データフロー解析を用いたアスペクトの干渉問題の解決

M2004MM043 八木 晴信
指導教員　野呂 昌満

1　はじめに

POP 技術の必要性が指摘され，アスペクト指向プログラミング言語に関する研究 [6] がさかんに行われてきた．なかでも，AspectJ[10] のようにオペレーションフックを用いるプログラミング言語の場合，アスペクトの干渉 [1][3][4][9] が問題になる．[1][3][4][9] では，“アスペクトの干渉は，同一ジョインポイントをもつアドバイスが複数存在するとき，実行時にアスペクト内のコードが開発者の意図しない振舞いをすることである” としている．一方で，異なるジョインポイントのアスペクトについても，他のアスペクトの影響で，実行時にアスペクト内のコードが意図しない振舞いをする場合がある．本研究では，これらの問題は，ともにオブジェクト内の副作用が意図しない形で顕在化する問題であると考える．アスペクトの干渉とは，あるアドバイス内のコードと別のアドバイス内のコードが同一オブジェクトの同一属性の値を変更や参照するという欠陥が実行時に顕在化し，故障となることと定義する．

アスペクトの干渉が起こる可能性のある箇所 (以後，干渉生起可能性箇所) を検出することは一般的には困難である．アスペクトの干渉を起こすアドバイスの記述を発見するためには，それぞれ別々に記述されているアドバイスの記述をすべて知る必要がある．

本研究の目的は，干渉生起可能性箇所をプログラム実行前に検出する方法を提案することである．アスペクトの干渉の可能性は，実行前にできる限り取り除いておくべきであると考える．アスペクトの干渉が起こった場合に，検出結果をもとに原因箇所の特定作業が軽減できると考える．

データフロー解析を用いて干渉生起可能性箇所の検出を行う．アスペクトの干渉は，オブジェクトの属性に対しての副作用が引き起こす問題なので，データフロー解析が検出方法として有効であると考えられる [2]．AspectJ を対象として，干渉生起可能性箇所の検出を行う．提案する方法に従い，干渉生起可能性箇所の検出ツールを実現する．

提案した方法が干渉生起可能性箇所の検出において有効であることを確認した．開発者は，検出ツールを使用することで，プログラム実行前に干渉生起可能性箇所を特定できる．

2　アスペクトの干渉

アスペクトの干渉を簡略化して考えるために，アドバイス内にジョインポイントはないものと仮定する．アドバイスの種類を before, after だけと考えた場合，干渉生起可能性箇所となるアドバイスの組合せは**表 1** のようになる．JP はジョインポイントを表している．JP1.before はジョインポイントが JP1 で，アドバイスの種類が before であるアドバイスを表している．A の組合せは，同一ジョインポイントをもつアドバイスの組合せなので，before アドバイスが先に実行される．before アドバイスでオブジェクトの属性の値が変更され，after アドバイスでその属性が参照される場合，干渉生起可能性箇所となる．B の組合せでは，アドバイスの実行順序は一意に決まらない．同一ジョインポイントでアドバイスの種類が同じの場合，アドバイスの実行順序は処理系が決定する．一方のアドバイスが属性の値を変更し，もう一方のアドバイスがその属性を参照するとき，干渉生起可能性箇所となる．

before，after 以外のアドバイス (around，aftre returning，after thowing) についても，干渉生起可能性箇所となるアドバイスの組合せを考える．before，after 以外のアドバイスの組合せにおいても，A と B の組合せだけであることを確認した．一般性を失わないことが確認できたので，以下ではアドバイスの種類として before と after だけについて議論する．

表 1　干渉生起可能性箇所となるアドバイスの組合せ

advice1 \ advice2	JP1.before	JP1.after	JP2.before	JP2.after
JP1.before	B	A	B	B
JP1.after	A	B	B	B

3　干渉生起可能性箇所の検出

本研究では，データフロー解析を用いて干渉生起可能性箇所を検出する．廣田ら [8] の提案したメソッド利用可能定義とメソッド未定義使用を参考にし，アドバイス利用可能定義とアドバイス未定義使用を定義する．アドバイス利用可能定義とアドバイス未定義使用を用いて，干渉生起可能性箇所の検出を行う．

3.1　干渉生起可能性箇所の検出手順

干渉生起可能性箇所を検出するための手順を以下に示す．

1. 各オブジェクトのすべてのメソッドについてメソッド利用可能定義とメソッド未定義使用を計算
2. すべてのアドバイスにおいて，アドバイス利用可能定義とアドバイス未定義使用を計算
3. 検査アルゴリズム (**図 1** 参照) を用いて干渉生起可能性箇所であるかを検査

3.2 メソッド利用可能定義とメソッド未定義使用

メソッド利用可能定義とメソッド未定義使用は，以下の手順にしたがって計算する．

1. メソッドに対応するフローグラフを作成
2. フローグラフを用いて変数の定義と使用を解析
3. メソッド利用可能定義とメソッド未定義使用を計算

   メソッド利用可能定義：メソッド M 中の属性 a の定義 d に対して，d から出口点まで，属性 a に関する定義透過パス[*1]が存在するとき，d を「メソッド M における属性 a のメソッド利用可能定義」と呼ぶ．

   メソッド未定義使用：メソッド M 中の属性 a の使用 u に対して入口点から u まで，属性 a に関する定義透過パスが存在するとき，u を「メソッド M における属性 a のメソッド未定義使用」と呼ぶ．

オブジェクト指向プログラムでは，あるメソッドで定義された属性が別のメソッドで使用されることがある．メソッド利用可能定義とメソッド未定義使用を用いることで，属性の定義と使用の関係を解析することができる．

3.3 アドバイス利用可能定義とアドバイス未定義使用

アドバイスの記述内において，あるオブジェクトにメッセージを送信した場合に，実行されるメソッドのメソッド利用可能定義の集合をアドバイス利用可能定義と呼ぶ．同様に，アドバイスの記述内において，あるオブジェクトにメッセージを送信した場合に，実行されるメソッドのメソッド未定義使用の集合をアドバイス未定義使用と呼ぶ．

3.4 必然定義と必然使用

フローグラフの全パスにおいて，必ず通過するノードでの変数 x の定義が存在するとき，その定義を変数 x における必然定義と呼ぶ．必然定義でない定義を蓋然定義と呼ぶ．また，フローグラフの全パスにおいて，必ず通過するノードでの変数 x の使用が存在するとき，その使用を変数 x における必然使用と呼ぶ．必然使用でない使用を蓋然使用と呼ぶ．必然定義と必然使用を追加することで，アドバイスの実行において，必ず属性 x に対して定義，または使用が存在することを示すことができる．

3.5 干渉生起可能性箇所の検査

図 1 のアルゴリズムでは，干渉生起可能性箇所の検査をアドバイス利用可能定義とアドバイス未定義使用を用いて検査する．**図 1** のアルゴリズムは，検出対象のプログラム中のアドバイスの集合を入力とし，干渉生起可能性箇所であるアドバイスの組合せを出力する．入力するアドバイスの集合から 2 つの要素を取り出し，干渉生起可能性箇所であるかの検査を行う．アドバイス advice1 と advice2 が干渉生起可能性箇所となるのは，advice1 で定義された属性が advice2 で使用される場合である．また，advice2 で定義された属性が advice1 で使用される場合も同様である．アルゴリズム中で使用する型を定義し (**表 2** 参照)，変数を以下に示す．

表 2　型の定義

型	Advice	Result
属性	result ： Resultの集合	name　:文字列 defVal : boolean useVal :boolean

変数

- ADV，FADV ：Advice の集合
- advice1，advice2 ： Advice
- output ： 干渉生起可能性箇所であるアドバイスの組合せの集合

```
while(ADV の要素数が 2 個以上){
  ADV から取り出した要素を advice1 とする
  ADV から advice1 を取り除いた集合を G とする
  while(G の要素数が 1 個以上){
    G から取り出した要素を advice2 とする
    集合 ATT = { n |r1 ∈ advice1.Result ∧
                    r2 ∈ advice2.Result ∧
                    n = r1.name = r2.name}
    while(ATT の要素数が 1 個以上){
      if n について def-def，use-use の関係以外
      {
        if(B の組合せ または
           実行順序が決定するアドバイスが
                          use-def の関係以外)
          output に ((advice1,n),(advice2,n))
                                    を追加
      }
      ATT から n を削除
    }
    G から advice2 を削除
  }
  ADV から advice1 を削除し，FADV に追加
}
```

図 1　干渉生起可能性箇所の検査アルゴリズム

[*1] パス p=< $n_j, n_{j+1}..., n_{k-1}, n_k$ > において最初と最後のノード以外のすべてのノード $n_{j+1}..., n_{k-1}$ に変数 x の定義が存在しないとき，パス p はノード n_j からノード n_k への変数 x に関する定義透過パスと言う．

必然定義と必然使用を用いることで，検出結果の確度，すなわち干渉の生起の可能性の高さ，を定義できる．干渉生起可能性箇所となるアドバイスの組合せとして必然定義，必然使用の組合せが検出された場合，プログラマが意図したものでなければ，かならずアスペクトの干渉が起きることになる．この組合せは，確度が一番高い組合せである．確度が一番低い組合せは，アドバイスとメソッドの両方において蓋然定義，蓋然使用の場合である．確度を示すことで，開発者はアスペクトの干渉が起こる可能性の高い干渉生起可能席箇所を特定できる．

3.6 検出ツールの実現

干渉生起可能性箇所の検出ツールを実現する．検出ツールは，統合開発環境である Eclipse[5] 上で利用可能なツールとして実現する．ツールを開発環境に組み込むことで開発者はツールを利用しやすくなる．また，Eclipse は構文木を自動生成することや，構文木を操作する API が提供されいるのでツールの実現にあたっての開発労力が軽減できる．

干渉生起可能性箇所の検出結果の出力情報を以下に示す．以下の情報は干渉生起可能性箇所として検出されたアドバイスの組で出力する．

- aspect 名
- advice 名 (アドバイスの種類とジョインポイント)
- オブジェクト名と属性名

アスペクトの干渉が起こった場合に，検出結果をもとに原因箇所を特定する．アドバイスを特定するための情報として，上記の情報は十分であると考える．

4 考察

本研究で提案した干渉生起可能性箇所の検出方法の妥当性について以下の考察を行う．

- 検査アルゴリズムの正しさについて
- アドバイスの検査について

4.1 検査アルゴリズムの正しさについて

図 1 に示した検査アルゴリズムが仕様を満たしていることを証明し，検査アルゴリズムの正しさを確認する．アルゴリズムが仕様を満たしていることを Floyd[7] の方法を用いて証明する．干渉生起可能性箇所の検査アルゴリズムの仕様を以下に示す．

事前条件

$$\sharp ADV \geq 0 \ \wedge \ \sharp ADV = MAX \ \wedge \ \sharp FADV = 0 \ \wedge \ \sharp FADV = MAX - \sharp ADV \ \wedge \ output = \phi$$

入力である ADV の要素数は 0 以上であり，FADV の要素数は 0 である．検査の終了した ADV の要素を FADV 要素となる．定数 MAX は検査対象となるアドバイスの個数であり，ADV の要素数と等しい，事前条件では，output は空集合である．

事後条件

$$\begin{aligned}
&\sharp ADV < 2 \wedge \sharp FADV = MAX - \sharp ADV \ \wedge \\
&output = \{((advice1, advice2), n)) \quad | \\
&advice1 \in FADV \ \wedge \ advice2 \in ADV \cup FADV \quad \wedge \\
&advice1 \neq advice2 \quad \wedge \\
&\exists r1 \exists r2 (r1 \in advice1.result \wedge r2 \in advice2.result \quad \wedge \\
&\neg (r1.defVal \ \wedge \ \neg r1.useVal \\
&\qquad \wedge r2.defVal \ \wedge \ \neg r2.useVal) \quad \wedge \\
&\neg (\neg r1.defVal \ \wedge \ r1.useVal \\
&\qquad \wedge \neg r2.defVal \ \wedge \ r2.useVal) \quad \wedge \\
&n = r1.name = r2.name)\}
\end{aligned}$$

検査終了後は，ADV の要素数は 2 未満となる．ADV と FADV の関係は，$\sharp FADV = MAX - \sharp ADV$ となる．同一属性 n に対して advce1 と advice2 のアドバイス利用可能定義 r1.defVal，r2.defVal とアドバイス未定義使用 r1.useVal，r2.useVal を比較する．advice1 と advice2 が干渉生起可能性箇所となるの条件は，$\neg \ (r1.defVal \ \wedge \neg \ r1.useVal \ \wedge \ r2.defVal \ \wedge \neg \ r2.useVal) \ \wedge \neg \ (\neg \ r1.defVal \ \wedge \ r1.useVal \ \wedge \neg \ r2.defVal \wedge r2.useVal)$ である．

検査アルゴリズムの各箇所における表明が帰納的表明であることを確認した．事前条件，事後条件についても帰納的表明であることが確認できる．したがって，検査アルゴリズムが干渉生起可能性箇所の検査アルゴリズムとして正しいと言える．

4.2 アドバイスの検査方法について

アスペクトの干渉は，3 項以上のアドバイスが関連して起こることはない．本研究では，干渉生起可能性箇所として最小の組合せである 2 項のアドバイスの検査を行なっている．2 項間で関連し，他のアドバイスとさらに関連する場合が存在する．例えば，アドバイス 1 で属性 x が定義され，アドバイス 2 で使用する．アドバイス 2 の使用は，属性 x の定義に用いられ，アドバイス 3 で属性 x を使用する場合である．これらの関連は，2 項間の検査で十分であり，検査することに意味をもたないと考える．干渉生起可能性箇所の検出方法として，2 項間のアドバイスの検査で十分であるといえる．

一般に N 個の組合せは組合せ爆発を起こすとされている．本研究では，N 個の組合せを 2 個の組合せの検査と考えることで，組合せ爆発を回避している．検査は，nC_2 通りの組合せを検査する．nC_2 通りの組合せの検査は，$O(N^2)$ で検査可能であり，検出時間について，計算可能である．

5 関連研究

石尾ら [9] は，プログラムスライスを用いてアスペクト指向プログラムのデバッグを支援している．アスペクトの予期せぬ動作やアスペクトによって発生する故障は発見が困難であり，原因調査を支援する必要がある．アスペクトによって動作不良に陥った場合に，プログラムスライスを用いて，対象となるプログラムを開発者に提示し，原因の特定作業を軽減している．プログラムスライスでは，スライスとしてプログラムを抜き出すだけであり，アドバイス間の関係を考慮していない．例えば，スライスとして 2 つのアドバイスが抜き出される．アドバイス内の記述がともに変数の使用として抜き出される場合があるが，この場合はアスペクトの干渉は起こらない．本研究では，データフロー解析を用いて，アドバイス間の関係の検査を行い，この 2 つのアドバイスを干渉生起可能性箇所として検出しない．プログラムスライスを用いた場合と比べ，より厳密な検査が可能であると考える．

Douence ら [3] は，アスペクトの合成の枠組みを形式的に記述することで，アスペクトの干渉を解決している．アスペクトの干渉が起きるのは，本質的な理由として仕様を与えていないからであるとしている．提案された枠組みに従ってアスペクトを記述することで，アスペクトの干渉が起きるアスペクトの検査を行うことができる．この枠組みに従って実現されたソフトウェアでは，アスペクトの干渉は起きないとしている．しかし，仕様を記述することがプログラムを記述することより困難になる可能性がある．実用性を考えた場合，作成したツールは必要である．

6 おわりに

本研究では，アスペクトの干渉をプログラム実行前に取り除くためにデータフロー解析を用いた干渉生起可能性箇所の検出方法を提案した．干渉生起可能性箇所をプログラム実行前に開発者に提示することで，アスペクトの干渉の可能性を示唆することができる．

本研究では，アドバイス内にジョインポイントは存在しないという仮定のもとに，干渉生起可能性箇所が検出可能であることを確認した．アドバイス内にジョインポイントが存在する場合も考慮した干渉生起可能性箇所の検出方法を提案することを今後の課題とする．

謝辞

本研究を進めるにあたり，熱心に御指導下さいました野呂昌満教授，有益なアドバイスを頂いた張漢明助教授に深く感謝致します．

参考文献

[1] L. M. J. Bergmans, “ Towards Detection of Semantic Conflicts between Crosscutting Concerns, ” in *Proc. Workshop on Analysis of Aspect-Oriented Software*, July. 2003.

[2] D. Callahan, “ The Program Summary Graph and Flow-Sensitive Interprocedural Data Flow Analysis, ” in *Proc. the SIGPLAN ’88 Conference on Programming Language Design and Implementation*, June. 1988, pp. 47-56.

[3] R. Douence, P. Fradet, and M. Sudholt, “ Composition, Reuse and Interaction Analysis of Stateful Aspects, ” in *Proc. the 3rd International Conference on Aspect-Oriented Software Development*, Mar. 2004, pp. 141-150.

[4] P. Durr, T. Staijen, L. Bergmans, and M. Aksit,“ Reasoning About Semantic Conflicts Between Aspects, ”in *Proc. European Interactive Workshop on Aspects in Software*, Sep. 2005.

[5] Eclipse Project, http://www.eclipse.org/, Feb 2006.

[6] T. Elrad, M. Aksits, G. Kiczales, K. lieberherr, and H. Ossher, “ Discussing Aspects of AOP, ” *COMMUNICATION of the ACM*, vol. 44, no. 10, pp. 33-38, Oct 2001.

[7] R. W. Floyd, “ Assigning Meanings to Programs, ”in *Proc. Symposium in Applied Mathmatics*, vol. 19, American Mathematical Society, 1967, pp. 19-32.

[8] 廣田豊彦, 橋本正明, “ オブジェクト指向プログラムのためのデータフローテスト, ” 電子情報通信学論文誌, Vol. J79-D-I, No. 10, pp. 707-718, Oct 1996.

[9] 石尾隆, 楠本真二, 井上克郎, “ プログラムスライスを用いたアスペクト指向プログラムのデバッグ支援環境, ” オブジェクト指向最前線 2003 , pp. 129-136, Sep 2003.

[10] G. Kiczale, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, “ An Overview of AspectJ, ” in *Proc. the European Conference on Object-Oriented Programming* Springer Verlag, June. 2001.