プログラムの領域をポイントカットすることが可能なアスペクト指向言語

赤井 駿平 † 千葉 滋 †
Shumpei AKAI Shigeru CHIBA

† 東京工業大学大学院情報理工学研究科
Graduage School of Information Science and Engineering, Tokyo Institute of Technology
akai@csg.is.titech.ac.jp chiba@is.titech.ac.jp

本稿では，プログラム中の領域をポイントカットすることが可能な AspectJ 言語の拡張を提案する．既存のアスペクト指向言語ではプログラム中のメソッド呼び出しや，フィールドへのアクセスなどの実行点しかポイントカットすることができない．そのため，同期処理や例外処理をアスペクトに分離しようとする際，うまく範囲を指定できなかった．提案する言語拡張では，パターンマッチを用いて，言語の構造を考慮した上で領域のポイントカットすることを可能にした．

1 はじめに

既存のアスペクト指向言語では，メソッド呼び出しやフィールドへのアクセスなどの実行点の振る舞いを変更することが出来る．しかし，複数の処理が並んだようなプログラム中の領域の振る舞いを変更するための機能が存在しない．

例えば，同期処理を行う際には同期の粒度，すなわち同期を行う領域の広さによって性能が変化する．通常は粒度の細かい同期を行った方が，並列な実行が行いやすくなり，性能が向上する．しかし，負荷が高い場合などは，計算機の環境によっては粒度の粗い同期を行った方が良い性能を示す場合がある．同期処理をアスペクトに分離して記述できれば，そのような粒度の切り替えが行いやすくなる．同期処理の対象となるコードは一般にメソッド中の一部分である．そのため，アスペクトを用いてメソッドの一部の振る舞いを変更して同期処理を行うには，プログラム中の領域をポイントカットできる必要がある．

本稿では，プログラムの領域をポイントカット可能なアスペクト指向言語を提案する．これは，アスペクト指向言語である AspectJ[7] の言語拡張として提供される．ポイントカットしたい領域の指定には，領域の開始・終了位置となる join point 及び，領域に含まれなければならない join point，含まれてはいけない join point を記述することで行う．

我々は領域をポイントカットするアスペクト指向言語の拡張を，abc[1] を拡張することで実装した．また，Javassist[9] の同期処理に適用することで，実際に同期処理の粒度の切り替えに本言語拡張を利用できることを示す．

以下では，2 章では，領域をポイントカットできないことによる問題点を示し，3 章では領域をポイントカットするための手法，4 章ではその実装について述べる．5 章では本言語拡張の応用例について述べる．6 章では関連研究を挙げ，7 章でこれらをまとめる．

2 AspectJ における問題点

AspectJ でポイントカットすることが出来る join point には，メソッドの呼び出し (*call*)，メソッドの実行 (*execution*)，フィールドの参照 (*get*)，ィールドへの代入 (*set*)，例外ハンドラの実行 (*handler*)，が存在する．AspectJ では，これらのような特定の実行点のみが join point となるため，プログラム中の非常に限られた場所しかポイントカットすることができない．そのため，同期処理のような処理をアスペクトを用いて記述することが難しい．

例えば，図 1 のリストの各要素の値を累乗させるメソッド mapPow が並列に実行されていた場合，5〜7 行目がクリティカルセクションとなっている．そのため，この領域を同期するという処理をアスペクトを用いて記述しようとしても，AspectJ では，この領域だけをポイントカットすることが出来ない．

before アドバイスと after アドバイスを用いて，5 行目の get メソッドの呼び出しの直前でロックを取得し，7 行目の set メソッドの呼び出しの直後でロックを解放すれば，同期処理を行うことが出来るよう

に見える．しかし，6 行目 convert メソッドの実行中に例外が発生した場合，ロックが解放されないままこのメソッドを終了してしまうという問題が起こる．そのため，mapPow メソッドの実行全体をポイントカットして同期を行うことしか出来ない．

```
1  void mapPow(List list,double exponent){
2      ...
3      for(int i=0;i<list.size();i++){
4          double tmp;
5        /*begin critical section*/
6          tmp=((Double)list.get(i)).doubleValue
               ();
7          tmp=Math.pow(tmp,exponent)
8          list.set(i,new Double(tmp));
9        /*end of critical section*/
10     }
11     ...
12 }
```

図 1: 同期の必要なプログラム

3 領域に対するポイントカット

我々は，プログラム中の領域をポイントカットするための，AspectJ 言語の拡張を提案する．この言語拡張では，ポイントカットを行いたい領域を選択するためのポイントカット指定子を導入する．

3.1 区間の指定

ポイントカットを行いたい領域を指定するには，*begin* ポイントカット，及び *end* ポイントカットを使用する．begin では領域の開始位置となる join point を，end では終了位置となる join point を指定する．

```
1  pointcut criticalSection() :
2      begin(call(public List.get(*))) &&
3      end(call(public List.set(*)));
```

図 2: get〜set までのポイントカット

例えば，図 1 の get メソッドの呼び出しから set メソッドの呼び出しまでをポイントカットするには，図 2 のようにポイントカット指定子を記述することで実現できる．図 3 に対し，図 2 の記述でポイントカットを行ったときのことを考える．この時，領域の開始位置が if 文の条件が真の場合に実行されるブロック中に存在しているが，終了位置は if 文の外に存在する．このメソッドに対し図 2 のポイントカットを

```
1  void mapPow(List list,double exponent){
2      ...
3      for(int i=0;i<list.size();i++){
4        /*begin critical section*/
5          double tmp;
6          if(i%2 == 0){
7              tmp=((Double)list.get(i)).
                   doubleValue();
8              tmp=Math.pow(tmp,exponent)
9          }else{
10             tmp=getDefaultValue();
11         }
12         list.set(i,new Double(tmp));
13       /*end of critical section*/
14     }
15     ...
16 }
```

図 3: 同期の必要なプログラム

用いて，get メソッドの真に直前にロックを獲得するアドバイスを，set メソッドの直後にロックを解放するアドバイスを挿入すると，偽の場合のブロックが実行された場合にはロックが取得されていないのに解放しようとしてエラーとなってしまう．

本言語拡張では，開始・終了位置となる join point が異なるブロックに存在する場合には，ブロックを遡り両方を共に含むブロックを探す．そして，そのブロック直下の，開始位置を含むブロック/文から，終了位置を含むブロック/文までをポイントカットする．図 3 では，if 文の直前から，set メソッドの直後までがポイントカットされる．

このように，ポイントカットする領域を開始・終了位置の存在するブロックに応じて自動的に拡大を行うことで，ソースコード中の制御構造やブロック構造と，ポイントカットされる領域が矛盾を起こさなくなる．

3.1.1 領域への条件の指定

領域を指定する際に，開始・終了位置の 2 点だけで指定すると，1 つのメソッド中に開始・終了位置それぞれににマッチする join point が複数存在する場合に問題となる．図 4 のプログラムに対し，図 2 を適用すると，6〜8 行目と 12〜14 行目の両方がポイントカットされる．このポイントカットに対しアドバイスを記述すると，両方の領域に対してアドバイスを実行してしまう．そのため，メソッド中の特定の特定の (例えば最初の) 領域だけでアドバイスを実行することが出来ない．

本言語拡張では，複数の領域を区別するために，

```
1  void mapPow(List list,double exponent){
2      ...
3      for(int i=0;i<list.size();i++){
4          double tmp;
5        /*begin critical section*/
6          tmp=((Double)list.get(i)).doubleValue
               ();
7          tmp=Math.pow(tmp,exponent);
8          list.set(i,new Double(tmp));
9        /*end of critical section*/
10     }
11     ...
12     String str=(String)stringList.get(0);
13     str+="_";
14     stringList.set(0,str);
15
16 }
```

図 4: 終了位置と複数の join point がマッチ

include ポイントカット，*exclude* ポイントカットを提供する．include ポイントカットでは，選択される領域中に含まれるべき join point を記述する．また，exclude ポイントカットでは，選択される領域中に含まれてはいけない join point を記述する．図 4 の for ループの中の get メソッドから set メソッドまでをポイントカットする場合は，次のように記述すればよい．

```
1 pointcut criticalSection() :
2     begin(call(public List.get(*))) &&
3     end(call(public List.set(*))) &&
4     include(call(Math.pow(*)));
```

3.2 変数の取得

アスペクトを用いてアドバイス側で同期処理を行おうとする場合，アドバイスの中で同期処理の対象となるオブジェクトにアクセスできる必要がある．そのオブジェクトが，ポイントカットする領域を含むメソッドのクラスのフィールドに格納されているならば，this ポイントカットを利用して目的のフィールドを持つオブジェクトを取得し，そのフィールドにアクセスすればよい．しかし，同期処理を行いたいオブジェクトがローカル変数に格納されている場合は，AspectJ の既存のコンテキスト渡しの機構を利用して取得することが出来ない．ローカル変数の値をアドバイスで取得する場合，どのローカル変数を取得するかを指定しなければならない．しかし，ローカル変数をその名前で指定すると，ローカル変数の名前が変更された場合，それまで取得できていた変数が取得できなくなってしまう．

本言語拡張では，取得したいローカル変数の型を利用して指定する．begin・end ポイントカット指定子と共に，型のみを記述した args ポイントカットを用いることで，取得したいローカル変数を定める．図 1 のプログラムのローカル変数 list を取得して，それに対して同期を行う例は以下のようになる．

```
1 void around(List l):
2     begin(call(public List.get(*))) &&
3     end(call(public List.set(*))) &&
4     args(l) && args(List)
5     {
6         synchronized(l){
7             proceed(l);
8         }
9     }
```

このように記述することで，ローカル変数 list がアドバイスの引数 l として取得できる．

4 実装

本言語拡張は，AspectJ 言語のコンパイラの実装の 1 つである，AspectBench Compiler[5](abc) を拡張することで実装した．abc では，バックエンドとして Soot[4] を利用している．Soot には Java バイトコードの中間表現である Jimple が含まれており，abc でのアスペクトの織り込みは Jimple 上で行われている．そのため，本言語拡張において，条件にマッチする領域の探索や書き換えは Jimple 上で行う．

4.1 領域の探索

最初に領域の探索を行う手順を述べる．まず，命令列を制御構造やブロック構造毎に分割し，木構造を作る．しかし，Jimple の命令列にはソースコード上のブロックの開始・終了位置を示す情報は残っていない．そのため本実装では，制御構造や，ブロック構造の開始・終了位置を示す Jimple の新たな命令を追加し，抽象構文木から Jimple へのコンパイル時に，開始・終了位置の情報を記録するようにした．

ブロックの木構造が作成した後，まず，begin，end，include で指定されたポイントカットにマッチする join point を全て含む最小のブロックを探す．そして，そのブロックに対し，以下のこと調べる．

- begin にマッチするブロックが end にマッチするブロックの前に存在する
- begin にマッチするブロックから end にマッチするブロックの間に存在するブロックに，include にマッチする join point が含まれている

- begin にマッチするブロックから end にマッチするブロックの間に存在するブロックに，exclude にマッチする join point が含まれない

この 3 つの条件に適合した場合，begin にマッチするブロックの先頭の命令から，end にマッチするブロックの末尾の命令までをポイントカットする．

4.2 命令列の書き換え

abc では，around アドバイスを適用する場合，ポイントカットされた join point をコピーし，新たな static メソッドを作成する．そして，アドバイスの中で proceed() の呼び出を行った際には，そのコピーされたメソッドを呼び出すようになる．

メソッドの一部を単純にコピーして別のメソッドを作った場合，元のメソッドで宣言されているローカル変数をコピーされた join point 内でアクセスする場合に問題が起きる．abc では，コピー元で宣言されていて，コピーされた join point で参照しているローカル変数は，コピーされたメソッドの引数として渡されるようになる．しかし，ローカル変数への代入が存在した場合，元のローカル変数へ反映されない．また，メソッド内の一部を別のメソッドにコピーしているため，join point の中から外へのジャンプを行うことが出来ない．本言語拡張の実装では，これらの問題に対処するための命令列の書き換えを行う．

ローカル変数の代入に対処するにはまず，join point の外で宣言されていて，join point 内で代入を行っているローカル変数を探す．そして，それらのローカル変数と同じ型・名前のフィールドを持つ，ローカル変数を格納するためのクラスを生成する．join point の直前でそのクラスのインスタンスを生成し，join point をコピーしたメソッドの引数として渡すようにする．そして，join point の直前と，join point 内の末尾でローカル変数をそのインスタンスのフィールドへ格納し，join point の先頭と直後では，フィールドからローカル変数へ代入を行う．

join point の外へのジャンプを行うには，まずそれぞれのジャンプ先に対し，それらを識別するための番号を定める．そして，ジャンプ先を示すローカル変数を用意し，それぞれのジャンプ命令を「ジャンプ先を示す変数へ対応するジャンプ先の番号を代入し，join point の末尾へジャンプする」という命令列に置き換える．join point を抜けた直後の場所では，変数に格納されたジャンプ先の番号を調べ，対応するジャンプ先へジャンプする．ここで用意したローカル変数は，先述のローカル変数への代入に対する処理により join point の内外で共有されているため，このようにすることで join point の外へのジャンプを行うことが出来る．

5 応用例

5.1 領域のポイントカットの同期処理への適用

我々は，Java バイトコードの編集を行うライブラリである Javassist における同期処理を，本言語拡張を用いてアスペクトで記述した．

Javassist では，同期処理に関するバグ [3] を修正する際に，細かい粒度で同期を行った場合に粗い粒度の場合に比べて遅くなることがあることが分かったため，粗い粒度で同期を行うことにしている．通常，細かい粒度の同期を行った方が同時実行性が良くなり，並列に実行した際の性能が向上すると思われる．しかし，負荷が高い状態の時は，コンテキストスイッチ等のコストが大きくなり，性能が悪化する．

このように，システムの環境や利用状況により最適な同期の粒度が異なる．そこで，提案した言語拡張を用いて，同期処理をアスペクトに分離し，同期の粒度を切り替えることができるようにした．

```
1  public Class createClass() {
2      if (thisClass == null) {
3          ClassLoader cl = getClassLoader();
4          //(A)
5              if (useCache)
6                  createClass2(cl);
7              else
8                  createClass3(cl);
9          //(A')
10     }
11     return thisClass;
12 }
```

図 5: 粗い粒度の同期を行うメソッド

図 5 と 6 は Javassist の ProxyFactory クラスの一部である．粗い粒度で同期を行う際は図 5 の (A)〜(A') で同期を行い，細かい粒度で同期を行う際は図 6 の (B)〜(B') と (C)〜(C') を同期する．

この時，粗い粒度で同期を行うためのアドバイスは図 7 のように，細かい粒度で同期を行うためのアドバイスは図 8 のように記述できる．

```
1  private static WeakHashMap proxyCache;
2  private void createClass2(ClassLoader cl){
3      CacheKey key = new CacheKey(...);
4      //(B)
5          HashMap cacheForTheLoader =
6              (HashMap)proxyCache.get(cl);
7          if (cacheForTheLoader == null) {
8              ...
9              proxyCache.put(...);
10             ...
11         }else{ ... }
12     //(B')
13
14     //(C)
15         Class c = isValidEntry(key);
16         if (c == null) {
17             createClass3(cl);
18             key.proxyClass =
19                 new WeakReference(thisClass);
20         }else
21             thisClass = c;
22     //(C')
23 }
```

図 6: 細かい粒度の同期を行うメソッド

```
1  void around():
2      begin(call(* *.createClass2(..)))&&
3      end(call(* *.createClass3(..))) &&
4      withincode(* ProxyFactory.createClass
           (..))
5  {
6      synchronized(ProxyFactory.class){
7          proceed();
8      }
9  }
```

図 7: 粗い粒度の同期を行うアドバイス

5.2 実験

先述の Javassit の 2 種類の同期の粒度を，アスペクトを用いて切り替えて実行時間の計測を行った．

Javassist のバージョンは 3.4 を対象とし，実行時間を計測するために，Javassist のバグレポート [3] に投稿されたテストプログラム (TestJavassist.tgz) を利用した．このテストプログラムでは，クライアントがサーバからデータを受け取る際に，同期処理が必要となる．ここでは，クライアント側の，CPU のコア数を変化させその実行時間の差を調べる．

以下の実験用計算機を 2 台用意し，一方をサーバー，他方をクライアントとし，テストプログラムを実行する．

CPU Intel Xeon CPU 5160 3.00GHz 最大 4 コア

メモリ 2GB

```
1  void around():
2      begin(call(* WeakHashMap.get(..))) &&
3      end(call(* WeakHashMap.put(..))) &&
4      withincode(* ProxyFactory.createClass2
           (..))
5  {
6      synchronized(ProxyFactory.class){
7          proceed();
8      }
9  }
10
11 void around(ProxyFactory.CacheKey key):
12     begin(call(* *.isValidEntry(..))) &&
13     end(call(WeakReference.new(..))) &&
14     exclude(call(* HashMap.put(..))) &&
15     args(ProxyFactory.CacheKey) &&
16     args(key)&&
17     withincode(* ProxyFactory.createClass2
           (..))
18 {
19     synchronized(key){
20         proceed(key);
21     }
22 }
```

図 8: 細かい粒度の同期を行うアドバイス

OS Ubuntu Linux 7.10 Server Edition x86_64

JVM 1.6.0_03

20 スレッドでテストプログラムを 1000 回試行して，それぞれのコア数・同期の粒度における平均実行時間，標準偏差は表 1 のようになった．この表から，2 コアの場合は，実行時間は同期の粒度にかかわらず同程度，4 コアの場合は粒度が細かい方が速くなることが分かる．図 9 と図 10 は，2 コア，4 コアの時のそれぞれの実行時間の分布である．これらの図から，4 コアの場合は細かい粒度の方が速くなり，2 コアの時に細かい粒度で同期を行った場合，ばらつきが大きくなることが分かる．また，アスペクトを用いた場合と，アスペクトを用いず synchronized ブロックを直接記述して同期を行った場合とでは，有意な差は見られなかった．これより，2 コアの計算機で実行するときに，最大実行時間が遅くなると困る場合は，アスペクトによるオーバーヘッドのコストを払ってでも，粗い粒度の同期に切り替えた方が良いことが分かる．

6 関連研究

メソッド内の一部の領域をポイントカットする技術はいくつか存在する．LoopsAJ[6] は，for や while などのループをポイントカットするための AspectJ

	細かい粒度		粗い粒度	
コア数	平均	標準偏差	平均	標準偏差
2	13.0s	0.92	13.0s	0.23
4	8.6s	0.34	11.9s	0.18

表 1: 20 スレッド時の平均実行時間・標準偏差

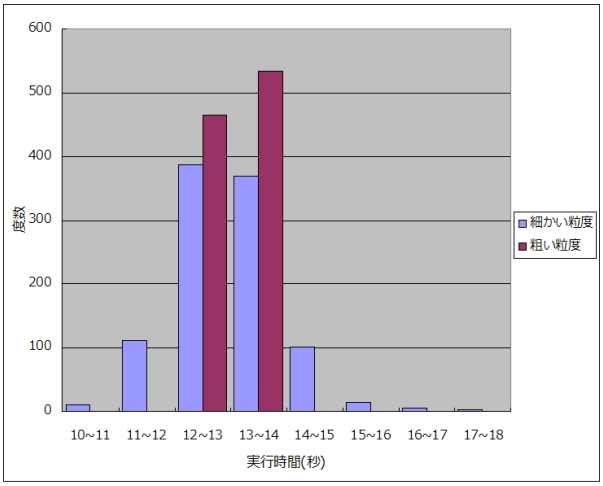

図 9: 2 コア時の分布

の拡張である．これは，ループをポイントカットすることは出来るがそれ以外の領域をポイントカットすることは出来ない．また，同じメソッド中に存在する複数のループに別々のアドバイスを適用することは出来ない．

GluonJ/R[10] は領域をポイントカットし，アドバイスとして例外処理を記述するための研究である．これはアスペクト指向言語の GluonJ[2] の拡張として実装されている．GluonJ/R での領域のポイントカット方法は，我々の提案する言語拡張と同様に，開始・終了位置の 2 点を指定するものである．しかし，include・exclude のような条件の指定はできない．また，制御構造を考慮せず，開始位置から終了位置までを素朴にポイントカットするため，指定の仕方によってはプログラムの構造を壊してしまう．

Xi らによる synchronized ブロックをポイントカットする研究 [8] が存在する．しかし，我々の研究は同期処理以外にも適用可能である．

7 まとめと今後の課題

本稿では，プログラム中のコードの領域をポイントカットすることを可能にする AspectJ 言語の拡張

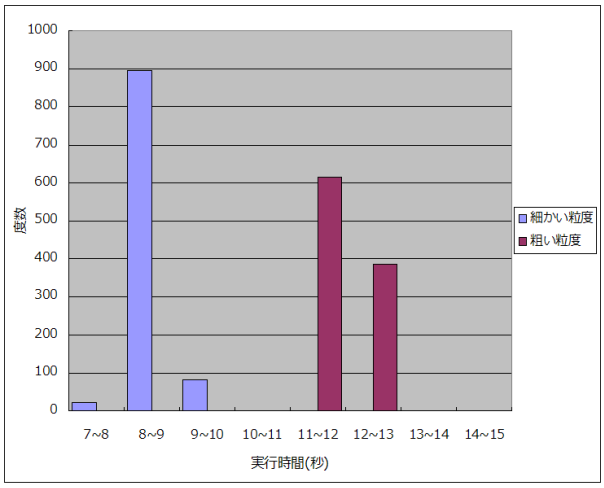

図 10: 4 コア時の分布

を提案した．本言語拡張により，従来のアスペクト指向言語では上手く記述できなかった同期処理などを，アスペクトを用いて分離することが出来ることを示した．また，この言語拡張を abc を拡張することで実装した．

今後の課題としては，領域の指定方法の改善が挙げられる．例えば，include や exclude での条件の指定では，join point が存在するかしないかだけを調べていたが，必要に応じて join point の存在する順序も指定できるようにすることが考えられる．また，ローカル変数を取得する場合に，変数の型で指定していたが，同じ型のローカル変数が複数存在する場合に上手く指定できない場合が有るなどの問題がある．そのため，アドバイスへの受け渡すコンテキストの指定方法をより適切なものにする必要がある．

参考文献

[1] . abc: The aspectbench compiler for aspectj. `http://abc.comlab.ox.ac.uk/introduction`.

[2] . Gluonj. `http://www.csg.is.titech.ac.jp/projects/gluonj/`.

[3] . [#jassist-28] javassist enhancement failed on deserializing hibernate proxies - jboss.org jira. `http://jira.jboss.org/jira/browse/JASSIST-28`.

[4] . Soot: a java optimization framework. `http://www.sable.mcgill.ca/soot/`.

[5] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented*

software development, pp. 87–98, New York, NY, USA, 2005. ACM.

[6] Bruno Harbulot and John R. Gurd. A join point for loops in aspectj. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pp. 63–74, New York, NY, USA, 2006. ACM.

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP '01*, Vol. 2072 of *LNCS*, pp. 327–355, 2001.

[8] Chenchen Xi, Bruno Harbulot, and John R. Gurd. A synchronized block join point for aspectj. In *FOAL '08: Proceedings of the 7th workshop on Foundations of aspect-oriented languages*, p. 39, New York, NY, USA, 2008. ACM.

[9] 千葉滋. Javassist home page. `http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html`.

[10] 熊原奈津子, 光来健一, 千葉滋. 例外処理のためのアスペクト指向言語. 情報処理学会論文誌. プログラミング, Vol. 48, No. 10, pp. 189–198, 20070615.