位が `ClusterAttribute` の優先順位よりも高く設定されている．

5.2.2　AJHotDraw プロジェクト

AJHotDraw とは，2 次元図形を描画するアプリケーション JHotDraw をアスペクト指向リファクタリングにより書き直したものである．290 個の Java ファイルと 31 個の AspectJ ファイルで構成されている．このソースコードに対して本ツールを適用した結果，11 カ所の実行時点で衝突が発見された．干渉検出に要した時間は，10 回平均で 1,799 ms（例外を検査）と 1,885 ms（例外を無視）であった．さらに，判定結果の正しさは 9/9（判定結果 1）と 7/9（判定結果 2）となった．以下，干渉と判定された衝突に関して，簡単に説明する．

衝突 H と J　`GroupCommandUndo` および `DuplicateCommandUndo` アスペクト内に記述されているそれぞれ 2 つのアドバイスと，`CommandContracts` アスペクトに記述されているアドバイスの衝突である．実際には，`GroupCommandUndo` および `DuplicateCommandUndo` に関する片方のアドバイスの中身はすべてコメントアウトされており（空であり），それぞれ 3 つではなく，2 つのアドバイスの衝突と見なすことができる．衝突するアドバイスのコードを見ると，`CommandContracts` 内のアドバイスに例外を投げるコードが記述されていた．このアドバイスが `GroupCommandUndo` あるいは `DuplicateCommandUndo` アスペクト内のアドバイスよりも先に実行され，かつ，例外が投げられた場合，これらのアドバイスは実行されない．つまり，アドバイスの実行順序により振舞いが変化する可能性がある．よって，判定結果 1 は正しい．例外が発生しない場合には，どちらのアドバイスを先に実行しても振舞いは変化しない．よって，判定結果 2 では非干渉と判定されている．本検出ツールでは例外処理に対する制御依存関係解析を行っていないため，このような誤検出が発生する．

衝突 K，L，M，N，O　`DeleteCommandUndo`，`CutCommandUndo`，`ChangeAttributeCommandUndo`，`BringToFrontCommandUndo`，`AlignCommandUndo` アスペクトに記述されている 2 つのアドバイスと，`CommandContracts` アスペクトに記述されている 1 つのアドバイスの衝突である．これらの衝突が干渉と判定される原因はすべて同じである．よって，ここでは，衝突 K を取り上げて説明する．衝突 H および J と同様に，`CommandContracts` 内のアドバイスは例外を投げる可能性がある．これにより，判定結果 1 は干渉となる．さらに，`DeleteCommandUndo` 内の 2 つのアドバイスにおいて，一方は this ポイントカットにより取得したインスタンスの `myUndoableActivity` フィールドを定義し，他方はそのフィールドを使用している．衝突するアドバイスが同じクラス（インスタンス）の同一フィールドをそれぞれ定義および参照しているため，実行順序が異なると依存関係が変化する．これにより，判定結果 2 が干渉となる．

⋆1 ツールのオプションで，アドバイスが例外を投げるかどうかの検査を外すことが可能である．

ここで，myUndoableActivity フィールド，および，このフィールドに対するアクセッサメソッド（getUndoActivity() および setUndoActivity()）はインタータイプ宣言により，AbstractCommand クラスに織り込まれている．AbstractCommand は，DeleteCommandUndo 内のアドバイスが織り込まれる DeleteCommand の祖先クラスである（CutCommandUndo，ChangeAttributeCommandUndo，BringToFrontCommandUndo，AlignCommandUndo アスペクト内のアドバイスが織り込まれるクラスに対しても共通の祖先となっている）．

5.3　ツールの実装における制約と設計判断

本ツールでは，衝突するアドバイス間の依存関係の変化を検査することで，従来の干渉検出手法に比べて，より精度の高い干渉検出を目指している．しかしながら，検出の精度と解析コストは一般的にトレードオフの関係になり，実用性という観点からは，より深い解析を単純に追い求めればよいというわけではない．このような干渉検出ツールを設計する場合，どの程度の検出時間でどの程度の検出の正しさが得られるのかを検討することが重要である．ここでは，本手法において，検出時間と検出精度に影響を与えるいくつかの項目について考察する．

5.3.1　対象とする衝突の種類

AspectJ におけるポイントカット条件の中には，織り込まれるジョインポイントが動的に決定されるものがある．具体的には，制御のフローに関係するポイントカット（cflow や cflowbelow）と，条件に基づくプリミティブポイントカット（if）である．AJDT 内部の実装では，これらのポイントカットを利用しているアドバイスは，可能性のあるすべてのジョインポイントに織り込まれると見なしている．このような静的解析では，実行時に発生しない衝突に対しても干渉と判定してしまう可能性が高く，開発者に無用な警告を与えることになる．

このような事態を避けるため，本ツールではジョインポイントが静的に解析可能なポイントカットのみを現時点で対象としている．これは，制御フローおよび条件に関するポイントカットの扱いが，その他のポイントカット（たとえば，call，execution，set，get）に比べて面倒であり，これらを用いた記述は比較的少ないと考えられるからである．ただし，より実用的なツールの構築という観点において，動的に決定されるジョインポイントの扱いに関しても，今後検討が必要である．

5.3.2　CFG 結合における変数の統合

通常，アドバイスの引数は，アドバイスが織り込まれるジョインポイントの種類によって異なる．本ツールの現在の実装では，衝突したアドバイスの中で，this，target，args ポイントカットに着目し，同一のポイントカットで指定されている変数（インスタンス）を同一であると見なしている．逆に，異なるポイントカットで指定されている変数は異なるインスタンスを格納していると仮定している．しかしながら，execution ポイントカットのように，this と target で指定された変数が同一のインスタンスを格納している場合がある．

本ツールでは，格納するインスタンスが異なる可能性のある変数は統合対象とせず，必ず同一のインスタンスを格納する変数だけを統合している．このため，発生する干渉を見逃すことはないが，干渉に関する無用な警告を数多く提示する可能性がある．これを回避するためには，プリミティブポイントカットをより詳細に解析して，同一のインスタンスを識別する手法を新たに考案する必要がある．引数の統合に関しては，文献 17) におけるアドバイスの統合を取り入れた改良を検討している．

5.3.3　PDG 作成におけるインスタンスの解析

本ツールで作成する PDG は，クラス依存グラフ（Class Dependence Graph: ClDG）[18] であり，同じクラスから生成した複数のインスタンスを区別しない．つまり，インスタンスメソッドの呼び出しやインスタンスフィールドへのアクセスは，クラスメソッドの呼び出しやクラスフィールド（クラス変数）へのアクセスと同様に扱うことになる．このため，衝突するアドバイスが互いに異なるインスタンスのフィールドを独立に定義している，あるいは，独立に定義と使用を行っている場合にも，本ツールでは干渉と判定する．これは誤判定である．このような誤判定を避けるために，インスタンスを区別するように拡張された System Dependence Graph（SDG）[19] の導入を検討している．

さらに，本ツールでは，参照に関するポインタ解析（型解析）を適用せず，動的に束縛される可能性のあるメソッドはすべて呼び出されると仮定して，PDG を作成している．このため，実際には呼び出されないメソッド内部の依存関係の違いにより，干渉しない衝突を干渉と判定する場合がある．また，衝突する複数のアドバイスが，振舞いの異なる（フィールドの値を定義および参照する）多相的な（オーバライド関係を持つ）メソッドをそれぞれ呼び出している場合，メソッドの呼び出し先を厳密に特定しなければ，それぞれのアドバイスに対応する PDG が不必要なデータ依存関係を含むことになる．この場合，実際には振舞いが異なるアドバイス群に対して PDG が等価になり，干渉の可能性を見逃すことになる．このような問題を軽減するために，プログラムのフローを考慮した別名解析[20] をはじめとするポインタ解析技術の導入を検討している．

5.3.4　リフレクションへの対応

今回の評価実験において，衝突アドバイス内にジョインポイントの文脈情報を表す thisJoinPoint インスタンスを利用したリフレクションが記述されていた（DBClusterFramework の

衝突 E と F）．このようなリフレクションを用いたコードは衝突アドバイスにも現れており，AspectJ では一般的な記述になりつつあるといえる．よって，正確に干渉を検出するためには，リフレクション解析が必須である．しかしながら，リフレクションが利用されたコードの解析は一般的に困難である．そこで，リフレクションが利用されている場合は，判定結果が正確でないことを開発者に通知する仕組みを本ツールに組み込む予定である．

5.4　実用性に関する議論

本節では，5.1 と 5.2 節で示した評価実験に関して，干渉検出時間および検出結果の妥当性を議論する．

5.4.1　干渉検出時間に対する考察

ここでは，まず本干渉検出ツールにおける判定アルゴリズムの計算量を示す．いま，ADJT の提供する衝突の数を N_C，各衝突に関わるアドバイスの（最大）数を N_A とする．各アドバイスから CFG を作成する処理，および，各 CFG から PDG を作成する処理は多項式時間で実行可能である．よって，アドバイスの規模（AST の要素数）や CFG の規模（節点数と矢印数）を N_G とすると，計算量は $O({N_G}^c)$（c は定数）と表現できる．次に，CFG の結合を考える．本ツールでは，各衝突に関わるアドバイス群に対して，それらの実行順序を変えた組合せに基づき結合 CFG を作成する．つまり，1 つの衝突に対して，$N_A!$ 個の結合 CFG とその PDG が作成される．また，本ツールでは，$N_A!$ 個の PDG がすべて同形の場合を非干渉と判定する．よって，PDG 一致判定においては，任意の PDG を 1 つ選択し，残りのすべての PDG に対して $(N_A! - 1)$ 回の同形比較を行えばよい．ここで，グラフの同形判定の計算量は，グラフの節点数 N_N に対して $O(N_N!)$ となる．以上より，全体（CFG 作成処理 + PDG 作成処理 + 同形比較処理）の計算量は，次のようになる．

$$O(N_C \times N_G^c \times N_A) + O(N_C \times N_G^c \times N_A!) + O(N_C \times N_A! \times N_N!)$$

この式より，本ツールにおける干渉検出時間は，衝突アドバイスの数 N_A と PDG の同形比較における同一節点の候補数 N_N に大きく影響を受けることが分かる．ただし，4.5 節で述べたように，本ツールでは PDG の同形比較に節点のラベルを活用している．よって，節点数が N_N の PDG どうしの同形比較において，1 つの節点に対応する節点が必ず N_N 個存在するわけではない．たとえば，片方の PDG に同一ラベル節点（ラベルの等しい節点）が重複して存在していない場合，節点の対応は一意に決定できる．節点のラベルを活用することで，判定アルゴリズムのおける $N_N!$ は，同一ラベル節点の数 N_i（$1 \leq i \leq m$）の階乗の積（$N_1! \times N_2! \times \cdots \times N_m!$）に置き換えられる（$m$ は PDG 全体に存在する同一ラベル節点の集合の数を指す）．以上より，本ツールにおける干渉検出時間は，衝突アドバイスの数と PDG に存在する同一ラベル節点の数に支配されているといえる．

これは，表 1 において，衝突アドバイスの数が増加すると干渉検出時間が大幅に増加している事実に一致する．さらに，5.1 節の評価実験では，同一のアドバイスを複数個用意して衝突を実現している．このため，衝突するアドバイスの数に比例してラベルの等しい節点の数が増加し，結果的に干渉検出時間が大幅に増加する．残念ながら，衝突するアドバイスどうしに，どの程度の数の同一ラベル節点が存在するかを本論文で明確に示すことはできない．しかしながら，筆者らは，比較する 2 つの PDG に，ラベルだけが等しい節点が 10 個以上含まれることはあっても，ラベルだけでなく入出力矢印の種類ごとの数まで等しい節点が 10 個以上含まれることはまれであると考えている．筆者らの予備実験では，10 個の同一ラベル節点を含む PDG に対して約 11 秒で同形比較が終了することを確認している．

5.4.2　検出判定結果の妥当性に対する考察

本ツールの干渉検出において，正しくない判定結果が出力されることがあることは容易に予想できる．3.1 節で述べたように，本ツールでは，文献 7) における干渉の定義に基づき，対象とするアドバイスのデータ依存関係と制御依存関係をすべて正しく表現した PDG が構築可能であることを前提としている．しかしながら，実際には 5.3 節で述べた実装上の制約や設計判断により，正確な PDG を作成していない．このため，PDG が等価であるにもかかわらず，振舞いの異なるアドバイスが存在する．つまり，PDG が等価であるからといって，必ず非干渉といえるわけではない．たとえば，本ツールでは，5.3.3 項で述べたように，多相的なメソッド呼び出しを含むアドバイスに対して，干渉を見逃すことがある．

また，本ツールでは，メソッド呼び出しにおける引数間のデータ依存関係を保守的に抽出する．これにより，不要なデータ依存関係が現れ，振舞いの異なるアドバイス群に対して PDG を等価と判定してしまうことや，振舞いの等しいアドバイス群に対して PDG を等価でないと判定してしまうことがある．たとえば，5.2 節の評価実験で用いた AJHotDraw に対して，衝突するアドバイスから呼び出されるメソッドの内部を解析せずに，データ依存関係を保守的に抽出した場合，衝突 G と I は干渉と判定される．これは，誤判定である．本ツールでは，干渉検出コストを削減するため，JDK 内部のメソッドに関しては解析対象とせず，データ依存関係を保守的に抽出している．このことは，衝突アドバイスが JDK 内部のメソッドを呼び出している場合に，同様の誤判定が発生する可能性を示している．

次に，本ツールの干渉検出に関する限界を述べる．PDG が一致しなくても等価な振舞いを持つ複数のプログラムが存在することを，3.1 節で述べた．たとえば，5.1 節で用いた BubbleSort と ShellSort の 2 つのアドバイスを同一実行時点に織り込んだ場合，プログラム

全体で見ると並べ替えが 2 回実行されても実行結果は変わらない．しかしながら，どちらのアドバイスも，もとのデータ（フィールドの値）を定義および使用しているため，結合したアドバイス内部で見ると振舞いは変化していることになる．よって，本ツールはこのような衝突を干渉と判定する．この検出結果は開発者の直観に反する可能性があり，大量に警告が発生する場合には本手法以外の方法で対処する必要がある．

このように，本ツールには干渉検出に対して誤判定を行う場面がある．一般的に，干渉を検出するツールの利用において，非干渉なアドバイス群を干渉と判定することよりも，干渉しているアドバイス群を非干渉と判定する（干渉を見逃す）ことの方が深刻である．ただし，本ツールではすべての衝突を提示したうえで，それらが干渉であるかどうかの判定結果を提供している．もし現実のアスペクト指向プログラムに膨大な数の衝突アドバイスが存在する場合，開発者がすべての衝突を調べることは不可能である．このような状況では，開発者は本ツールの判定結果のみを信頼することになるので，干渉検出時間がより長くなっても判定の精度を向上させるべきである．反対に，現実のプログラムが，開発者が調査可能な数の衝突アドバイスしか含まないのであれば，簡易検査という位置付けで開発者は本ツールを利用することができる．この場合，プログラムのコンパイルと同時に干渉の検出が実施できることが望まれる．5.2 節で示した DB Cluster Framework および AJHotDraw のソースコードを用いた適用実験では，衝突するアドバイスの数は最大 3 個であり，実用的な時間（2.5 秒程度）で干渉の判定が完了した．また，例外に関する干渉を除いて，正しい判定結果を出力した．わずか 2 つのプロジェクトを対象にしただけではあるが，実プロジェクトのソースコードに対して，本ツールが実用的であることが示せた．しかしながら，今回の評価実験の結果が，他の実プロジェクトのソースコードに対しても同様にあてはまるのかどうかについては現時点で不明である．

さらに，5.2 節の評価実験の結果を見る限り，検出された衝突の数はそれほど多くない．このような状況では，人手で干渉の検査を行えば十分であり，本ツールを用いて干渉を検出する意義はそれほど高くないかもしれない．また，5.2 節の評価実験においては，数多くの衝突アドバイスから例外が投げられている．このようなアドバイスに対して干渉を検出するには，衝突アドバイスから例外が投げられる可能性だけを検査するだけでよく，PDG の等価性の活用は冗長である．さらに，別の干渉に関しては，インタータイプ宣言と関係している．たしかに，衝突アドバイスの静的構造を変化させることで発生した干渉を人手で検出することは面倒である．しかしながら，このような干渉は，衝突アドバイスの静的構造の変化を単純に追跡するだけで検出できるかもしれない．

本干渉検出ツールに対して，実用性や有効性という観点で判定結果の妥当性を主張するためには，数多くのアスペクト指向プログラムに対して，さらなる追加実験が必須である．そのうえで，本ツールにおける実装上の制約や設計判断が妥当であるのかどうかを吟味していくことが重要である．

6. 関 連 研 究

AspectJ において，アドバイスの干渉は大きく，アスペクトがプログラムの静的な構造に作用する際に発生するものと動的な振舞いに作用する際に発生するものに分けられる．本論文では後者の干渉を扱っている．

動的な振舞いに関するアドバイスの干渉を扱う研究では，一般的に制御フロー解析とデータフロー解析が用いられている．たとえば，Rinard らの研究[21] では，制御フロー解析を適用することで，アドバイスがメソッドに与える直接的相互作用を，増加（augmentation），削減（narrowing），置換（replacement），合成（combination）に分類している．さらに，ポインタ解析とエスケープ解析を適用することで，メソッドやアドバイスからアクセス（読み書き）されるフィールドを決定する．フィールドへのアクセス状況に基づき，アドバイスがメソッドに与える間接的相互作用を，直交（orthogonal），独立（independent），観測（observation），駆動（actuation），干渉（interference）に分類している．また，Rinard の間接的相互作用の分類を応用したものに，Falcarin らのツールがある[22]．このツールでは，メソッドおよびアドバイスが定義および使用（読み書き）する変数の集合を作成することで，間接的相互作用のうち観測，駆動，干渉を衝突（conflict）として検出する．さらに Balzarotti らは，織り込まれた複数のアドバイスのバイトコードに対して，スライシング（slicing）を適用することで干渉の検出を試みている[23]．この手法では，すでにアスペクトが織り込まれたプログラムに対して，新たに別のアスペクトが織り込まれた場合を対象とし，それぞれのアスペクトが与える影響範囲を表すスライスの積集合が空でない場合を干渉と見なす．

これらの手法はどれも，アドバイスとメソッド（あるいは，すでに埋め込まれたアドバイス）との干渉を検出することを目的としており，アドバイスの実行順序に起因する副作用[⋆1]を検出（アドバイスの実行順序により実行結果が変化するかどうかを判定）することはできない．よって，本論文で提案する手法とは目的が異なる．

⋆1 文献 8) では，このような副作用に起因する問題を，アドバイス優先順位問題（advice precedence problem）と呼んでいる．

アドバイスの実行順序に起因する副作用を扱った研究として，張らの研究[24] と Störzer らの研究[8] がある．

張らの手法では，それぞれのアドバイスに対してデータフロー解析を適用することで，アドバイスをまたぐフィールドの定義および使用を決定する．この手法における干渉とは，織り込まれる 2 つのアドバイスに対して，先に実行されるアドバイスでフィールドを定義し，後で実行されるアドバイスでそのフィールドを使用していることを指す．アドバイスが織り込まれるそれぞれの組合せに対して，フィールドの定義・使用関係を検査することで，干渉するアドバイスの組を発見する．また，制御フロー解析を適用することで，干渉を，定義・使用関係が必ず成立する（干渉が確実に発生する）必然と，成立する場合がある（干渉が発生する可能性がある）蓋然に区別している．アドバイスに関連するプログラムに対して依存解析を適用せず，アドバイス間のデータフローのみに基づき干渉を検出しているため，誤検出や検出漏れが発生することが指摘されている[24]．また，例外処理を含む制御フローに関する干渉を取り扱っていないため，検出される干渉に対する確信度が実用上問題となる可能性が高い．

本論文で提案する手法に最も類似しており，かつ，数多くの干渉を検出可能なものが，Störzer らの手法である．この手法では，異なるアスペクトに記述された複数のアドバイスが，(1) 同一のジョインポイントかつ同一の実行タイミングで織り込まれるとき，(2) 同一のジョインポイントに織り込まれ，そのうち少なくとも 1 つが around アドバイスであるとき，を衝突と定義している．そのうえで，次に示すアドバイスに依存関係を定義し，この定義を実現した規則に基づき干渉を検出する．

- アドバイスデータ依存関係（advice data dependence）
  - （a）変数の定義・使用関係
  - （b）around アドバイス内での `proceed()` 呼び出しにおける実引数や戻り値の変更関係
- アドバイス制御依存関係（advice control dependence）
  - （c）アドバイスが外部に例外を投げる場合
  - （d）around アドバイス内で `proceed()` 呼び出しが 1 回もない，または，2 回以上存在する場合

データフロー解析と制御フロー解析を適用することで，アドバイスの内部に存在する依存関係をアドバイス間の依存関係に抽象化する．これにより，検出時間の短縮に成功している．また，制御依存関係も扱っているため，データの定義・使用関係だけに基づく従来の干渉検出手法に比べて，その精度は向上している．一方，依存関係の抽象化により，干渉検出の精度は我々の手法に比べて劣る．また，アドバイス呼び出しや `proceed()` 呼び出しの引数の扱いが曖昧であり，`proceed()` 呼び出しに対応していない可能性が高い（我々の手法は，`proceed()` 呼び出しにも対応している）．

従来手法との大きな違いは，我々の手法がプログラムの振舞いの等価性に基づき干渉を検出する点である．これは，アドバイスの干渉に関して，プログラムの振舞いという観点から厳密な定義を与えたことに等しい．また，我々の手法では，データ依存関係（データフロー）と制御依存関係（制御フロー）を同一の枠組みで扱うことが可能である．このため，より精度の高い依存解析手法を応用するだけで，干渉検出の精度の向上が実現できるという利点が得られる．

7. おわりに

本論文では，同一実行時点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とらえ，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．そのうえで，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を検出するツールを提案した．さらに，提案ツールの実行時間に関する実験結果と実プロジェクトに適用した実験結果により，ツールの実用性に関して議論した．実際の開発に提案ツールを導入することで，アスペクトの実行順序に起因する干渉を開発時に発見することができ，開発者が予期していなかった副作用によるエラーの発生を防ぐことができる．

今後の課題として，ツール実用性の評価という観点から，さらなる評価実験を行い，解析精度と検出時間との関係や検出結果の妥当性を明らかにしていく予定である．また，5.4 節で述べたように，動的ポイントカットへの対応，プリミティブポイントカットのより詳細な解析に基づく CFG の結合，同一クラスから生成されたインスタンスを区別する手法やインスタンスに対するポインタ解析，リフレクションへの対応を検討している．

謝辞　本研究に関して討論および論文の改良を支援してくれた，立命館大学情報理工学部助手の大森隆行氏に感謝いたします．本研究の一部は，文部科学省科学研究費補助金基盤研究（C）研究課題番号：21500043，および，立命館グローバル・イノベーション研究機構（R-GIRO）研究プログラムによる．

参考文献

1) Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., marc Loingtier,

J. and Irwin, J.: Aspect-Oriented Programming, *ECOOP'97*, pp.220–242 (1997).
2) Filman, R.E. and Friedman, D.P.: Aspect-Oriented Programming is Quantification and Obliviousness, *Workshop on Advanced Separation of Concerns, OOPSLA 2000* (2000).
3) Bergmans, L.M.J.: Towards Detection of Semantic Conflicts betwwen Croscutting Concerns, *Workshop on Analysis of Aspect-Oriented Software* (*AAOS*) (2003).
4) Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Kersten, M. and Griswold, W.G.: An Overview of AspectJ, *ECOOP'01*, pp.327–353 (2001).
5) Eclipse Foundation: AspectJ. http://www.eclipse.org/aspectj/
6) Ferrante, J., Ottenstein, K.J. and Warren, J.D.: The Program Dependence Graph and Its Use in Optimization, *ACM TOPLAS*, Vol.9, No.3, pp.319–349 (1987).
7) Horwitz, S.B., Reps, J. and Reps, T.W.: On the Adequacy of Program Dependence Graphs for Representing Programs, *POPL'88*, pp.146–157 (1988).
8) Störzer, M., Forster, F. and Sterr, R.: Detecting Precedence-Related Advice Interference, *ASE'06*, pp.317–322 (2006).
9) 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，情報処理学会研究報告，SIG-SE-153, pp.7–14 (2006).
10) Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., Moor, O.D., Sereni, D., Sittampalam, G. and Tibble, J.: abc: An extensible AspectJ compiler, *AOSD'05*, pp.87–98 (2005). http://abc.comlab.ox.ac.uk/
11) Horwitz, S., Prins, J. and Reps, T.: Integrating Noninterfering Versions of Programs, *ACM TOPLAS*, Vol.11, No.3, pp.345–387 (1989).
12) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley (1986).
13) Eclipse Foundation: Eclipse. http://www.eclipse.org/
14) Eclipse Foundation: AspectJ Development Tools (AJDT). http://www.eclipse.org/ajdt/
15) Horwitz, S. and Reps, T.: Efficient Comparison of Program Slices, *Acta Informatica*, Vol.28, pp.713–732 (1991).
16) SourceForge.net: SourceForge. http://sourceforge.net/
17) Zhao, J. and Rinard, M.: System Dependence Graph Construction for Aspect-Oriented Programs, Technical Report MIT-LCS-TR-891, Laboratory for Computer Science, MIT (2003).
18) Rothermel, G. and Harrold, M.J.: Selecting Regression Tests for Object-Oriented Software, *ICSM'94*, pp.14–25 (1994).
19) Liang, D. and Harrold, M.J.: Slicing Objects Using System Dependence Graph, *ICSM'98*, pp.358–367 (1998).
20) Woo, J., Woo, J., Attali, I., Caromel, D., Gaudiot, J.-L. and Wendelborn, A.L.: Alias Analysis for Java with Reference-Set Representation, *ICPADS*, pp.459–466 (2001).
21) Rinard, M., Sălcianu, A. and Bugrara, S.: A Classification System and Analysis for Aspect-Oriented Programs, *FSE'04*, pp.147–158 (2004).
22) Falcarin, P. and Torchiano, M.: Automated Reasoning on Aspects Interactions, *ASE'06*, pp.313–316 (2006).
23) Balzarotti, D. and Monga, M.: Using Program Slicing to Analyze Aspect Oriented Composition, *FOAL'04*, pp.25–29 (2004).
24) 張　漢明，野呂昌満，蜂巣吉成，八木晴信：データフロー解析を用いたアスペクトの干渉の検出，ソフトウェア科学会 FOSE'06，ソフトウェア工学の基礎 XIII，近代科学社，pp.19–28 (2006).

(平成 21 年 4 月 3 日受付)
(平成 21 年 9 月 11 日採録)

丸山　勝久（正会員）

1991 年早稲田大学理工学部電気工学科卒業．1993 年同大学院理工学研究科修士課程修了．同年日本電信電話株式会社入社．2000 年 4 月より立命館大学理工学部助教授．2007 年 4 月より同大学情報理工学部教授．2003 年 9 月〜2004 年 9 月，University of California，Irvine 客員研究員．ソフトウェア保守，ソフトウェア開発環境，プログラム解析の研究に従事．情報処理学会 1997 年度山下記念研究賞受賞．博士（情報科学）．電子情報通信学会，日本ソフトウェア科学会，IEEE-CS，ACM 各会員．

平井　　孝（正会員）

2006 年立命館大学理工学部情報学科卒業．2008 年同大学院理工学研究科博士前期課程修了．同年株式会社野村総合研究所入社．現在，同社勤務．在学中は，プログラム解析，アスペクト指向ソフトウェア開発の研究に従事．情報処理学会 2008 年度 CS 領域奨励賞授賞．