情報処理学会論文誌　プログラミング　Vol. 4　No. 1　1–12 (Mar. 2011)

コード領域を対象とする関心事を扱うための アスペクト指向プログラミング言語の拡張

赤井駿平[†1]　千葉滋[†1]

本稿ではアスペクト指向プログラミング言語の新たな言語機構である regioncut と assertion for advice を提案する．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ではモジュール化を行いにくい横断的関心事をアスペクトという単位でモジュール化する技術である．しかし，従来のアスペクト指向プログラミング言語では同期処理や例外処理などの，コード領域を対象とする関心事のモジュール化をうまく行えないという問題があった．そこで我々は regioncut という指定子をアスペクト指向プログラミング言語である AspectJ に導入することで，コード領域に対する関心事をアスペクトとしてモジュール化できるようにする．しかしながら，ある時点で正しく適用されていたアスペクトがコードの変更を行っていく過程で，適用されなくなってしまう危険性が従来よりも大きくなってしまう．その危険性を低下させるため，我々は assertion for advice という言語機構をあわせて導入する．これを用いて，アスペクトが意図した箇所に適用されているかどうかを静的に検査する．これら 2 つの言語機構をオープンソースのライブラリである Javassist および Hadoop に適用して評価を行い，同期処理に関する関心事をアスペクトに分離できることを確認した．

Regioncut: A Designator for Selecting a Code Region in Aspect-oriented Programming Language

Shumpei Akai[†1] and Shigeru Chiba[†1]

This paper proposes two language constructs for aspect-oriented programming language: Regioncut and Assertion for Advice. Aspect-oriented programming helps programmers to modularize crosscutting concerns, which object-oriented programming hard to do, as an aspect. However, in the existing aspect-oriented programming languages, it is difficult to deal with concerns related to code regions, for example, synchronization and exception handling concerns. To address this problem, we propose a new language construct named regioncut, a new kind of pointcut designator, for aspect-oriented programming language. It enables programmers to select code regions and to modularize such a concerns. To select the region which programmers need to modularize, they should specify more information than ordinary pointcuts. It increases risk that selected regions get unselected. To decrease the risk, we also propose Assertion for Advice. By Assertion for Advice, programmers can check programs whether aspects are applied to the expected points. We evaluated the design of our constructs by applying them to the open-source software product Javassist and Hadoop. We found that they have enough power to separate synchonization concern into aspect.

1. はじめに

同期処理は主となるプログラムの中から，モジュールとして分離することが望ましい関心事である．同期処理はその粒度，つまり同期処理を行うコードの範囲の大きさによってその性能特性が変化する．粒度が細かい同期処理ではプログラムの並列性が向上するが同期処理のオーバヘッドが大きくなる．一方，粒度の粗い同期処理では，並列性が低いがオーバヘッドが小さくなる．そのため，コア数の多い CPU を持つ計算機上で動作させる場合は粒度の細かい同期処理の方が良い性能を発揮することが期待できるが，コア数の少ない計算機では粒度の粗い同期処理の方が性能が良くなる場合が存在する．このように，同期処理の最適な粒度はそのプログラムを動作させる環境によって左右される．このような場合に，同期処理をモジュールとして分離して記述し，複数の粒度の同期処理を実装したモジュールを選択できるようになっていれば，簡単に良い性能の同期処理を得ることができる．

アスペクト指向プログラミング（AOP）はオブジェクト指向プログラミングではモジュールとして記述することが難しい横断的関心事を，アスペクトというモジュールとして分離できるようにするプログラミング手法である．AOP をサポートするプログラミング言語としては Java 言語を拡張した AspectJ [7] が有名である．横断的関心事の例としては，ロギング，例外処理などがあげられる．AspectJ では，プログラム中に存在する，メソッド呼び出しやフィールドの参照などの処理を実行する点を *join point* と呼び，*pointcut* という仕組みを用いて，プログラム中に存在する多くの join point から，関心のあるものを選択する．そして，選択した join point で実行するコード片を *advice* と呼ぶ．そして，pointcut と advice をまとめたモジュールを**アスペクト**と呼ぶ．

AspectJ を用いることで同期処理のような関心事を分離して記述可能なように思える．し

†1 東京工業大学
Tokyo Institute of Technology

かし，冒頭の例にあげたような異なる粒度の同期処理をアスペクトとして記述する場合には問題が生じる．1 つ目の問題は，AspectJ ではメソッド本体全体のような限られたもの以外のコード領域を，join point として扱うことができないことである．このことにより，AspectJ およびその基となる Java で同期処理を行う際に用いられる synchronized 文で囲まれているブロックのようなコード領域に対して処理を行うアスペクトを記述することができない．

2 つ目の問題は，AspectJ ではアスペクトが確実に実行されることを保証していない点である．AspectJ では，コンパイル時またはロード時に同期処理を行うアスペクトが正しく織り込まれていなくても警告はしない．同期処理は欠けてはならない関心事であり，クリティカルセクションにおいて同期処理は必ず実行されなければならない．加えて，同期処理の有無は事前に発見する必要性が高い．同期処理はプログラムの外から見える振舞いを変えるわけではないため，同期処理用のアスペクトが織り込まれていないことはタイミングにより，偶然に競合が発生して初めて発覚するからである．そのため，アスペクトの欠如を発見する機構が必要である．こういった機構が存在しない場合，同期処理をアスペクトとして実装できても，実用性は低い．

これらの問題を解決するために，我々は論文 2) において 2 つの新しい言語機構，*regioncut* と *assertion for advice* を AspectJ に導入した．regioncut は AspectJ 上で pointcut 指定子と同様に利用することができる言語機構であり，点である join point の代わりにコード領域を選択する．regioncut では，同期処理，例外処理，トランザクションなどに関するコード領域を選択し，それらの処理をアスペクトを用いて記述することを目的とする．領域を選択するためには，選択したい領域に含まれている join point の列を pointcut 指定子を用いて記述する．assertion for advice は advice が希望する箇所で本当に実行されるかどうかを検査するための言語機構である．これは，指定した advice が指定したメソッドに作用する場所に織り込まれているかをコンパイル時に検査する．これを用いることで同期処理のような必ず実行されなければならないアスペクトがコードの変更によって織り込まれなくなってしまう，といった誤りを検知する．

本稿では，文献 2) で導入した regioncut に対し操作的なセマンティクスを与えるとともに，従来は動的な検査を行っていた assertion for advice を静的に検査を行う方法を述べる．さらに，実験により同期処理をアスペクトとして記述するために regioncut が有用であること示し，また，assertion for advice を用いることで，ソースコードの修正により領域を選択できなくなってしまった regioncut を発見可能であることを示す．

本稿の残りでは，2 章で既存のアスペクト指向プログラミング言語における問題点を指摘する．3 章では，regioncut の言語機構，4 章では assertion for advice の言語機構について説明する．5 章では，regioncut と assertion for advice の実装について述べる．6 章では，本稿で導入する 2 つの言語機構を実際のプログラムに適用し，それを通じて評価を行う．7 章で関連研究を提示し，8 章で本稿をまとめる．

2. 同期処理とアスペクト

2.1 Javassist での同期処理

Javassist[5] は Java バイトコードを編集するためのライブラリであり，web アプリケーションフレームワークなどで利用されている．2006 年に Javassist に対し 1 つのバグレポート[1] が提出された．これは，Javassist にスレッドセーフでない箇所が存在する，というバグを報告するものである．このバグを単純に直すだけならば，そのスレッドセーフでない箇所で同期処理を行うように，Java の synchronized 文を追加するだけで修正できる．

しかし，同期処理を行う際に，同期処理の粒度による性能の違いが問題となる．細かい粒度の同期処理を行えば並列性が向上し，性能が向上する可能性がある．しかし，同期処理の回数が増えるためオーバヘッドが大きくなる．逆に粗い粒度で同期処理を行うと，並列性は低下するがオーバヘッドが小さくなる．我々が過去の論文 8) でも示したように，過度の並列性はパフォーマンスを悪化させる場合がある．2006 年時点でのプロセッサはコア数の少ないものが多く，そのようなプロセッサを持つ計算機上では細かい粒度の同期処理を行うと性能が悪化することが確認されたため，Javassist ではそのバグを修正させるための同期処理を粗い粒度で行うことを選択した．

このことから，同期処理はアスペクトとして分離して記述するとよい関心事であることが分かる．複数の実装の同期処理をアスペクトとして提供することができれば，ユーザがそれらのうちの適切な方を選択できるようになる．アスペクトを用いているため，ユーザはソースコードを修正することなく性能の良い方の同期処理を得ることができる．Javassist でバグを修正した際は，AspectJ ではなく Java で書かれているため，同期処理はハードコーディングする必要があり，ある計算機では速く動くが別の計算機では遅い，という状況が発生した．

2.2 AspectJ の問題点

AspectJ は様々な関心事をモジュール化することができるアスペクト指向プログラミング言語であり，そのような関心事の例には，ロギングやオブザーバパターンなどがあげられ

る[7]．しかし，AspectJ によって同期処理をモジュールとして分離するには，2 つの問題点が存在する．1 つ目は，AspectJ における join point の粒度が，同期処理を行うには適切でないという点である．AspectJ の call pointcut が選択する join point はメソッドを呼び出す点であり，get，set pointcut が選択する join point はそれぞれフィールドの参照と書き込みの点である．これらの join point は単一の式だけを対象にするため，粒度が細かすぎて同期処理を行いたい箇所の選択には適さない．execution pointcut が選択する join point は 1 つのメソッド全体であるため，粗い粒度の同期には適するが，細かい粒度の同期には適さないため，異なる粒度の同期アスペクトを提供することができない．同期処理を行う可能性のある場所をあらかじめメソッドとして抽出して実装していれば，execution pointcut を利用して，同期処理をアスペクトとして記述することができる．しかし，様々な粒度の同期処理をすべて見越して各コード領域をメソッドとして抽出することは煩雑で手間がかかる．また，すべての粒度に対応したコードを書くことは困難である．さらに，必要以上にメソッドを分割して記述することになり，コードの見通しも悪くなってしまう．同期処理をアスペクトとして実装するためには，コード領域を join point として扱うための仕組みが必要となる．

2 つ目の問題点は，同期処理が必要な箇所に関して少なくとも 1 つの同期処理が実行されることをどのように保証するか，という点である．AJDT[12] などの統合開発環境に組み込まれる開発ツールを用いれば，advice が織り込まれている場所を表示することができる．しかし，織り込まれなくなってしまった場所は表示されない．そのため，ソースコードを目視で眺めて，織り込まれているべき場所に織り込みを示す表示があることを確認しなければならず，この問題には有用ではない．

3. Regioncut

2 章の問題を解決するために，我々は AspectJ に対し *regioncut* という新たな言語機構を追加する．regioncut は pointcut の一種であるが，実行点ではなくコード領域を選択する．regioncut では，同期処理，例外処理，トランザクションなどの処理を実装しているコード領域を選択することを目的とする．

regioncut では選択する対象となる領域を特定するために，その領域に含まれている join point の列を指定する．このときに指定できる join point は call，get，set の 3 種類の join point である．call はメソッド呼び出しを行う点，get はフィールドの値を取得する点，set はフィールドへ値を書き込む点を表す．同期処理を行っている領域には副作用が起こる処理が，例外処理やトランザクションを行う領域には，例外を発生させる可能性のある処理が含まれている．そのような処理とは，メソッド呼び出しやフィールドへのアクセスであため，regioncut を選択するために指定できる join point として，call，get，set の 3 つを使用する．

3.1 概　　要

regioncut の文法は region[*pointcut1,pointcut2*] のように，AspectJ の pointcut 指定子をカンマ区切りで並べたものである．regioncut は引数に指定された pointcut に合致する join point をその順番どおりに含んでいる領域を選択する．regioncut の中で使用できる pointcut 指定子は，call，get，set である．regioncut は単一のメソッドの中の領域を選択し，メソッド呼び出しの先の他のメソッドの中の join point は考慮しない．

以下のコードは regioncut の例である．

```
pointcut rc1():
  region[
    call(Object ArrayList.get(int)),
    get(int Foo.bar)
  ];
```

この例では 2 つの pointcut 指定子を regioncut の引数として与えている．この regioncut は ArrayList クラスの get(int) メソッドの呼び出しから始まり，Foo クラスの bar フィールドの参照で終わる領域を選択する．

regioncut の引数には 3 つ以上の pointcut 指定子を指定することができる，

```
pointcut rc2():
  region[
    call(Object ArrayList.get(int)),
    set(* Foo.foo),
    get(int Foo.bar)
  ];
```

この regioncut は，ArrayList.get(int) メソッドの呼び出しから始まり，途中に Foo.foo フィールドへの書き込みがあり，Foo.bar フィールドの参照で終わるような領域を選択する．中間の pointcut 指定子を指定することで，同じメソッド内に似た領域が複数存在するときに，それらを区別するために利用することができる．

regioncut によって選択された領域では before，after，around のそれぞれの advice を実行することができる．領域は返り値を持たないため，around advice の戻り値の型はつねに void となる．

現在の仕様では，異なる regioncut が選択した領域が部分的に重なるような場合には，around advice を適用することができない．次節で述べるセマンティクスを素朴に実装した場合は，前方にある領域が選択されて，後方にある領域が無視されてしまう．しかし，実際の実装では安全のためにコンパイル時にエラーを示すようにしている．

3.2　セマンティクス

regioncut の実行時のセマンティクスは**図 1** のように，Scheme 風のコードで操作的に与えられる．実装上は do-while や，try-catch-finally などを含むのすべての Java の制御構造に対応しているが，説明を簡単にするために，以下では制御構造として if-else および while に限定して考える．

```
(define (eval−method method−body env)
        (eval−stmts method−body '() '() env))

(define (eval−stmts stmts following−stmts proceed env)
   (let ((advice (lookup−advice stmts)))
      (cond (advice
               (eval−advice advice stmts) ; マッチするadvice があれば advice を評価
               (eval−stmts following−stmts '() proceed env)) ; 後続の文を評価
            ((= (length stmts) 1)
               (eval−statement (car stmts)) ; 文が 1つだけならそれを評価
               (eval−stmts follwoing−stmts '() proceed env)) ;後続の文を評価
            ((> (length stmts) 1) ; 文が 1つ以上
               ; 列の最後の文を後続の文の先頭に加え評価し直す
               (eval−stmts (take stmts (− (length stmts) 1))
                           (cons (last stmts) follwoing−stmts) proceed env)))))

(define (lookup−advice stmts) ; 文の列にマッチするadvice を返す
   (find all−advice−regioncut−pairs
         (lambda (advice regioncut) (match stmts regioncut))))

(define (eval−advice advice proceed env) ; advice を評価
  (eval−stmts (advice−body advice) '() proceed env))

(define (eval−statement stmt proceed env)
  (cond ((expr? stmt) (eval−expr stmt env))
        ((if? stmt) (if (eval−expr (cond−part stmt) env) ; if 文の評価
                        (eval−stmts (then−part stmt) '() proceed env)
                        (eval−stmts (else−part stmt) '() proceed env)))
        ((while? stmt) (cond ((eval−expr (cond−part stmt) env) ; while 文の評価
                              (eval−stmts (while−body stmt) '() proceed env)
                              (eval−statement stmt proceed env))))
        ((block? stmt) (eval−stmts (block−body stmt) '() proceed env)) ; ブロックの評価
        ((proceed? stmt) (eval−stmts proceed '() '() env))))
                        ; advice での proceed() 文の評価
```

図 1　regioncut を用いたプログラムを評価するための eval-method 関数
Fig. 1　eval-method function for evaluating a program with reginocut.

メソッドの評価は，この eval-method 関数によって行われる．eval-method の引数 method-body および eval-stmts の引数 stmts は，文のリストである．評価関数の本体である eval-stmts が，次に評価すべき文を先頭に含むリストを引数 following-stmts として受け取る点がこのセマンティクスの特徴である．これを使って，まず文のリスト全体に合致する regioncut があるか調べる．合致する regioncut があればそれに対応する advice を実行する．ない場合は，文のリストの末尾の文を 1 個取り除いて，つまり対象の領域を小さくし，再び合致する regioncut があるか調べる．このとき取り除いた文は following-stmts の先頭に入れられる．文のリストの長さが 1 になっても合致する regioncut がない場合は，その文を eval-statement で評価し，後続の文 following-stmts について同様のことを繰り返す．

つまり，メソッドの先頭から最後までの領域から regioncut に合致する領域を探し，なければ少しずつ領域を小さくして探していく．合致する regioncut が見つかれば，対応する advice を実行，文が 1 つになるまで見つからなければ，その文を再帰的に評価する．評価した後は後続の文の評価を続ける．したがって，合致する regioncut があるか否かを検査する文のリストは，たとえば**図 2** の 1 から 7 の順番で選ばれる．

文のリストと regioncut のマッチング

文のリストと regioncut のマッチングは，regioncut の引数の pointcut の列に合致する join point が指定された順番どおりに文のリスト内に登場すかどうかで判断する．このとき，文のリストの先頭の文と regioncut の先頭の pointcut および，最後の文と最後の pointcut が合致していなければならない．

領域を大きな範囲から順に探索していくので，pointcut の列の先頭と終端に合致する join

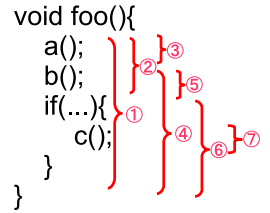

図 2　regioncut において評価される領域の範囲と順序
Fig. 2　Range and order of evaluation with reginocut.

```
;stmts が rc とマッチするか
(define (match stmts rc)
  (let ((rc−rest (match∗ (car stmts) rc)))
    (if (eq? rc−rest rc) ; stmts の先頭と rc の先頭が合致しないか
        #f   ; しなければ偽
        (null? (match−seq (cdr stmts) rc−rest))))) ;rc が最後まで合致したら真

;stmts(文の列) と rc (pointcut 列)のマッチングを行い,合致しなかった残りの rc を返す
(define (match−seq stmts rc)
  (if (null? stmts)
      rc
      ;先頭から順にマッチングを繰り返す
      (match−seq (cdr stmts) (match∗ (car stmts) rc))))

;stmt と rc のマッチングを行い，合致しなかった残りの rc を返す
(define (match∗ stmt rc)
  (cond ((expr? stmt) (if (include−jp? stmt) (cdr rc) rc))
        ((while? stmt) ; while では条件，ボディの順にマッチング
                       (let ((rc−after−cond (match∗ (cond−part stmt) rc)))
                         (match∗ (body−part stmt) rc−after−cond)))
        ((if? stmt)  ; if では，条件の後に，then と else をそれぞれ独立にマッチング
                       (let ((pc−rest (match∗ (cond−part stmt) rc)))
                         (shorter (match∗ (then−part stmt) rc−rest)
                           ; 残った rc の短い方を採用
                                  (match∗ (else−part stmt) rc−rest))))
        ((block? stmt) (match−seq (block−body stmt) rc))))
```

図 3　regioncut のマッチング関数 match
Fig. 3　Matching function match for regioncut.

point が構文上異なる入れ子レベルのブロックに存在した場合でも，それら全体を包含するようなブロックが領域として選択される．このため，if 文の途中で領域が終わるような，不自然な領域の選択は行われない．たとえば，`region[ call(* *.a()), call(* *.b()), call(* *.d())]` は次のブロック全体に合致する．

```
{
  a();
  if(cond){ b(); }else{ c(); }
  d();
}
```

与えられた regioncut rc が文のリスト stmts 全体と合致するか検査するのは**図 3** の match 関数である．リストの要素は，式文か，while 文か，if 文か，文のリスト（つまり入れ子になったブロック）であるとする．

文が while 文である場合は，条件式とループ本体とが連続する 2 つの文であるかのように，条件式，ループ本体，の順にマッチングを行う．if 文の場合は，条件式と then 部，もしくは，条件式と else 部とをそれぞれ regioncut とマッチングを行う．2 つのうち，pointcut が多く合致した方，つまり合致せずに残った pointcut の列が短い方を通るものとして，後続の文の列に対し，マッチングを続ける．このアルゴリズムに従った場合，たとえば，`region[ call(* *.a()), call(* *.c())]` は，次のブロックに合致する．

```
{ a(); b(); c(); }
```

3.3　コンテキストの取得

Java では synchronized 文を使用する際に，ロックを行うオブジェクトを指定する．同期処理をアスペクトとして分離して記述しようとする場合，ロック対象のオブジェクトに advice の中からアクセスできる必要がある．そこで regioncut を用いて，選択された領域のコンテキスト，つまりその領域内でアクセス可能な変数やフィールドの値．を取得できるようにする．ただし，選択された領域の途中の値には実行前にアクセスすることができないため，取得可能なコンテキストは選択された領域の直前の位置でアクセス可能なものに限定する．

regioncut では引数として与える pointcut 指定子に args および target pointcut を `&&` で結合することができる．args と target は，AspectJ に従来から存在する pointcut である．args は `&&` で結合された pointcut に合致するメソッド呼び出しの引数を advice 内で取得できるようにする．target は同様にメソッド呼び出しのレシーバを advice 内で取得できるようにする．結合された args または target pointcut が領域内の join point に合致し，かつ，その join point の引数およびレシーバの値が選択される領域の先頭で取得可能な場合，その値が advice への引数として渡される．

図 4 は，コンテキスト取得の例である．regioncut 内の 2 つ目の pointcut に結合されている args(n) が b() メソッドの引数に与えられている変数 i に合致する．そして，変数 i は領域の先頭の a() メソッドの呼び出しの前で宣言されているので，advice の引数 n に渡される．

4.　Assertion for Advice

アスペクト指向プログラミングには *fragile pointcut*[10] という問題が存在する．これは，プログラムの編集を行うと，今まで正しく join point を選択していた poincut が，正しい箇所を選択しなくなってしまうという問題である．regioncut では，選択したい領域に含ま

```
1  class Foo{
2    SomeObject obj;
3    void bar(){
4      int i=10;
5      a();
6      b(i);
7      obj.c;
8    }
9  }
10
11 void around(int n):
12   region[
13     call(∗ ∗.a()),
14     call(∗ ∗.b(int)) && args(n),
15     get(∗ SomeObject.c)
16   ]
17   { proceed(n); }
```

図 4　regioncut によるコンテキストの取得
Fig. 4　Context exposure by regioncut.

れる join point を複数指定するため，call などの通常の pointcut よりもプログラムの編集に弱くなってしまう．

```
1  void update(){
2      Data data=storage.getData(key);
3      modifyDataA(data);
4      modifyDataB(data)
5      storage.writeData(key);
6      // その他の処理
7      Data data2=storage.getData(key2);
8      modifyDataA(data2);
9      storage.writeData(key2);
10 }
```

上の例は，単純なデータベースのデータを更新するプログラムの例である．このとき，メソッド内の前半の getData メソッドから，writeData メソッドまでを regioncut を用いて選択して同期処理を行おうとした場合，後半の領域と区別するために，

```
1  region[ call(Data ∗.getData(..)), call(void ∗.modifyDataA(..)),
2          call(void ∗.modifyDataB(..)), call(void ∗.writeData(..))]
```

と記述する．ここで，update メソッドが，

```
1  void update(){
2      Data data=storage.getData(key);
3      modifyDataB(data)
4      modifyDataA(data);
5      storage.writeData(key);
6      // その他の処理
7      Data data2=storage.getData(key2);
8      modifyDataA(data2);
9      storage.writeData(key2);
10 }
```

のように，modifyA と modifyB の順序を入れ替えるような変更を行うと，regioncut は目的の領域を選択できなくなってしまう．このほかにも，modifyA と modifyB を実行する部分を別のメソッドに切り分けて，そのメソッドを呼ぶようにするリファクタリングを行った場合も，目的の領域を選択できなくなってしまう．このような場合に，regioncut を用いて実装した同期処理がプログラムから欠如してしまうことになり，バグが発生する．

同期処理が fragile であるとバグの発見は困難であるため，単純に regioncut を用いて同期処理を実装すると実用性が低い．同期処理の実装に regioncut を用いる場合の実用性を向上させるため，我々は AspectJ に assertion for advice という言語機構を regioncut と合わせて導入する．これを用いることで，advice が想定したメソッドに適用されているかどうかを静的に検出できる．これにより regioncut が fragile であるという問題を軽減でき，実用性が向上する．

4.1 概　　要

assertion for advice は，@AssertAdvised と@SolveProblem という 2 つのアノテーションから構成される．@AssertAdvised はメソッドを修飾し，そのメソッドが advice による影響を受けなければならないことを宣言する．@SolveProblem は advice を修飾し，@AssertAdvised の付いたメソッドに影響を与えることを示す．

図 5 と**図 6** は assertion for advice を利用する例である．@AssertAdvised アノテーションの引数には advice によって解決されることを期待する関心事の名前を指定する．@SolveProblem アノテーションの引数には，advice が適用されるべきメソッドが定義されているクラス名，および解決する関心事の名前を記述する．この 2 つのアノテーションを基に，以下の 2 つの条件を満たすかを検査する：(1) @SolveProblem の引数のクラスの中に，対応する関心事の名前を持つ@AssertAdvised アノテーションで修飾されたメソッドが存在するか，(2) そのメソッドに対し，advice が影響を与えているかどうかを検査する．図 6 の例では，検査器は，"name_of_problem" という名前の関心事を引数に持つ@AssertAdvised アノ

```
1 class A{
2   @AssertAdvised("name_of_problem")
3   void foo(){ bar(); }
4 }
```

図 5　メソッドへの@AssertAdvised アノテーション
Fig. 5　@AssertAdvised annotation for a mehtod.

```
1 @SolveProblem("A.name_of_problem")
2 void around(): call(* *.bar()) {
3   //do something
4 }
```

図 6　advice への@SolveProblem アノテーション
Fig. 6　@SolveProblem annotation for advice.

テーションを持つクラス A のメソッドに関して検査を行う．ここでいう影響を与えているとは，@SolveProblem アノテーションを持つ advice と@AssertAdvised を持つメソッドとの間に，一方が他方を呼び出す可能性があるか，ということである．たとえば，図 5 と図 6 では，foo メソッドの中で advice が適用されている bar メソッドの呼び出しが起こるので影響を与えている，といえる．また，

```
1 @SolveProblem("A.name_of_problem")
2 void around(): execution(* A.foo()) {
3   proceed();
4 }
```

と記述した場合，proceed 呼び出しを通して，foo メソッドが呼ばれるので，これも影響を与えている，といえる．

assertion for advice の検査では，advice が指定した join point に対し適用されているかという単純な検査ではなく，advice が間接的に影響を与えているかどうかも検査する．これは，*extract method*[⋆1]などのリファクタリングにより，同期処理の advice が適用されている領域が異なるメソッドとして抽出される場合が考えられるためである．このようなコードの意味を変えずに構造だけが変更されている場合には，検査を通す必要があるからである．単純な検査では，そのような場合も検査器が警告を発してしまう．また異なる粒度の同期処理の advice を記述した場合，各粒度の advice が同じメソッド内に適用されるとは限らない．粗い粒度の同期処理は，同期が必要な領域を間接的に呼び出している別のメソッド内に適用される場合も存在する．たとえば，a()，b()，c() のメソッドがあり，a() が b() を，b() が c() を呼び出しているとする．そして，本当に同期処理が必要なメソッドが c() であるとすると，同期処理は c()，b()，a() のどこで行ってもよい．そのような場合には，アノテーションが付いているメソッドのみを検査対象としては警告が出てしまう．これに対処するため，直接対象となるメソッドから，直接・間接に呼び出されたメソッドを含んだ範囲のどこかで，advice が影響を与えているかどうか，静的な検査を行う．

@AssertAdvised アノテーションはサブクラスでオーバライドするメソッドには継承されない．サブクラスのメソッドでも advice が適用される必要があると宣言したい場合は，メソッド，advice ともに明示的にアノテーションを付けなければならない．advice が適用される必要があるかどうかは実装によるので，あるメソッドで advice が必要でも，それをオーバライドするメソッドでは advice を適用する必要がないかもしれない．そのため，サブクラスでは必要に応じて@AssertAdvised を明示的に宣言する必要がある．

⋆1 メソッドの一部を別の（特に private な）メソッドに分割，抽出するリファクタリング方法

5. 実　　装

5.1 Regioncut

我々は regioncut を，*the AspectBench Compiler*（abc）[4] 1.3.0 JastAdd フロントエンド版の拡張として，AspectJ 上に実装した．3.2 節で述べたセマンティクスでの領域と regioncut のマッチングは，静的に評価することができる．マッチングをコンパイル時に行い，プログラム変換により advice の織り込みを行うことで，実行時のオーバヘッドを削減した実装とした．abc の内部で使用している中間言語の Jimple[13] 上で領域のマッチングを行うことで regioncut を利用したアスペクトの織り込みを行う．

ブロックと文の解析

3.2 節で述べたセマンティクスに従って織り込みを行う場合，制御構造やブロック，文の構造を知っている必要がある．しかし，Jimple 上にはそれらの区切りの情報は残っていない．そこで，制御構造，ブロック，文の区切りの位置を，構文木から Jimple への変換の際に保存するようにコンパイラを変更した．

我々は，Jimple に**マーカ**という命令を追加した．マーカは，文，ブロック，if，while，synchronied などの構造の種類と，それが開始位置か終了位置か，という 2 つの情報を持つ．このマーカを AspectJ の構文木から Jimple へ変換する際に，適切な位置に挿入する．このマーカを利用して文や制御構造の区切りを考慮して織り込みを行う．マーカは織り込みが終

了した後，Java バイトコードに変換される前にすべて取り除かれる．

5.2 Assertion for Advice

我々は，assertion for advice も the AspectBench Compiler 上に実装した．assertion for advice の検査はコンパイル時に静的に行われる．

検査をするために，コンパイル時に Jimple 中間言語を走査し，コールグラフを生成する．@SolveProblem アノテーションが付いた advice が@AssertAdvised アノテーションが付いたメソッドを呼び出す可能性があるか，逆に呼び出される可能性があれば，検査は通過する．そうでなければ，警告を出す．

6. 評　　価

regioncut と assertion for advice のデザインを評価するために，我々は 2 つのオープンソースソフトウェアに適用して評価を行った．評価に用いたコンパイラは 5 章のものである．

6.1 Javassist

我々は，2.1 節で述べた Javassit のそれぞれの粒度の同期処理を，regioncut を利用してアスペクトとして記述した．**図 7** は，Javassist に細かい粒度で行う同期処理を手で埋め込んだものである．createClass2 メソッドの中で 2 回に分けて synchronized 文で同期処理を行っている．**図 8** は粗い粒度の同期処理を手で埋め込んだものであり，図 7 の createClass2 内で同期処理を行う代わりに，そのメソッドを呼び出す createClass メソッドで同期処理を行う．createClass メソッドは先述の createClass2 メソッドを呼び出していて，その周りで 1 つの synchronized 文で同期処理を行っている．regioncut を使うことで，これらの同期処理をアスペクトとして記述することができた．**図 9** と **図 10** はそれぞれ，細かい粒度と粗い粒度の同期処理を実装した advice のコードである．

regioncut を実行した場合の性能を比較するため，2.1 節で述べたバグレポート[1] を基にしたベンチマークを，粗い粒度と細かい粒度の同期を，それぞれ regioncut で実装した場合とハンドコーディングした場合とで，性能を比較した．ベンチマークで行った処理は，2000 個のスレッドを起動し，それぞれのスレッドが図 8 で示した createClass メソッドの呼び出しを含むメソッドを 1000 回呼び出す，というものである．実行した環境は，Ubuntu 9.04（Linux 2.6.28），Intel Xeon CPU 2.83 GHz（2 コア），8 GB メモリ，Sun JVM 1.6.0 である．このベンチマークをそれぞれの条件で 100 回繰り返したときの実行時間の平均値と標準偏差は**表 1** のようになる．この結果を見ると，regioncut を用いて同期処理を実装してもそれほど大きなオーバヘッドがないことが分かる．このベンチマークでは，オーバヘッドを見るために通信などを排除して実験を行った．この例では同期の粒度の違いによる性能に差が見られなかったが，論文 2) の異なるハードウェアによる実験では粗い粒度の同期処理により性能の向上が見られた．

```
1  private static WeakHashMap proxyCache;
2  private void createClass2(ClassLoader cl) {
3    CacheKey key = new CacheKey(superClass, interfaces, methodFilter, handler);
4    synchronized (proxyCache) {
5      HashMap cacheForTheLoader = (HashMap)proxyCache.get(cl);
6      if (cacheForTheLoader == null) {
7        cacheForTheLoader = new HashMap();
8        proxyCache.put(cl, cacheForTheLoader);
9        cacheForTheLoader.put(key, key);
10     } else {
11       CacheKey found = (CacheKey)cacheForTheLoader.get(key);
12       //中略
13     }
14   }
15
16   synchronized (key) {
17     Class c = isValidEntry(key);
18     if (c == null) {
19       createClass3(cl);
20       key.proxyClass = new WeakReference(thisClass);
21     }else{ thisClass = c; }
22   }}
```

図 7　Javassist での細かい粒度の同期処理
Fig. 7　Fine-grained synchronization in Javassist.

```
1  public Class createClass() {
2    if (thisClass == null) {
3      ClassLoader cl = getClassLoader();
4        synchronized (proxyCache) {
5          if (useCache)
6            createClass2(cl);
7          else
8            createClass3(cl);
9      }}
10   return thisClass;
11 }
```

図 8　Javassist での粗い粒度の同期処理
Fig. 8　Coarse-grained synchronization in Javassist.

```
1  void around():
2    region[ call(* WeakHashMap.get(..)), call(* WeakHashMap.put(..)) ]
3  {
4    synchronized(ProxyFactory.class){ proceed(); }
5  }
6
7  void around(Object key):
8    region[ call(* *.isValidEntry(*)) && args(key), set(* *.proxyClass) ]
9  {
10   synchronized(key){ proceed(key); }
11 }
```

図 9　細かい粒度の同期処理を行う advice
Fig. 9　Advice for fine-grained synchronization.

```
1  void around():
2    region[ get(static boolean *.useCache), call(* *.createClass2(..)) ]
3  {
4    synchronized(ProxyFactory.class){ proceed(); }
5  }
```

図 10　粗い粒度の同期処理を行う advice
Fig. 10　Advice for coarse-grained synchronization.

表 1　Javassist のベンチマークの実行時間
Table 1　The execution time of the Javassist benchmark.

	実行時間（秒）	標準偏差
細かい粒度（アスペクト）	41.8	3.3
細かい粒度（ハンドコーディング）	41.3	2.9
粗い粒度（アスペクト）	41.2	3.1
粗い粒度（ハンドコーディング）	41.9	2.8

6.2　Hadoop

我々は，regioncut が十分に同期処理に関する領域を選択できるか，および，適用されなくなった advice を assertion for advice が発見できるかを確認するために，オープンソースの分散アプリケーションのためのフレームワークである Hadoop [11] にの同期処理をアスペクトに分離した．この際，Hadoop バージョン 0.16.4 の一部を，AspectJ および我々の提案する言語機構を用いた．

表 2　TaskTracker クラス内の，同期処理の関心事
Table 2　Synchronization concerns in TaskTracker class.

選択された領域の種類	数
synchronized 文	21
従来の pointcut で分離できたもの	9
regioncut で分離できたもの	12

```
1  @AssertAdvised("removeTaskFromJob")
2  private void removeTaskFromJob(String jobId, TaskInProgress tip) {
3    synchronized (runningJobs) {
4      RunningJob rjob = runningJobs.get(jobId);
5      if (rjob == null) {
6        LOG.warn("Unknown␣job␣" + jobId + "␣being␣deleted.");
7      } else {
8        //以下が synchronized 文の中が選択対象の領域
9        //synchronized (rjob) {
10         rjob.tasks.remove(tip);
11         if (rjob.tasks.isEmpty()) {
12           runningJobs.remove(jobId);
13         }
14       //}
15     }}}
16
17 @SolveProblem("org.apache.hadoop.mapred.TaskTracker.removeTaskFromJob")
18  void around(Object t):region[
19      get(java.util.Set *.tasks) && target(t),
20      call(* java.util.Map+.remove(*)),
21    ]
22 {
23   synchronized(t){ proceed(t); }
24 }
```

図 11　Hadoop において同期処理をアスペクトで記述した際の対象のメソッドと advice
Fig. 11　An example of a method and advice for synchronization in Hadoop.

6.2.1　regioncut による同期処理の分離

まず我々は，Hadoop の中に含まれる TaskTrakcer クラス（2357LOC）内の，synchronized 文によって行われている同期処理を regioncut を用いてアスペクトに分離した．

表 2 は分離した結果である．TaskTrakcer クラスの中には 21 個の synchronized 文が存在し，そのうち 9 個は従来の AspectJ の pointcut（call，get，set など）でアスペクトに分離することができた．残りの 12 個は，従来の pointcut ではアスペクトに分離できなかったが，それらはすべて regioncut を用いることで分離することができた．**図 11** は Hadoop で

```
1   @AssertAdvised("removeTaskFromJob")
2   private void removeTaskFromJob(JobID jobId, TaskInProgress tip) {
3     synchronized (runningJobs) {
4       RunningJob rjob = runningJobs.get(jobId);
5       if (rjob == null) {
6         LOG.warn("Unknown␣job␣" + jobId + "␣being␣deleted.");
7       } else {
8         // 以下の synchronized 文が対象の領域
9         //synchronized (rjob) {
10          rjob.tasks.remove(tip);
11        //}
12      }}}
```

図 12　ソースコードの更新により選択できなくなった領域
Fig. 12　A region that regioncut cannot select after modifying source code.

regioncut を用いて同期処理をアスペクトとして記述した場合の対象のメソッドと advice の例である．

また，regioncut の引数に 3 つ以上の pointcut 指定子を与えなければならなかったコード領域は，3 カ所存在した．これにより，領域の始点と終点だけでなく，中間の join point を指定することができる regioncut のデザインは妥当であることが分かる．

6.2.2　Assertion for Advice の適用

assertion for advice がうまく効果を発揮するかを評価するために，6.2.1 項において regioncut を適用した Hadoop 0.16.4 の TaskTracker クラスのメソッドと，同期処理を実装した advice に，@AssertAdvised アノテーションと@SolveProblem アノテーションを付加した．そして，Hadoop のバージョンを 0.18.3 に更新し，0.16.4 と同じメソッドに，@AssertAdvised アノテーションを付加し，advice が適用されているかの検査を行った．

その結果，適用されなくなった advice が 1 つ存在することが，assertion for advice により発見された．バージョン 0.16.4 では図 11 のとおりであった removeTaskFromJob メソッドは 0.18.3 では**図 12** のようになり，regioncut で領域を選択できなくなった．

また，リファクタリングが行われ，別のメソッドに移動された領域が 1 つ存在した（**図 13**, **図 14**）．バージョン 0.16.4 の getMapCompletionEvents(String, int, int) メソッド（図 13）の本体は，引数の型が異なる getMapCompletionEvents(JobID, int, int) メソッドに移動された．そして，元の getMapCompletionEvents(String, int, int) メソッドは，移動されたメソッドを，引数の型を変換して呼ぶようにリファクタリングされている（図 14）．このリファクタリングに対して警告は出なかった．これにより，assertion for advice が間接的な

```
1   @AssertAdvised({"getMapOuter","getMapInner"})
2   public TaskCompletionEvent[]
3         getMapCompletionEvents(String jobId, int fromEventId, int maxLocs) {
4     TaskCompletionEvent[]mapEvents = TaskCompletionEvent.EMPTY_ARRAY;
5     RunningJob rjob;
6     synchronized (runningJobs) {
7       rjob = runningJobs.get(jobId);
8       if (rjob != null) {
9         synchronized (rjob) {
10          FetchStatus f = rjob.getFetchStatus();
11          if (f != null) { mapEvents = f.getMapEvents(fromEventId, maxLocs); }
12        }}}
13    return mapEvents;
14  }
```

図 13　リファクタリングが行われる前の getMapCompletionEvents メソッド
Fig. 13　getMapCompletionEvents method before refactoring.

```
1   @AssertAdvised({"getMapOuter","getMapInner"})
2   public TaskCompletionEvent[]
3         getMapCompletionEvents(String jobid, int fromid, int maxlocs) {
4     return getMapCompletionEvents(JobID.forName(jobid), fromid, maxlocs);
5   }
```

図 14　リファクタリングが行われた後の getMapCompletionEvents メソッド
Fig. 14　getMapCompletionEvents method after refactoring.

表 3　TaskTracker クラスのコンパイルにかかった時間
Table 3　Compile time of TaskTracker class.

処理の種類	実行時間（秒）
コンパイル全体	14.81
検査	0.061

呼び出し関係を考慮して検査する機構がうまく働いていることが分かった．

6.2.3　Assertion for Advice の検査の性能

6.2.1 項で同期処理をアスペクトに分離した，Hadoop 0.16.4 の TaskTrakcer クラスと，同期処理を実装したアスペクトに assertion for advice のアノテーションを付けた状態で，コンパイルと検査を行った．実験環境は，6.1 節と同様である．コンパイルを 10 回行ったときの平均をとった結果は，**表 3** となる．コンパイル全体にかかる時間のうち，検査にかかる時間の占める割合は，0.41%となることから，検査のオーバヘッドは十分に許容できる

レベルであるといえる．

7. 関連研究

Declarative event patterns[14] と *Tracematch*[3] は実行時の履歴に基づいた pointcut を提供している．メソッドがどのような順番で実行されたかという履歴と，pointcut で指定したパターンが合致した場合に，その時点で advice が実行される．これらは，実行済みの履歴に対して pointcut が合致するか調べるため，同期処理を必要とする領域の前後で advice を実行することができない．そのため，これらの pointcut では同期処理などのコード領域を対象とする関心事をアスペクトとして記述することは難しい．

LoopsAJ[6] は，ループを join point として選択できる pointcut を提供している．これは，ループを並列に動作させることなどに利用できる．一方，ループのみを対象としているため，同期処理などの関心事には用いることができない．

synchronized block join point[15),16)] では Java の synchronized 文を選択する pointcut を提供している．これを用いることで，同期処理の方法を，synchronized 文の代わりに Lock オブジェクトを使う，などといったようにアスペクトを用いて変更することができる．しかし，粒度の違う同期処理をアスペクトで実装することはできない．

Transactional Pointcuts[9] は，regioncut と同様にコード領域を join point と同じように選択する pointcut であり，本稿の前身となる論文 2) と同時に，2009 年の国際会議 GPCE'09（Generative Programming and Component Engineering）に採択されたものである．join point の列を指定し，その join point の列が含まれる領域を選択する pointcut という点では，regioncut と同じであるが，合致する join point の一部が if 文や while 文の中にあり，実行されない可能性がある場合はその領域は選択されない．regioncut では，合致する join point が if 文や while 文の中に存在しても，実行される可能性があるものとして，その領域を選択する．一方，transactional pointcuts では，regioncut と異なり指定した join point の列に対し，メソッド呼び出しの実引数の 1 つが，別のメソッド呼び出しのある実引数と同じオブジェクトでなければならない，といった制約条件を付けることができる．また，transactional pointcuts には，assertion for advice のような，fragile pointcut 問題に対処するための機構は存在しない．

8. まとめ

本稿で我々は，コード領域に関する関心事，特に同期処理，をアスペクトに分離して記述するために，2 つの言語機構をアスペクト指向言語 AspectJ に導入した．*regioncut* により従来の AspectJ よりも多くのコード領域を選択し，その領域で advice を実行することができる．*assertion for advice* は，advice が期待する箇所に適用されていないことを静的に検出するための言語機構である．

我々は，regioncut を Javassist に適用し，粒度の異なる同期処理をアスペクトとして分離記述できることを示した．また，Hadoop の中の同期処理をアスペクトとして記述した．そして，Hadoop のバージョンを更新することで適用されなくなってしまった advice を assertion for advice を用いることで検出できることを示した．

今後の課題としては，異なる regioncut が部分的に重なった領域を選択した場合の問題に対処することがあげられる．重なった一方の領域を自動的に拡大することで，around advice をそのような領域に適用することができると考えられる．このとき，どの領域を拡大するか判断するルールを決めるか，プログラマが拡大するべき領域の優先度を指定できるようにする，などの方策をとる必要がある．また，assertion for advice を統合開発環境に組み込むことによって，advice の欠如を容易にかつ即座に発見できるようにすることで，regioncut および assertion for advice の実用性を向上させることも課題である．

参考文献

1) [#JASSIST-28] javassist enhancement failed on deserializing hibernate proxies - jboss.org JIRA. http://jira.jboss.org/jira/browse/JASSIST-28
2) Akai, S. and Chiba, S.: Extending AspectJ for separating regions, *GPCE '09: Proc. 8th international conference on Generative programming and component engineering*, New York, NY, USA, pp.45–54, ACM (2009).
3) Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G. and Tibble, J.: Adding trace matching with free variables to AspectJ, *OOPSLA '05: Proc. 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages and applications*, New York, NY, USA, pp.345–364, ACM (2005).
4) Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G. and Tibble, J.: abc: an extensible AspectJ compiler, *AOSD '05: Proc. 4th international conference on Aspect-oriented software development*, New York, NY, USA, pp.87–98, ACM (2005).
5) Chiba, S.: Load-Time Structural Reflection in Java, *ECOOP '00: Proc. 14th European Conference on Object-Oriented Programming*, pp.313–336, London, UK, Springer-Verlag (2000).

6) Harbulot, B. and Gurd, J.R.: A join point for loops in AspectJ, *AOSD '06: Proc. 5th international conference on Aspect-oriented software development*, New York, NY, USA, pp.63–74, ACM (2006).
7) Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An Overview of AspectJ, *ECOOP '01: Proc. 15th European Conference on Object-Oriented Programming*, pp.327–353, London, UK, Springer-Verlag (2001).
8) Kourai, K., Hibino, H. and Chiba, S.: Aspect-oriented application-level scheduling for J2EE servers, *AOSD '07: Proc. 6th international conference on Aspect-oriented software development*, New York, NY, USA, pp.1–13, ACM (2007).
9) Sadat-Mohtasham, H. and Hoover, H.J.: Transactional pointcuts: Designation reification and advice of interrelated join points, *GPCE '09: Proc. 8th international conference on Generative programming and component engineering*, New York, NY, USA, pp.35–44, ACM (2009).
10) Stoerzer, M. and Graf, J.: Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software, *ICSM '05: Proc. 21st IEEE International Conference on Software Maintenance*, Washington, DC, USA, pp.653–656, IEEE Computer Society (2005).
11) The Apache Software Foundation: Welcome to Apache Hadoop!. http://hadoop.apache.org/
12) The Eclipse Foundation: AspectJ Development Tools (AJDT). http://www.eclipse.org/ajdt
13) Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P. and Sundaresan, V.: Soot — a Java bytecode optimization framework, *CASCON '99: Proc. 1999 conference of the Centre for Advanced Studies on Collaborative research*, p.13, IBM Press (1999).
14) Walker, R.J. and Viggers, K.: Implementing protocols via declarative event patterns, *SIGSOFT '04/FSE-12: Proc. 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, New York, NY, USA, pp.159–169, ACM (2004).
15) Xi, C., Harbulot, B. and Gurd, J.R.: A synchronized block join point for AspectJ, *FOAL '08: Proc. 7th workshop on Foundations of aspect-oriented languages*, New York, NY, USA, pp.39–39, ACM (2008).
16) Xi, C., Harbulot, B. and Gurd, J.R.: Aspect-oriented support for synchronization in parallel computing, *PLATE '09: Proc. 1st workshop on Linking aspect technology and evolution*, New York, NY, USA, pp.1–5, ACM (2009).

(平成 22 年 7 月 8 日受付)
(平成 22 年 11 月 14 日採録)

赤井　駿平

2008 年東京工業大学理学部情報科学科卒業．2010 年同大学大学院情報理工学研究科数理・計算科学専攻修士課程修了．現在，同専攻博士課程に在学中．プログラミング言語およびプログラムのモジュール化手法の研究に従事．

千葉　　滋（正会員）

1991 年東京大学理学部情報科学科卒業．1996 年同大学大学院理学系研究科情報科学専攻博士課程退学．東京大学助手，筑波大学講師を経て，現在，東京工業大学大学院情報理工学研究科教授．博士（理学）．システムソフトウェアの研究に従事．