アスペクト指向プログラムのデバッグ支援環境 ——プログラムスライスとコールグラフの利用

石尾 隆[†] 楠本真二[†] 井上克郎[†]

アスペクト指向プログラミングは，複数のオブジェクトに関連した処理をオブジェクトから分離し，新しいモジュール単位「アスペクト」として記述する．アスペクトは保守性や再利用性を向上させるが，同時に，プログラムに新たな複雑さを導入する．たとえば，アスペクトが相互に干渉して動作を阻害するなど，原因の特定が難しい欠陥が発生することがある．このような問題に対して，本論文では，コールグラフとプログラムスライシングを用いたデバッグ支援を提案する．コールグラフは，オブジェクトとアスペクトの制御関係を可視化することで，アスペクトによる無限ループの検出を支援する．また，プログラムスライシングは，プログラムの依存関係を解析することで開発者が扱う必要があるコードを抽出する手法であり，アスペクトがプログラムに与える依存関係の変化をユーザに提示する．アスペクト指向言語 AspectJ を対象にツールの実装を行い，適用実験を行った．その結果，アスペクト指向プログラムにおいてアスペクトが与える影響を効果的に開発者に提示できることを示した．

Debugging Support Environment for Aspect-oriented Program Using Program Slicing and Call Graph

Takashi Ishio,[†] Shinji Kusumoto[†] and Katsuro Inoue[†]

Aspect-Oriented Programming (AOP) introduces new software module named aspect for encapsulating crosscutting concerns. Although a crosscutting concern is written as aspects to improve maintainability and modularity, AOP introduces a new factor of complexity. For example, finding defects caused by an aspect, which modifies or prevents behavior of another aspect, needs much cost. In this paper, we examine a method to support such issues in developing aspect-oriented programs. We propose an application of call graph generation and program slicing to assist debugging. A call graph visualizes control dependencies between objects and aspects and can detect infinite loops. On the other hand, program slicing shows changes of dependencies to a user. We implement a program slicing tool for AspectJ and apply it to certain programs. The experiment shows that our approach is effective for a developer in finding the influence of aspects in a program.

1. まえがき

近年，プログラムの新しいモジュール化手法としてアスペクト指向プログラミングが提案され，利用されるようになってきている[1)]．従来のオブジェクト指向プログラミングにおいて，ロギングや同期処理のような複数のクラスやメソッドが関わる処理のことを横断要素と呼ぶ．横断要素のコードはその処理に参加するすべてのクラスに分散するため，保守性を悪化させる要因となっている．アスペクト指向プログラミングは，横断要素を単一モジュールに記述するための新しい単位「アスペクト」を導入する．

† 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

アスペクトは，アドバイスと呼ばれるプログラム中の特定の実行時点に連動して実行される処理の集合として記述される．実行時点とは，オブジェクト間でのメッセージの送受信などのイベントのことである．アドバイスはクラスやメソッドの境界を越えて作用しうるため，横断要素をクラスから分離して記述することが可能となる．横断要素のモジュール化は，再利用性および保守性の向上につながる．

アスペクトの応用事例は数多く報告されている[2)~4)]．しかし，アスペクトの導入が，プログラムに新しい複雑さをもたらすことも知られている．アスペクトは，オブジェクトの外部からその振舞いを変えることができる．そのため，開発者がオブジェクトの振舞いを把握するには，オブジェクトに関連したアスペクトをすべて調べる必要がある．開発者の予期しない時点でア

スペクトが動作するといった，発見が困難な欠陥を作り込む可能性があることに加えて，アスペクトの干渉と呼ばれる，単体ではそれぞれ正しいアスペクトが相互に動作を阻害するという問題も発生している[5]．

このような問題に対して，アスペクトの予期せぬ動作の可能性や，アスペクトによって発生した障害の原因調査を支援する方法が必要である．

本研究では，アスペクト指向プログラムの開発を支援するために，コールグラフによる呼び出し関係の提示と，プログラムスライシング技術[6]を用いたデバッグ支援を行う．コールグラフとは，メソッドを頂点とし，呼び出し関係を有向辺とするグラフである．これにアドバイスの頂点と辺を加えて，予期せぬアスペクトの動作や，アスペクト間の相互依存による無限ループ発生の可能性を提示する．これによって，開発者がプログラムの実行制御に関連した欠陥を検出しやすくなる．プログラムスライシングとは，プログラム内部の依存関係を解析することで，開発者が注目すべきコードを提示する技術である．アスペクト干渉が与える影響を調べるためには，依存関係を解析し利用するプログラムスライシング，特に実行時情報を用いた DC スライス計算[7]が有効であると期待できる．

統合開発環境 Eclipse[8]をベースに，コールグラフおよび DC スライスを計算するツールの実装を行い，いくつかのアスペクト指向プログラムに対して適用実験を行った．その結果，プログラムスライシングがアスペクト指向プログラミングのデバッグ支援に適していることを示した．

以降，2 章ではアスペクト指向プログラミングの特徴と問題点について説明する．3 章ではコールグラフを用いた，アスペクトを組み込む際の問題検出について説明する．4 章ではプログラムスライシングの概要とアスペクト指向プログラミングへの拡張について説明し，5 章で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ツールの実装と評価について説明する．最後に 6 章でまとめと今後の課題を述べる．

2. アスペクト指向プログラミング

2.1 アスペクト指向の特徴

アスペクト指向プログラミングは，オブジェクト指向プログラミングなど既存のモジュール機構を基に，その弱点を補うプログラミング手法である．たとえばオブジェクト指向プログラミングでは，システムに必要な個々の機能をオブジェクトが分担し，それらが相互通信し，協調することでシステムの目的を実現する．しかし，オブジェクトという単位でシステムの機能を分担する都合上，担当するオブジェクトを 1 つに決められないような特性はうまく扱えない．たとえばシステムの動作を記録していくロギング，エラーが起こったときの例外処理，データベースなどにおけるトランザクションの手続きは，システム内の複数のオブジェクトが連携して実現することが多い．このような処理は横断要素と呼ばれており，横断要素に関わる複数のクラスにコードが分散し，次のような事態を引き起こす．

- 横断要素に関係したクラスやメソッドの集合が変わると，すべての関連したコードをもれなく変更しなければならない．
- クラスやメソッドに，本来の機能と横断要素のコードが混在することにより，横断要素を含んだオブジェクトの再利用性が低下する．クラスやメソッドを横断要素に関連したコードを除去して再利用するか，あるいは，関連したオブジェクト群をまとめて再利用する必要がある．
- 横断要素だけを独立して再利用することができない．もし別の場面で同じ種類の横断要素が必要であれば，そのつど実装しなければならない．

これに対して，アスペクト指向プログラミングは，横断要素を分離・記述するためのモジュール単位「アスペクト」を導入する．

AspectJ[9]などのアスペクト指向プログラミング言語では，オブジェクト指向プログラム内に含まれる実行時点（*join points*）の中から，結合の基準となる集合（*pointcut*）を選択し，手続きを関連付ける．関連付けられる手続きのことをアドバイス（*advice*）と呼ぶ．アスペクトは，通常，1 つ以上のアドバイスによって構成される．

アドバイスが結合可能な実行時点は，言語処理系によって異なるが，次のようなものが使用されている．

- オブジェクトに対するメソッド呼び出し．
- オブジェクトのメソッドの実行（動的束縛の解決後）．
- オブジェクトの持つフィールドへのアクセス．
- オブジェクトでの例外の発生．

このような実行時点に対して，アドバイスはその直前（*before*），直後（*after*），あるいは本来の処理を置き換える（*around*）形式で動作する．言語処理系にもよるが，このようなアドバイスは，その実行コンテキスト（たとえば実行時点が何であるか，呼び出されようとしているオブジェクト，メソッド，引数といった）情報にアクセスし，適切な処理を実行することができる．

```
aspect ParameterValidationAspect {
  before(int x):
    args(x) && call(void *.doSomething(..)) {
    if ((x < 0)||(x > Constants.X_MAX_FOR_SOMETHING)) {
      throw new RuntimeException("invalid parameter!");
    }
  }
}
```

図 1 AspectJ におけるパラメータ検査アスペクトの例
Fig. 1 A parameter checking aspect written in AspectJ.

AspectJ で記述したアスペクトのコード例を**図 1** に示す．ここで定義している ParameterValidationAspect は，どのクラスであれ doSomething という名前のメソッドに対する呼び出しの直前に引数を検査し，条件に違反した呼び出しに対して例外を発生させるアスペクトである．

2.2 アスペクトのもたらす複雑さ

アスペクトの用途については広く研究されている．オブジェクト指向プログラミングで用いられているデザインパターン[10] は，複数のオブジェクトがどのように連携するかを説明した設計部品である．これは，オブジェクトの横断要素の一種であると考えられるため，アスペクトとして記述することで，パターン自身が再利用可能なソフトウェア部品となる場合があることが知られている[3]．また，ソフトウェア開発時のデバッグ支援や，分散アプリケーションでのオブジェクトを横断した性質の記述など，様々な場面でその有用性が示されつつある[2),4]．しかし，アスペクトはプログラムに次のような新しい複雑さをもたらす．

(a) 同一の条件下で動作する複数のアスペクトが存在しうる．アスペクトの動作順序によって，結果が異なる場合がある．

(b) あるアスペクトの動作中に，自分自身あるいは他のアスペクトの動作条件が成立する場合がある．2 つのアスペクトが相互に条件を満たすときは，無限ループを生じることもある．

(c) 動作条件の記述が必要十分であることを確認することが難しい．また，ソフトウェアを変更した際に，動作条件も変更が必要となる場合もある[11]．

これらのうち，(a) と (b) はアスペクトの干渉と呼ばれる問題である[5]．このような干渉を検出する，あるいは予防するための研究が数多くなされている．また，(c) はアスペクトが関係しているオブジェクトの情報をどのように開発者に提示するかという問題である．これに対しては，統合開発環境による部分的なサポートが行われている[12),13]．

アスペクトの干渉や誤作動によって生じる障害の原因を見つけ出すことは難しい．この理由としては，通常，開発者はクラスやアスペクトのようなモジュール単位でプログラムを読んでいくことによって局所的な理解を行うが，読んでいるモジュールから参照しているモジュールなど，開発者から見えない部分でアスペクトから受けている影響には気づきにくい，ということがあげられる[14]．

アスペクトの干渉は必要な場合もあるため，一概に禁止するというわけにはいかない．たとえば，先に登場した，引数の値の範囲をチェックするアスペクトは，他のアスペクトからのメソッド呼び出しに対しても動作しなければならない．これに対して，アスペクトが他のアスペクトの振舞いを破壊しないようにアスペクトの種類を制限することで安全なアスペクトを記述する手法[15] や，アスペクトの扱いを開発者が記述可能とすることで干渉を回避する手法[5] も研究されている．

本研究では，アスペクトの干渉による動作不良の原因を特定する作業を支援することを考える．アドバイスの実行を，対応する実行時点でのメソッド呼び出しに変換するというアイディアに基づいて，アスペクトをプログラムに組み込む時点でのコールグラフを用いた制御誤りの検出と，プログラムに組み込んだ後のプログラムスライシングを用いた欠陥特定の支援を提案する．

コールグラフは，(b) によるアスペクトの無限ループの発生を検出するために使用する．プログラムスライシングは，特定の変数の値に注目して依存関係を解析する．通常は，誤った出力値となった変数に注目し，(a) における同時に動作するアスペクトや，(c) における誤った動作条件がその変数の値に関わっていれば，それらを検出し，問題の原因を追究する作業を支援する．(a) については，アスペクトの動作順序が変化した場合に問題が発生する可能性を検出するという方法も考えられるが，本研究では，実際に発生した問題の原因を探す作業に注目した．また，(c) におけるプログラムの変更時に動作条件が変化する問題[11] については対象外とした．

3. コールグラフによるアスペクト干渉の検出

アスペクト指向プログラミングでは，アスペクトが実行される条件の誤りやアスペクト間の干渉によって，本来動作するべきではない時点でアスペクトが動作するといった故障が発生する．

特に，アスペクトによって無限ループが生じやすいことが知られている[9]．実行することで無限ループに陥ってしまう可能性は，実行してから実際にループに陥った状態をデバッガなどで検査するのではなく，プ

ログラムをコンパイルした段階で検出できることが望ましい．そこで，プログラム実行前にソースコードに対する静的な解析を行い，無限ループの発生可能性を調べる．

無限ループが発生する可能性を簡単に検出するものとして，メソッドを頂点とし，メソッド間の呼び出し関係を有向辺とするコールグラフ（Call Graph）がある．グラフ内のある頂点 v から出発して v 自身に到達可能な経路があるとき，無限ループの可能性がある，ということになる．

コールグラフをアスペクト指向に対して拡張するために，あるメソッド内部にアドバイスが連動する実行時点が含まれているとき，そのメソッドからアドバイスへの呼び出しがあると考え，アドバイスの頂点と動作辺を追加する．メソッド，アドバイスを頂点とし，メソッド呼び出し，アドバイスの起動を辺としたコールグラフを構築する．あるメソッド m の頂点 v_m からアドバイス adv の頂点 v_{adv} に到達可能であるとき，m は adv の実行に影響を与えている，と考える．あるアドバイスが自分自身の影響を受けるとき，無限ループに陥る可能性がある．

アドバイスの呼び出し順序は処理系依存であることから，呼び出しの順序に関する組合せの判定が必要となる．ここで，before および after アドバイスについては，実行順序が変化しても呼び出し関係そのものは変化しないため，無視できる．around アドバイスが複数存在する場合には，その順序が呼び出し関係に影響を与えるが，around アドバイスの個数によっては組合せ爆発を起こす可能性がある．本研究では，複数の around が同一のメソッドに追加された場合には，処理系依存の順序（AspectJ Project で開発されているコンパイラ[16]が用いる順序）による扱いを行うことで，この問題を回避するものとした．

コールグラフの構築においては，オブジェクト指向プログラミング言語の場合と同様，アスペクト指向プログラミング言語の実行時決定要素の扱いが問題となる．AspectJ では，動的に判定される pointcut として制御フローに関する条件を指定する `cflow`，条件式を記述する `if` などがある．このうち，`cflow` についてはコールグラフ上の到達可能性を調べることで，到達しないものを取り除くことは可能である[17]．しかし，それでも解決できない場合や，`if` による判定，フラグ変数による無限ループの回避を行っているようなコードは認識できず，無限ループとして判定してしまう．また，故意の再帰呼び出しに対しても同様である．この問題に対しては，特定の回避コードを使用していることを開発者側が XDoclet[18] などのメタデータを用いて宣言することで故意のループを検知するといった対処法が考えられる．

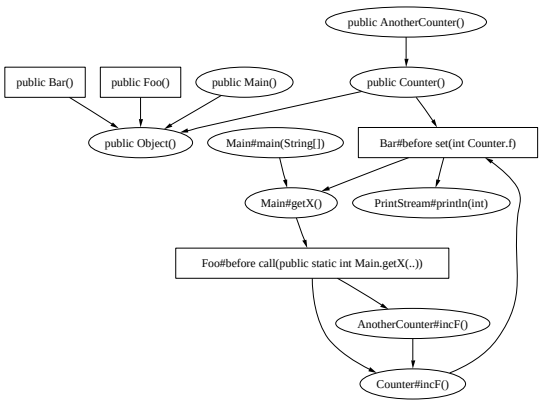


図 2 コールグラフの例
Fig. 2 A call graph.

コールグラフの具体例を**図 2** に示す．楕円形の頂点はメソッド，矩形の頂点はアドバイスを示しており，内部にシグネチャを示している．このグラフでは，メソッド Main.getX に対応する頂点からアスペクトを経由したループが存在していることから，無限ループに陥る可能性を知ることができる．

コールグラフは，プログラムのサイズに比例して巨大化していくことから，ツールの実装においては，どのように提示するかも重要な問題となる．本研究で実装したツールでは，ループが発見された時点でユーザに通知を行うと同時に，ループに含まれる頂点と，その頂点から隣接している頂点（辺の向きに関係なく，たかだか 1 回辺をたどって到達可能な頂点）だけをグラフから抽出して提示する方法を選択した．

コールグラフを用いて，開発者はアスペクト間の制御関係が意図したものであるかどうか確認し，無限ループの可能性を除去することができる．コールグラフを用いた手法以外には，他のアスペクトの干渉検出法として，形式的仕様記述を用いる手法などが考えられるが，それらに比べて実装が容易であり，開発者にとって直観的な情報が提示できる点が特徴である．

アスペクトの動作順序や相互依存関係を開発者が細かく指定することができる処理系[5]や，そもそも干渉を起こさないような限定された種類のアスペクトの利用[15]も研究されている．そのような場合でも，コールグラフを用いて依存関係を表示することで，予期せぬアスペクト間の依存関係の存在を開発者が確認できることは有益である．

4. プログラムスライシング

プログラムスライシングは，プログラムに含まれる変数の参照・代入関係，実行制御文による制御関係，手続き呼び出し関係などの依存関係を解析し，プログラム内の注目すべき部分だけを抽出，提示する技術である[6]．具体的には，ある文のある変数を入力として，その変数の値に影響を与える文の集合を取り出す．入力として与えるコード内の変数の参照位置をスライス基点と呼び，抽出されたプログラム文の集合をプログラムスライス，あるいは単にスライスと呼ぶ．

4.1 スライス計算のアルゴリズム

プログラムスライシングは，次の 3 つのフェイズからなる．

(1) プログラムからの依存関係の抽出
(2) プログラム依存グラフの構築
(3) グラフ探索によるスライスの抽出

フェイズ (1) では，データ依存関係，制御依存関係の 2 つの依存関係を解析する．データ依存関係とは，変数の代入と参照の関係である．プログラム中の 2 つの文 s，t について以下の条件がすべて成り立つとき，s から t に対して変数 v に関するデータ依存関係が存在するという．

- 文 s で変数 v に値を代入している．
- 文 t で変数 v の値を参照している．
- 文 s から t に到達可能な制御フローがある．
- 文 s から t に到達する制御フローのうち，途中で変数 v への代入文がないような経路が少なくとも 1 つ存在する．

また，制御依存関係とは，実行制御文と制御される文の関係である．プログラム中の文 s，t について，以下の条件がすべて成り立つとき，s から t への制御依存関係が存在するという．

- s が条件節である．
- t が実行されるかどうかが，s の判定結果によって決まる．

フェイズ (2) で，これらの依存関係と，手続き・メソッドの呼び出し関係を用いて，プログラム文を頂点，依存関係を有向辺としたグラフを作成する．このグラフをプログラム依存グラフ（Program Dependence Graph，以下 PDG）と呼ぶ．フェイズ (3) では，PDG 上でスライス基点となる頂点を選び，有向辺を逆向きに探索していくことで，スライス基点となった変数に影響を与える文を抽出していく．得られた頂点集合をエディタなどに表示されたソースコード上へ反映し，プログラマへ情報を提供する．

プログラムスライシングの性能は，その依存関係の抽出方法によって決まる．ソースコードから依存関係を取得する静的スライスは，すべての可能性を抽出するため，プログラム理解や検証に用いられる[6]．動的スライスはある特定の入力に対する実行系列上で依存関係を解析し，実際に依存関係が発生したコードだけを抽出するため，デバッグなどに利用される[19]．しかし，実行系列を完全に保存する必要があるため，必要なコストは非常に高い．それらの中間的な Dependence-Cache（DC）スライスは，実行系列を保存せずにデータ依存関係だけを動的に取得することで，実用的なコストで特定の実行系列に関する情報を取得できる[7]．

4.2 アスペクト指向プログラムに対するスライス計算の適用

アスペクト指向プログラムに対してプログラムスライシングを適用し，デバッグ支援を行う．デバッグ作業において，開発者は，まずコールグラフを用いて無限ループの可能性を除去し，デバッグ対象のプログラムの実行は少なくとも終了する状態とする．次に，プログラムに対しテストケースを与えて実行する．このとき得られた結果が不正であった場合に，その原因を調査するためにプログラムスライシングを用いる．

デバッグに用途を絞ると，どのようにしてその出力値が出たか，という実際の経過に注目し，プログラムスライスのサイズを小さく抑えることが重要となる．単一の実行時点で複数のアドバイスが動作する場合の実行順序は処理系に依存するため，静的スライスでは，起こりうるアドバイスの実行順序をすべて調べる必要がある．これは，デバッグ作業を支援する用途には不向きであると考えられる．そこで，本研究では，スライス計算手法として，実行時情報を用いる DC スライスを採用した．

実行結果が不正となるテストケースを実行し，その実行経過を観測して得られた情報を基にプログラム依存グラフの構築とスライス計算を行い，開発者にフィードバックを行う．開発者は，スライスに含まれたプログラム部分だけを調べればよいので，欠陥の原因を特定する作業を軽減することができる．

現在，オブジェクト指向言語 Java に対するプログラムスライシングがすでに実現されているが，アスペクト指向プログラムに対する適用は基本的な手段の提案にとどまっており，実装や評価は行われていない[20]．

本研究では，対象とするアスペクト指向言語として AspectJ を選択し，Java に対するプログラムスライシングを次のように拡張する．基本的なアイディアは，アドバイスの処理を，対応する実行時点でのメソッド

呼び出しに変換するというものである．

データ依存関係 アドバイスは，その実行時点のコンテキスト情報にアクセスすることができる．このデータは，本来の実行時点の頂点から，メソッド呼び出しの引数としてデータが渡されているものとして考える．AspectJ では，アドバイス内で，そのアドバイスが動作する実行時点を表現した *thisJoinPoint* オブジェクトを用いて，その時点でのメソッドの名前や引数情報などを取得するリフレクション機能が利用できる．これは，汎用的なアスペクトを記述するために使用される[9]．実行時情報を用いて実際の動作を解析する方法[21]も存在するが，その適用は将来の課題とし，本研究では，次のように，部分的な解決を行っている．

- メソッド名，ソースコードの位置情報のように，取得に引数が不要なものは，先に書いたように，本来の実行時点の位置からのデータ依存関係と見なす．
- メソッドの第 N 引数の情報を取得する getArgs (N) のように，引数に応じて参照される値が異なるものについては，「引数の列」という単一の存在として取り扱うものとした．このため，getArgs を使用している文は，メソッドの引数すべてを参照したのと同様の扱いとなる．結果として，プログラムスライスは冗長な文を含む可能性が生じる．

アドバイスの結合基準 アドバイスは，その結合基準と関連付けられている．そこで，アドバイスは，結合基準からの依存関係を持つと考え，アドバイスがスライスに含まれるときは結合基準もスライスに含まれるようにする．

アドバイスの呼び出し関係 アドバイスは，その実行時点の頂点に到達しない限りは実行されないため，その実行時点によって制御されていると考える．そこで，実行時点に対応する頂点を，アドバイスへの呼び出し頂点として考える．**図 3** に，メソッド呼び出し頂点における手続き呼び出しの関係を示す．

動的な結合基準 アドバイスが実行されるかどうか，特定のメソッドからの制御フローや変数の値など，実行時の情報によって決定される場合がある．プログラムスライスの計算では，これらの実行時情報の条件以外で指定される「実行される可能性がある」すべての実行時点に，そのアドバイスが実行されるかどうかの判定文があると考え，各実行時点からのアドバイス呼び出しを設定する．また，

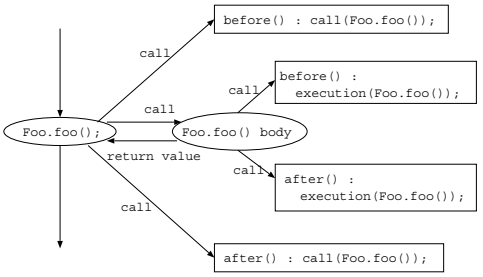


図 3 メソッド呼び出し頂点におけるアドバイス呼び出し
Fig. 3 Before advice call and after advice call at method call node.

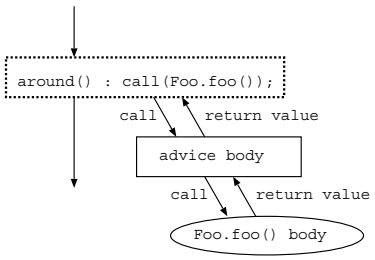


図 4 メソッド呼び出し頂点における around アドバイスの呼び出し
Fig. 4 Around advice call at method call node.

それらの実行時点に対して，条件判定に使われるデータの依存情報の設定も行う．制御フローに関しては，静的解析を行うことで実行される可能性を絞り込み，結合基準をより正確に指定することができる[17]が，それは今後の課題とする．

実行時点の置き換え AspectJ における around アドバイスは，実行時点を完全に置き換える．そこで，実行時点に around アドバイスが関連付けられた場合は，**図 4** のように，その実行時点に対応する頂点を取り除き，代わりに around アドバイスへの呼び出し頂点を追加する．

4.3 プログラム実行時情報の解析

DC スライスでは，プログラムの実行時情報が必要となる．ここでいう実行時情報とは，動的データ依存関係，メソッドの動的束縛情報である．このような実行時情報を収集する処理は，それ自体が横断要素の 1 つであると考えられるため，アスペクトとしてモジュールを記述し対象プログラムに組み込む．我々はすでに実行時情報収集アスペクトを，Java を対象として実現している[2]．

AspectJ ではローカル変数の監視はできないので，この方法で計算されたスライスはメソッド内部の依存関係を静的に計算し，メソッド間の依存関係を DC スライスを用いて計算したものとなる．その結果，静的

スライスよりも小さく，完全な DC スライスに比べると冗長なプログラム文が含まれる場合がある．しかし，実行時のオブジェクトの個々のインスタンスの識別と，フィールドのデータ依存関係を取り出すだけでも，静的なプログラムスライスに比べると十分に有効な結果であることは Java を対象としたプログラムスライシングの実現ですでに示されており[2]，この手法を AspectJ に対して拡張して利用する．このアスペクトを加えた際の干渉の発生については，コールグラフを用いた制御エラーの検出と，AspectJ でのアスペクト間の動作優先度の定義を用いて対処する．

5. 実装と評価

5.1 実装の概要

デバッグ作業は，コードを書き換え，テストケースを再実行するという反復作業として考えられる．このような反復テストをサポートする統合開発環境も多く，プログラムスライシングのような支援ツールも，それらと連携して利用可能であることが望ましい．本研究では，統合開発環境として Eclipse[8] を選択し，これに統合する形式でツールの実装を行った．

Eclipse はオープンソースの統合開発環境で，Java で記述したプラグインを追加することでエディタやコンパイラの機能を拡張することができる．Java および AspectJ を対象としたソースコード入力支援などのプラグインがすでに開発されているため，それらを利用してスライス計算機能を実装した．ソースコードの規模は約 5000 行となった．

Eclipse のプラグインでは，ファイルの保存やコンパイルの終了など，重要なイベントに応じて動作するアクションを定義することができる．そこで，コンパイル時に静的な依存関係の抽出，コールグラフの構築を行うものとした．コールグラフ内部にループなどが検出された場合には，それをメッセージとしてユーザに通知を行い，検出されたループに含まれる頂点と，その頂点から隣接している頂点（辺の向きに関係なく，たかだか 1 回辺をたどって到達可能な頂点）だけを抽出したグラフを生成するようにした．また，エディタ上でソースコードの選択を行い，その部分をスライス基点としてスライス計算を実行できるようにした．

本研究で開発したツールは，AspectJ 用の開発環境である AspectJ Development Tools（AJDT）プラグイン[13] に含まれた構文解析，意味解析，エディタなどを利用して，コールグラフおよびプログラム依存グラフの構築，プログラムスライスの提示を行う．AspectJ は最終的に Java プログラムとして実行されることになるため，たとえば Java バイトコードを対象としたプログラムスライシング手法[22] を適用することも可能である．このようなプログラム変換の結果に対する解析を行う場合，既存の様々な手法やツールが使えるという利点がある．しかし，変換を行う場合には，変換後の言語上での解析結果を変換前の言語に還元できるように，変換の前後での言語要素の対応付けを保存する必要がある．AspectJ における pointcut 記述は Java に変換された場合，言語要素として直接的に表現されなくなるといった問題が起こるため，言語要素の対応付けは難しいと考え，AspectJ の言語要素を直接解釈する実装とした．

表 1 適用対象
Table 1 Target codes.

名称	サイズ（LOC）
ChainOfResponsibility	517
Observer	667
Singleton	375
Mediator	401
Strategy	465

5.2 適用実験

作成したツールの効果を評価するために，2 つの実験を行った．まず最初に，作成したツールを，Java および AspectJ を用いたデザインパターンの実装例としてインターネット上で公開されているコード[23] に対して適用した．ここでは，デザインパターンのうち，**表 1** に示すような，アスペクトとして記述することが効果的とされている 5 種類のパターン[3] の実装を用いた．実装例をそれぞれコンパイルして実行し，スライス計算を行った．以降，5.3 節ではスライス結果について，5.4 節では計算コストについて述べる．次に，スライスのデバッグ支援に対する効果についての評価を行うために，学生の演習課題において，資料としてプログラムスライスを使用する実験を行った．この実験に関する詳細と結果については，5.5 節で説明する．

5.3 スライス結果の評価

デザインパターンを用いたプログラムは，それぞれ動作の経過を表現するようなテキストを出力するように実装されていた．そこで，それら出力用の変数を基点にプログラムスライスの計算を行った．

たとえば Observer パターンでは，監視対象となるオブジェクト（Subject）に対して発行された値を変化させる命令に対して，監視オブジェクト（Observer）が動作し，変化を検知したことを伝えるオブジェクトが動作する．この最後に実際の出力を行う文字列変数をスライス基点とした．ツールによって得られたスラ

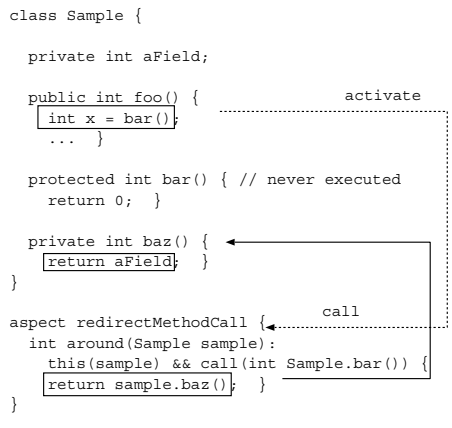


図 5 メソッド置換アスペクトのスライス結果
Fig. 5 A slice including an aspect which replaces a method call.

イス結果と同様の結果を手作業で得るためには，Observer パターンでは，実際の出力を行うメソッドから，Observer，アスペクト，Subject，Subject に値を渡すオブジェクトという順序で制御依存関係を追跡し，その後，制御フローに基づいてデータ依存関係が追跡する必要があった．

他のパターンにおいても同様に，複数のファイルに分散して定義されたクラスやアスペクトの定義を順次追跡する必要があった．プログラムスライシングを用いることで，必要な箇所がまとめて提示されるようになり，作業効率は向上したと考えられる．

このようなメソッドの制御関係を追跡する場合，クロスリファレンス計算ツールは大きな助けとなる．今回の実験規模ではメソッド数，クラス数が少ないために使用しなかったが，クロスリファレンスによって，実行されるメソッドやアドバイスの候補が多数得られるような状態では，それらの候補がプログラムスライスに含まれているかどうかという情報で候補を絞る，あるいはプログラムスライスの中で使われているメソッドをたどるためにクロスリファレンスを使う，といった連携が作業の効率化に重要であると考えられる．

また，プログラムスライスは，次のような場合に有効である．**図 5** は，あるメソッド呼び出しを別のメソッド呼び出しに置き換えるアスペクトを含んだプログラムに対するスライス結果である．このようなメソッド置き換えアスペクトは，単体テスト時の便宜的な実装の変更や，未実装部分の補完などに使用される．このようなアドバイスの存在による依存関係の変化を認識することは難しい．これは，クラスとアスペクトの定義位置が離れているためである．これに対するサポートとしては，統合開発環境において，アドバイスがどのプログラム文に対して動作するかエディタにマークを表示して知らせるためのツールが存在している[13]．しかし，アドバイスが動作することを知らせることはできるが，そのアドバイスが動作したことによって本来の処理が実行されなくなることを示すことはできない．図 5 の例では 1 行分の違いしかないが，実際にはこのクラスを継承したクラス，メソッド内で使用されている他のクラス，関連するアスペクトなどへと，誤認の影響が波及していく可能性がある．このような，アスペクトによって変化した振舞い，依存関係を指摘できることが，プログラムスライシングの大きな利点である．

表 2 動的解析処理の実行時間（単位：秒）
Table 2 Time cost of dynamic analysis (seconds).

対象	通常実行	動的解析付き実行
ChainOfResponsibility	3.76	3.93
Observer	0.32	0.37
Mediator	3.21	5.69
Singleton	0.14	0.32
Strategy	0.18	0.22

5.4 実行コストの評価

スライス計算の実行プロセスにおいて，実行コストは以下の場面で影響を与える．

コンパイル時 静的情報収集のために計算コストとメモリを必要とする．

動的情報解析時 動的情報収集モジュールを付加するため，通常の実行に比べて計算コストとメモリを必要とする．

スライス計算実行時 スライス計算は，構築されたグラフの探索問題である．頂点数と辺数の和に比例した時間で終了する[24]．

コンパイル終了時に行う静的解析は，AspectJ コンパイラが意味解析なども含めて構築したモデル情報にアクセスできるため，構文木を 1 回トラバースして情報を収集する．そのため，プログラム依存グラフ構築に必要な計算量の厳密な評価ではないが，トラバース処理そのものは構文木のサイズに比例している．

動的情報解析時には，プログラムの実行経過を観測し，そこから動的依存関係を抽出するコストが必要となる．この処理に要した時間コストについては，通常実行と比較する形で**表 2** に示しているが，実用的な範囲に収まっている．この種の動的解析は，大規模なテストケースになると莫大な情報を生成する可能性があり[25]，解析結果を保存するために必要な時間が大きなオーバヘッドとなる．テストケースの規模としてどの程度までが実用な範囲であるかについては，実験などを用いた検討が必要であると考えられる．

また，メモリ消費量に関しては，主にプログラムの規模によって影響を受けるが，アスペクトの種類によっても大きく異なる．たとえば，デザインパターンを実装したアスペクトを含んだ約 10000 行のコードに必要な解析コストは約 20 MB だったが，このプログラムのすべてのメソッド呼び出しと実行，フィールドのアクセスに対して動作する，実行時情報収集アスペクトを加えたところ，1 割に相当する約 1000 行程度のアスペクトであったにもかかわらず，動作箇所はメソッド呼び出しに連動するアドバイスだけでも 1000 カ所以上に及ぶため，解析に必要なメモリは 5 倍の約 100 MB に増加し，単純な比例関係とはならなかった．このような解析コストの問題は，AspectJ コンパイラの開発者らによって指摘されている[9] もので，スケーラビリティの実現への課題となっている．

5.5 デバッグ支援への効果

プログラムスライスの情報がデバッグ作業に与える影響を評価するために，大学院の授業において，学生の演習課題の資料としてプログラムスライスを使用する実験を行った．

この実験に参加した学生は全部で 12 名で，全員がコンピュータサイエンスを専攻している．学生は Java プログラミングについては学部の授業などでの使用経験があり，それまで AspectJ を用いたプログラミングの経験はなかった．そこで，実験は次のような 3 つの演習課題として実施した．まず，Java の開発環境として Eclipse の説明を行い，課題 1 として，小規模な Java プログラムに対するデバッグの演習を行った．次に，アスペクト指向プログラミングに関する解説を行い，課題 2 として，簡単なアスペクトを記述する演習を行った．最後に，課題 3 として，アスペクト指向プログラムのデバッグ課題を与え，演習を行った．

課題 3 に用いたプログラムは，AspectJ で記述された小規模なプログラムである．プログラムは，加算と乗算を行う計算式をグラフ構造で表現したオブジェクトを用いて計算を行うというもので，プログラムには次の 4 つのアスペクトが含まれている．

- 節点が評価された際に，その節点の計算式の文字列表現を構築し，結果を出力するアスペクト．
- グラフに循環があった場合に，計算途中でそれを検出するアスペクト．
- 共有節点があった場合に，その節点の再評価を防止するアスペクト．
- 節点を次の計算に再利用するために，計算が終了した節点からグラフの接続を解除するアスペクト．

課題の作成は，複数のアスペクトが干渉したことによって発生するデータ誤りを検出する作業を設定する，という方針で行った．上記のアスペクトのうち，再評価を防止するアスペクトは，節点の評価を省略する．文字列表現を構築するアスペクトは，計算式の評価時に処理を行うように記述されているため，再評価アスペクトによって節点の評価が省略されると，文字列表現の構築処理も飛ばされてしまい，正しい結果が出力されない．

表 3 作業に要した平均時間（単位：分）
Table 3 Time required to debugging task (minutes).

グループ	課題 1	課題 2	課題 3
1. スライスなし	150	186	200
2. スライスあり	200	210	190

学生に与えた課題は，このプログラムが正しく出力すべき結果を参考にプログラムのバグを修正するというものである．学生には，プログラムが出力すべき結果と，それぞれのアスペクトの役割について簡単に説明した資料を渡した．また，学生の半分にあたる 6 名を無作為に抽出し，追加の資料として，プログラムの結果出力を行う変数を基点として計算したプログラムスライスの情報を印刷したものを与えた．このプログラムスライスには，文字列構築アスペクトの最終結果の変数に，再評価防止アスペクトが何らかの形で依存関係を持っていること，そして残る 2 つのアスペクトはまったく関与していないことが示されている．

演習では，学生は，バグの原因，修正したコード，作業に要した時間を提出した．また，プログラムスライスを用いた学生に対しては，その情報が役立ったかどうか，利用した結果の感想についても提出した．以降，得られた提出物を分析した結果について説明する．

学生が，3 つの課題に要した時間の平均値を**表 3** に示す．この実験では，プログラムスライスを用いた学生のグループはプログラムスライスを用いなかったグループよりも良い結果を残したが，スライスを用いたグループと用いなかったグループとの間で，平均作業時間については統計的に有意な差は生じなかった．

プログラムスライスを使用した学生の意見によると，プログラムスライスは，デバッグ作業を開始したときの作業の指針となり，バグの原因を追跡する際には，無関係なアスペクトを調査の対象から除外するために役立つ．また，プログラムスライスとして依存関係のあるコードが明確に提示されることによって，アスペクト干渉の持っている複雑さの 1 つである，開発者から見えないプログラム部分でアスペクトによる影響を受けている可能性[14] を開発者が考慮せずに済んだ．以上が，プログラムスライシングの利点である．

その一方で，実際にプログラムを修正する段階では他のアスペクトへ影響を及ぼさないかどうか調べる必要があり，結局他のアスペクトなども読む必要があった．このことから，影響波及解析など，バグを特定した後の修正作業を支援するような他の手法と組み合わせることでデバッグ作業効率を向上させることが重要であると考えられる．

6. むすび

本研究では，アスペクト指向プログラムに対してプログラムスライシングを適用し，デバッグ支援を行うための開発環境の実装を行った．

アスペクトの起動をメソッド呼び出しの一種と見なすことでコールグラフを構築し，無限ループの可能性を検出することを提案した．また，部分的に実行時情報を用いる DC スライス手法を，同様の発想で拡張した．

コールグラフ構築および DC スライス計算ツールを統合開発環境 Eclipse へのプラグインの形式で実装し，評価実験を行った．その結果，プログラマが誤解しやすいアスペクトの依存関係などを効果的に提示でき，バグを特定する作業を支援できることを確認した．プログラムスライシングは，プログラムの誤った結果に対してどのアスペクトが関係しているかを開発者に提示することができ，開発者から見えないプログラム部分でアスペクトによる影響を受けている可能性[14]を開発者が考慮せずに済む．デバッグ支援においては，変更にともなうアスペクトの動作条件の変化[11]への対処もまた重要であるが，本手法では対応していない．

今後は，プログラムの変更にともなうアスペクトの動作条件への影響波及解析などを用いたプログラム修正作業支援について調査する予定である．また，制御フローに関する静的解析を用いてアスペクトが動作する可能性をより正確に絞り込む手法[17]，実行時情報を用いることでリフレクションの解析を行う手法[21]を適用することで，スライスの正確性を向上することを計画している．また，大規模なプログラム，多数のアスペクトを扱うためのスケーラビリティの改善を今後の課題とする．

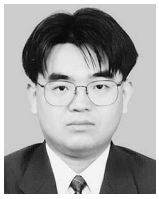
参考文献

1) Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J. and Irwin, J.: Aspect Oriented Programming, *Proc. ECOOP97*, Vol.1241 of LNCS, pp.220–242 (1997).
2) 石尾 隆，楠本真二，井上克郎：アスペクト指向プログラミングのプログラムスライス計算への応用，情報処理学会論文誌，Vol.44, No.7, pp.1709–1719 (2003).
3) Hannemann, J. and Kiczales, G.: Design Pattern Implementation in Java and AspectJ, *Proc. OOPSLA 2002*, pp.161–173 (2002).
4) Soares, S., Laureano, E. and Borba, P.: Implementing Distribution and Persistence Aspects with AspectJ, *Proc. OOPSLA 2002*, pp.174–190 (2002).
5) Pawlak, R., Seinturier, L., Duchien, L. and Florin, G.: JAC: A Flexible Solution for Aspect-Oriented Programming in Java, *Proc. REFLECTION 2001*, pp.1–24 (2001).
6) Weiser, M.: Program slicing, *IEEE Trans. Softw. Eng.*, SE-10(4), pp.352–357 (1984).
7) Ohata, F., Hirose, K., Fujii, M. and Inoue, K.: A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic Information, *Proc. APSEC2001*, pp.273–280 (2001).
8) Eclipse Project. http://www.eclipse.org/
9) AspectJ Team: The AspectJ Programming Guide. http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/
10) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley (1995).
11) Hanenberg, S., Oberschulte, C. and Unland, R.: Refactoring of Aspect-Oriented Software, *Proc. Net.ObjectDays 2003*, pp.19–35 (2003).
12) Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An Overview of AspectJ, *Proc. ECOOP 2001*, Vol.2072 of LNCS, pp.327–353 (2001).
13) AspectJ Team: AspectJ Development Environment. http://www.eclipse.org/ajdt/
14) Störzer, M. and Krinke, J.: Interference Analysis for ApsectJ, *Proc. Workshop FOAL, held in conjunction with AOSD 2003*, pp.35–44 (2003).
15) 一杉裕志，田中 哲，渡部卓雄：拡張ルール：安全に結合可能なアスペクトの記述ルール，日本ソフトウェア科学会，第 5 回プログラミングおよびプログラミング言語ワークショップ (*PPL 2003*)，pp.58–73 (2003).
16) AspectJ Team: AspectJ project. http://www.eclipse.org/aspectj/
17) Sereni, D. and de Moor, O.: Static Analysis of Aspects, *Proc. AOSD2003*, pp.30–39 (2003).
18) XDoclet Project. http://xdoclet.sourceforge.net/
19) Agrawal, H. and Horgan, J.: Dynamic Pro-

gram Slicing, *SIGPLAN Notices*, Vol.25, No.6, pp.246–256 (1990).
20) Zhao, J.: Slicing Aspect-Oriented Software, *Proc. IWPC2002*, pp.251–260 (2002).
21) Gschwind, T., Oberleitner, J. and Pinzger, M.: Using Run-Time Data for Program Comprehension, *Proc. IWPC 2003*, pp.245–250 (2003).
22) Umemori, F., Konda, K., Yokomori, R. and Inoue, K.: Design and Implementation of Bytecode-based Java Slicing System, *Proc. SCAM 2003*, pp.108–117 (2003).
23) Hannemann, J.: Aspect-Oriented Design Pattern Implementations. http://www.cs.ubc.ca/~jan/AODPs/
24) Preiss, B.R.: *Data Structures and Algorithms with Object Oriented Design Patterns in C++*, John Wiley & Sons, Inc., New Jersey (1999).
25) Reiss, S.P. and Renieris, M.: Encoding Program Executions, *Proc. ICSE 2001*, pp.221–230 (2001).

(平成 15 年 9 月 22 日受付)
(平成 16 年 4 月 5 日採録)

石尾　隆（学生会員）
平成 15 年大阪大学大学院博士前期課程修了．現在，同大学院博士後期課程在学中．アスペクト指向プログラミングおよびプログラム構造解析の研究に従事．



楠本　真二（正会員）
昭和 63 年大阪大学基礎工学部情報工学科卒業．平成 3 年同大学大学院博士課程中退．同年同大学基礎工学部情報工学科助手．平成 8 年同大学講師．平成 11 年同大学助教授．平成 14 年大阪大学大学院情報科学研究科コンピュータサイエンス専攻助教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価，プロジェクト管理に関する研究に従事．電子情報通信学会，IEEE，PM 学会，JFPUG 各会員．

井上　克郎（正会員）
昭和 54 年大阪大学基礎工学部情報工学科卒業．昭和 59 年同大学大学院博士課程修了．同年同大学基礎工学部情報工学科助手．昭和 59 年～昭和 61 年ハワイ大学マノア校情報工学科助教授．平成元年大阪大学基礎工学部情報工学科講師．平成 3 年同学科助教授．平成 7 年同学科教授．工学博士．平成 14 年大阪大学情報科学研究科コンピュータサイエンス専攻教授．ソフトウェア工学の研究に従事．電子情報通信学会，日本ソフトウェア科学会，IEEE，ACM 各会員．