平成19年度 学士論文

同期処理のモジュール化を可能にするアスペクト指向言語

東京工業大学 理学部 情報科学科
学籍番号 04-00290
赤井 駿平

指導教員
千葉 滋 准教授

平成20 年2 月27 日

概要

本論文では，プログラム中の範囲をポイントカットできる言語機構を提案する．プログラム中の同期処理や例外処理は適用したい範囲をブロックで指定するが，既存プログラムをしばしば横断してしまう．このような処理は分離して記述することが望ましい．特に同期処理は，指定したブロックの粒度や実行する計算機の特性によって，プログラムの動作速度が変化する．プログラムの動作速度を向上させるために同期処理を分離して記述し，計算機の特性に応じてその処理を変更できると便利である．

しかし，既存の技術で同期処理を分離して記述することは困難である．オブジェクト指向言語で同期処理を分離する場合，適用したい範囲をあらかじめメソッドとして宣言しておき，そのサブクラスのメソッドで同期処理を記述する．この手法では同期処理を行う範囲を変更する際に，対象ソースコードを書き換えなければならない．また既存のアスペクト指向言語では，プログラム中の実行点をジョインポイントとして指定できるが，ブロックのような実行点の集まりを指定することはできない．そのため，同期処理や例外処理をシンプルなアスペクトとして分離することができない．

この問題を解決するために，本論文ではプログラム中の範囲をポイントカットできる言語機構であるブロックポイントカットを提案し，それを提供するアスペクト指向言語を開発する．ブロックポイントカットを利用することで，開発者はプログラム中の範囲を指定し，その振る舞いをアドバイスで変更できる．指定した範囲の挙動を変更するために，アスペクト以外のソースコードを書き換える必要はない．

また，ソースコード上のブロックと一致しない範囲に対し，同期処理をアスペクトで挿入したい場合がある．本言語では，同一メソッド内の２つの実行点を選択することでその間の範囲を，ジョインポイントとして指定できる．ただしこの指定方法では，複数の範囲をジョインポイントとして指定してしまう場合がある．本言語では，指定したい範囲に含まれるべき実行点を利用者に選択させることで，指定される範囲の絞り込みを行うことができる．さらに，ソースコード上のブロックや制御構造を認識し，それらと重ならないように指定される範囲を決定するため，本言語のアスペクトを織り込んでも元のプログラムの構造を破壊することはない．

本言語は，汎用的なアスペクト指向言語として知られる AspectJ を拡張する．また abc を改造して処理系を実装した．そして，この言語機構を利用して処理の分離を行った場合に発生する実行時間のオーバーヘッドを計測し，それが実用上問題が無いレベルであることを確認した．また，このシステムを利用し同期処理の分離を行い，計算機の特性の違いに応じて同期処理の粒度を変えることで，性能の改善が見込めることを確認した．

謝辞

本研究の方針や進め方についてご指導頂きました千葉滋に心より感謝いたします．

西澤無我氏には，システムの設計や実験の方法，その他研究全般にわたり助言をいただきました．光来健一には，実験用計算機準備にご協力いただきました．研究生活においてご協力下さいました，堀江倫大氏，そのほか千葉研究室の皆様方に感謝いたします．

目 次

図 目 次

表 目 次

第1章　はじめに

同期処理や例外処理などは，メソッド中において，それを適用したいコードの範囲を囲むブロックといった形で記述される．このような処理をオブジェクト指向プログラミングにおいてモジュールに分割しようとした場合，対象となるブロックを予めメソッドとして分離しておき，継承を利用して元のメソッドの振る舞いを変更するなどしなければならない．

アスペクト指向プログラミングは，ロギングのような横断的関心事と呼ばれるオブジェクト指向ではモジュール化しづらい処理を，アスペクトという単位でモジュール化する手法である．アスペクト指向プログラミングでは，メソッド呼び出しやフィールドへのアクセスといった単一の実行点を選択し，その振る舞いを変更することが出来る．しかし，コードブロックのようなプログラム中の範囲の振る舞いを変更することはできない．そのため，同期処理や例外処理をアスペクトを用いてモジュール化しようとした場合には，オブジェクト指向のように，対象となるブロックをメソッドに分離しなければならない．

ブロックの振る舞いの変更を行う場合，対象となるソースコードを書き換える必要がないことが要求される．また，ソースコード上に存在するブロックと一致しないようなコードの範囲であっても，振る舞いを変更するブロックとして選択できる必要がある．さらに，ブロックが選択されることによって，既存のプログラムの構造を破壊して意図しない挙動を示すようなことが有ってはならない．

本研究では，コードブロックの振る舞いを変更し，同期処理や例外処理をモジュール化することができるアスペクト指向言語を提案する．本システムでは，コードブロックをポイントカットを行うための機構を提供する．これにより，上記の要求を満たした上で，コードブロックの振る舞いを変更でき，従来のオブジェクト指向言語やアスペクト指向言語では難しかった，同期処理や例外処理のモジュール化を行うことが出来る．

以下，2 章ではコードブロックの振る舞いを変更することの必要性と，既存技術の問題点について述べる．3 章では本研究で提案する言語の仕様について，4 章ではその実装方法について述べる．5 章では，本システムを用いた場合の性能を計測した実験と，既存のプログラムにおける同期処理に対して本システムを適用した例を示す．6 章で本稿のまとめを行う．

第2章　問題と関連研究

2.1　問題

同期処理や例外処理，ループの並列化を行う場合，通常はブロックや無名クラスなどを用いて，対象となる範囲を囲んで記述される．このような記述を用いた場合，本質的でないそれらの処理をモジュールに分割することが難しいため，問題となる．ここでは，同期処理，例外処理，ループの並列化がモジュール化できないことによる問題を述べる．

2.1.1　同期処理

同期処理は，マルチスレッドプログラミングにおいてプログラムの特定の範囲を実行するスレッドの数を制限すことにより，不整合が生じることを防ぐ処理である．Java においては，synchronized ブロック (図 2.1) で同期を行いたい範囲を囲むことで行う．

粒度の違いによる性能差

同期処理を行うにはオーバーヘッドが生じる．そのため，同期が必要な箇所を細かく分けて同期処理 (粒度の細かい同期) を行う場合，プログラムの並列性が上昇する反面，同期処理を行う回数が増えるため，オーバーヘッドの影響が大きくなる．一方で，同期が必要な箇所をまとめて大きな範囲で同期処理 (粒度の粗い同期) を行う場合は，同期処理を行う回数が減るためオーバーヘッドは小さくなるが，プログラムの並列性が悪くなる．

同期処理には，その粒度によって性能のトレードオフが存在するため，計算機の CPU の数などの環境によって最適な粒度が異なる．例えば，CPU の数が大きい場合は，多少オーバーヘッドが生じても，粒度の細かい同期を行って，複数のスレッドを同時に実行させた方が効率がいい．しかし，CPU の数が少ない場合は，並列性を上げても同時に実行できるスレッドが少ないため，オーバーヘッドの小さい粒度の粗い同期を行った方がいい．

このように，同期処理の粒度は常に細かくしておけばいいというものではない．そのため，同期処理は本質的な処理を行うプログラムからモ

```
1 ...
2 synchronized(obj){
3         //同期が必要な処理
4 }
5 ...
```

図 2.1: synchronized ブロック

ジュールとして分離することで，環境に応じた最適な同期の粒度を選択して，性能向上を行うことが簡単に出来るようになる．

また，ある環境での最適な同期の粒度を選択するための調査を行うためにも，粒度の変更を簡単に行うことが出来る必要がある．

同期の実装方法の変更

Java では同期処理を行うために，通常 synchronized メソッドや，synchronized ブロックを利用する．しかし，Java5 より導入された，java.util.concurrent.locks.ReentrantLock クラスを用いて同期処理を行うことも出来る (図 2.2)．

Java5 の実装では，ReentrantLock の方が，synchronized ブロックを利用した同期処理よりも性能が良い [11]．synchronized ブロックを利用した場合，取得したロックの解放を忘れることはないが，ReentrantLock を利用した場合，finally ブロックの中で明示的にロックの解放を行わないと，解放し忘れる場合がある．そのため，通常は synchronized ブロックを利用し，パフォーマンスが必要になる場合に ReentrantLock を利用するようにすべきである．

また，ReentrantLock よりも更に性能の良い同期の実装が登場した場合には，その実装に切り替えたほうが望ましい．

そのような時に，同期処理をモジュール化することで，同期処理の実装の変更が容易になる．

2.1.2　例外処理

例外処理では，指定したブロックの実行中に異常な動作 (例外) が起きた場合にその時点で実行を中止し，例外ハンドラで指定された例外に対処するための処理を行う．Java では try,catch,finally 文 (図 2.3) を利用して，例外を捕捉するブロック，例外ハンドラ，例外が起きても起きなくても必ず実行する処理を指定する．

```
1  class A{
2      final ReentrantLock lock = new ReentrantLock();
3
4      void a(){
5      ...
6      lock.lock();
7      try{
8          //同期が必要な処理
9      }finally{
10         lock.unlock();
11     }
12     ...
13   }
14 }
```

図 2.2: ReentrantLock を用いた同期処理

```
1  ...
2  try{
3          //例外が発生し得る処理
4  }catch(SomeException e){
5          //例外 SomeException に対する処理
6  }finally{
7          //必ず実行する処理
8  }
9  ...
```

図 2.3: try,catch,finally

Java では throws 節を用いて，そのメソッドから発生する可能性のある例外が明示されている．しかし，RuntimeException クラスのサブクラスである例外 (NullPointerException，ClassCastException 等) は，throws 節で明示されていなくても発生する可能性がある．そのような起こる可能性があるかはっきりとしない例外全てに対し，予め例外ハンドラを記述して対処しておくことは現実的でない．そのため，そのような例外に対処しなければならないと分かってから例外処理を追加する必要がでてくる．

しかし，後から例外処理を追加しようとした場合，既存のプログラムを直接編集して行うと，正しく動いていた既存のプログラムを破壊してしまうかもしれない．そのため，既存のソースコードに手を加えずに別のモジュールとして例外処理を追加することができるとよい．

また，例外処理は通常は本質的な処理ではない．別のモジュールとして例外処理を記述できることで，本質的な処理の記述に集中することができるようになる．

```
1  ...
2  for(int i=1;i<=1000;i++){
3    double sqrt=Math.sqrt(i);
4    System.out.println("sqrt("+i+")="+sqrt);
5  }
6  ...
```

図 2.4: 並列化をしていないループ

```
1  ...
2  for(int i=1;i<=1000;i++){
3      final int j=i;
4      new Thread(new Runnable(){
5          public void run(){
6              double sqrt=Math.sqrt(j);
7              System.out.println("sqrt("+j+")="+sqrt);
8          }
9      }).start();
10 }
11 ...
```

図 2.5: 並列化を行ったループ

2.1.3　ループの並列化

ループ処理において，ループの各ステップが他のステップに依存していない場合，それぞれのステップは並列に実行できる可能性がある．ループを並列化した場合，プログラムの実行の効率化が見込める．

Java では，Runnable インターフェースを実装したクラスと，Thread クラスを利用して，並列化を行える．例えば，図 2.4 のように，1 から 1000 までの平方根を表示するプログラムが有ったとする．これを，図 2.5 と書き換えることで，ループの各ステップを並列に実行することができる．

ループを並列化したときにメリットが有るのは複数の CPU が利用可能な場合であって，利用できる CPU が 1 つのみの場合並列化してもオーバーヘッドが掛かってしまう．また，ループは並列に動かすかどうかは結果には関係なく，効率にのみ影響がある．そのため，そのような本質的でない処理はモジュールに分割して，並列化を行うかどうかを変更できることが望ましい．

よって，ループを並列化する場合も，モジュール化されることによってプログラミングの負担が軽減される．

```
1   ...
2   for(int i=1;i<=10;i++){
3       someMethod1(); //同期の必要のないメソッド
4
5       //同期を開始すべき点
6       toBeSynchronized1();
7       toBeSynchronized2();
8       //同期を終了すべき点
9
10      someMethod2(); //同期の必要のないメソッド
11  }
12  ...
```

図 2.6: 同期の例

2.2　求められる能力

プログラムのモジュール化を行う場合，オブジェクト指向プログラミングでは，既存のメソッドをサブクラスでオーバーライドすることでメソッドの振る舞いを変更する．その場合，振る舞いを変更できるのはメソッド呼び出しの単位でしか振舞いを変更できない．そのため，2.1 のような問題は解決できない．

しかしながら，複数の操作を含むプログラムの範囲であるコードブロック (code block) の振舞いを変更することにより，プログラムの本質と関係ない処理を分離し，モジュール性を向上させることが出来る．

以下では，コードブロックの振る舞いを変更する機構に求められる能力について述べる．

2.2.1　プログラム中の任意の範囲の振る舞いを変更可能

振る舞いを変更するコードブロックを指定する場合，ソースコードに存在するブロック (Java では {} で囲まれた範囲) の途中から途中まで，といった指定が行うことが出来なければならない．

例えば，図 2.4 のような，ループを並列化する場合は，”for 文に与えらたブロックの振る舞いを変える”というような指定が出来れば十分であると考えられる．しかし，同期処理や例外処理を行いたい場合は，図 2.6 のように対象となるブロックがソースコード上のブロックと一致しない場合がある．したがって，そのようなブロックを指定して，その振る舞いを変更できなければならない．

図 2.7 では，start() メソッドの呼び出しから，end() メソッドの呼び出しまでの範囲で同期処理を行いたいとする．この場合，同期処理を行うパ

```
1  ...
2  start(); //同期の開始点 (1)
3  doSomething1();
4  end(); //同期の終了点 (1)
5
6  ...
7
8  start(); //同期の開始点 (2)
9  doSomething2();
10 end(); //同期の終了点 (2)
11 ...
```

図 2.7: 複数のブロック選択パターン

ターンとしては 2 パターン考えることが出来る．1 つは,「同期の開始点 (1)」から「同期の終了点 (1)」までと,「同期の開始点 (2)」から「同期の終了点 (2)」までの 2 箇所で同期処理を行うパターン．もう 1 つは,「同期の開始点 (1)」から「同期の終了点 (2)」までの 1 箇所で同期処理を行うパターンである．同期処理の粒度を切り替えるためには，この両方のパターンを指定できる必要がある．

また，2 箇所で同期を行うパターンの場合，それぞれの箇所の start() と end() の間で行われる処理が異なる．この場合，その 2 箇所の振る舞いはそれぞれ異なるように変更できなければならない．それには，この 2 箇所のブロックを識別できる必要がある．

2.2.2　対象のソースコードの変更が不要

コードブロックの振る舞いを変更しようとする場合，対象の箇所のソースコードを書き換えずにそのブロックの振る舞いを変更できることが望ましい．振る舞いを変更するために，何らかの変更を既存のコードに施さなければならない場合，誤ってプログラムをロジックを破壊してしまい，バグを発生させる危険性がある．

また，同期処理の粒度を変更する場合，同期を施さなければならない箇所が異なるメソッドやクラスに散らばっている可能性がある．そのような場合，散らばった箇所への変更を既存のコードから分離し，1 つのモジュールで管理できると，プログラムの編集を行う手間が軽減される．

2.2.3　プログラムの構造を破壊しない

ブロックに対し，同期処理の機能を追加するために，Java のバイトコードのレベルでプログラム挿入する場合を考える．その場合，何も制限を掛

```
1  if(isSomething()){
2      ...
3      start();
4      ...
5  }
6  end();
```

図 2.8: 既存ブロックにまたがるブロック指定

```
1  if(isSomething()){
2      ...
3      synchronized(obj){//synchronize start
4      start();
5      ...
6  }
7  end();
8      }// synchronize end
```

図 2.9: 既存ブロックを無視したプログラムの変更

けなければ，ソースコード上のブロック構造を無視したプログラムの変更を行うことが可能である．

例えば，図 2.8 の start() から end() までを同期すると指定すると，図 2.9 のようなプログラムの変更を行うことが出来る．この時バイトコード上では，図 2.9 の「synchronize start」の行に monitorenter 命令が,「synchronize end」の行に monitorexit 命令が挿入される．図 2.9 では，monitorenter 命令 が if 文の中に存在するため，実行されない可能性がある．しかし，monitorexit 命令は必ず実行される．カレントスレッドがモニタの所有者でないにもかかわらず，monitorexit 命令を実行した場合，例外 (IllegalMonitorStateException) が発生する [13, 303 ページ].

このように，ソースコード上の制御構造やブロック構造を考慮せずにコードブロックの振る舞いを変更した場合，意図しないプログラムの破壊が起きてしまう可能性がある．つまり，ソースコード上においてブロック構造を挿入できないような箇所をコードブロックとして指定して，その振る舞いを変更できてはいけない．そのため，コードブロックの振る舞いを変更するモジュールは，ソースコード上の制御構造やブロック構造の情報を基に，プログラムを不用意に破壊しないように注意して，振る舞いを変更する箇所を定めなければならない．

```
1 static Map dataMap;
2 public void addData(String key,int i){
3       SomeObject obj=(SomeObject)dataMap.get(key);
4       synchronized(obj){
5           obj.number+=i;
6       }
7 }
```

図 2.10: ローカル変数に対する同期処理

2.2.4　ローカル変数を取得可能

コードブロックの振る舞いを変更する場合，その振る舞いを変更するために書かれるモジュール側のコード内から，対象のコードブロックで有効なローカル変数の値を取得できる必要がある．

同期処理

同期処理を行う場合では，図 2.10 のように，ローカル変数に対して同期処理を行うことがある．この例では，同期処理をモジュール側のコードで行いたい場合，モジュール側でローカル同期の対象となるローカル変数 *obj* の値を取得できなければならない．

例外処理

モジュールによって例外処理を行う場合にも，ローカル変数の値をモジュール側のコード上で取得できる必要がある．

例えば，図 2.11 のようなファイルから文字列を読み込むメソッドが有る．これを，図 2.11 のようなファイルの読み込み中に例外が発生した場合に「スタックトレースを表示し，ファイルを閉じた上で呼び出し元へ例外を投げる」という例外処理を追加したいとする．この場合，ファイルを閉じるためにはローカル変数 *br* の値を，モジュール側の例外処理を行うコードを取得できなければならない．

このように，モジュールによってコードブロックの振る舞い変更することで同期処理や例外処理を行おうとした場合，その処理の対象となるコードブロック中で有効なローカル変数の値をモジュール上で取得できなければならない．

```
1  public String readString(String filename) throws Exception{
2      String result="";
3      String line;
4      BufferedReader br=new BufferedReader(new FileReader(filename
           ));
5
6      while((line=br.readLine())!=null){
7          result+=line;
8      }
9
10     br.close();
11     return result;
12 }
```

図 2.11: ファイルから文字列を読み込むメソッド

2.3　既存の研究とその問題点

2.3.1　アスペクト指向プログラミング

オブジェクト指向プログラミングでは，プログラミングをする上での関心事をオブジェクトやクラスという単位にまとめることでプログラムのモジュール性を高め，プログラミングの効率を上げてきた．ところが，ロギング，エラー処理など，オブジェクト指向プログラミングでは綺麗にモジュール化しづらい関心事が存在する．

アスペクト指向プログラミングでは，そのような関心事を**横断的関心事**と呼び，横断的関心事を**アスペクト**という単位でモジュール化を行う．また，アスペクトへのモジュール化を行う機能を持ったプログラミング言語を**アスペクト指向言語**と呼ぶ．

2.3.2　AspectJ

AspectJ [2][12] は，現在最も代表的なアスペクト指向言語である．AspectJ では，プログラム上のある実行点の振る舞いを変更や拡張することが可能である．

join point

AspectJ において，振る舞いを変更可能な実行点のことを *join point* と呼ぶ．join point には，メソッド呼び出し，メソッド本体の実行，フィールドアクセス，等の種類が存在する．

```
1  public String readString(String filename) throws Exception{
2      String result="";
3      String line;
4      BufferedReader br=new BufferedReader(new FileReader(filename
           ));
5      try{
6          while((line=br.readLine())!=null){
7              result+=line;
8          }
9      }catch(Exception e){
10         e.printStackTrace();
11         br.close();
12         throw e;
13     }
14
15     br.close();
16     return result;
17 }
```

図 2.12: ファイルから文字列を読み込むメソッド (例外処理を追加)

ポイントカット

プログラム上に多数存在する join point の中から，振る舞いを変更したい join point の集合を選択する方法を **ポイントカット** *(pointcut)* と呼ぶ．

AspectJ で join point を指定するポイントカットは，

```
call(void PrintStream.println(String)) &&
withincode(void Analyzer.analyze())
```

のように，*pointcut* **指定子**と呼ばれる，関数呼び出しのような形をした記述の，論理式によって表される．上の例は，void Analyzer.analyze() メソッドの中における，void PrintStream.println(String) メソッドの呼び出しを指定している．

AspectJ は主なものでは，以下のような pointcut 指定子を持つ．

call パターンにマッチするメソッドの呼び出しを選択する

execution パターンにマッチするメソッドの本体を実行をを選択する

get パターンにマッチするフィールドの参照を選択する

set パターンにマッチするフィールドへの代入を選択する

within パターンにマッチする型の中に存在する join point を選択する

withincode パターンにマッチするメソッドの中に存在する join point を選択する

```
1 before() : call(void PrintStream.println(String)&&
2                 withincode(void Analyzer.analyze()))
3 {
4     System.out.println("before␣println()␣in␣Analyzer.analyze()"
          );
5 }
```

図 2.13: アドバイスの記述例

this join point において，this が指定された型であるものを選択する

target join point において，target が指定された型であるものを選択する

args join point において，引数が指定された型であるものを選択する

名前付きポイントカット　ポイントカットには変数のように，名前を付けることができる．ポイントカットに名前をつけるにはアスペクト中で，

```
pointcut publicCall(): call(public void *.println(String));
```

のように記述する．すると，publicCall() が新たにポイントカットとして使用することが出来る．

アドバイス

pointcut によって選択された join point の集合において，その振る舞いを変更するために実行したいコードを指定することを**アドバイス** *(advice)* という．

アドバイスには，以下の 3 種類が存在する．

before 指定した join point の実行前にコードを実行する

after 指定した join point の終了後にコードを実行する

around 指定した join point を，実行したいコードで置き換える

アドバイスの記述例は，図 2.13 のようになる．

アドバイスには，join point におけるコンテキストを渡すことが出来る．コンテキストを利用するには，図 2.14 のように，args,this,target ポイントカットに対して変数名を記述することで，その名前でアドバイス内から参照できる．

```
1 after(String str,PrintStream stream) :
2     call(void PrintStream.println(String)&&
3     args(str) &&
4     target(stream) &&
5     withincode(void Analyzer.analyze()))
6 {
7     System.out.println("contexts␣are␣"+str+"␣and␣"+stream);
8 }
```

図 2.14: コンテキスト渡しの記述例

```
1 void around(String str) :
2     call(void PrintStream.println(String) &&
3     args(str)
4     withincode(void Analyzer.analyze()))
5 {
6     System.out.println("brefore");
7     proceed(str + "_adviced");
8     System.out.println("after");
9 }
```

図 2.15: around アドバイス

around アドバイスでは，2.15 のように，アドバイスのボディの中で proceed() を呼び出すことで，置き換える前の元の join point を実行することが出来る．この時 proceed() の引数にはアドバイスに渡ってきたコンテキストを与えるが，渡ってきたものと異なるコンテキストを与えて join point を実行することも出来る．

問題点

AspectJ では join point は文字通り点であり，call,set,get など，基本的に特定の 1 つの操作における振る舞いしか変更できない．execution join point は，メソッドのボディ全体を join point として，複数の操作を含む範囲の振る舞いを変更できる．

しかし，メソッドのボディ全体という限定された範囲しか選択できないため，例外処理，ループの並列化といったことを行うためには不十分である．

AspectJ において，java.util.concurrent.locks.ReentrantLock と，before・after アドバイスを用い，図 2.16 のように記述することで，アスペクトを用いて同期処理を行うことが出来るように見える．ところが，start() の呼び出しから end() の呼び出しの直前までの処理において例外が発生する

```
1  class ToSync{
2      void a(){
3
4          start();//同期が必要な最初のメソッド呼び出し
5          ...
6          end();//同期が必要な最後のメソッド呼び出し
7
8      }
9  }
10 aspect Lock{
11     final ReentrantLock ToSync.lock=new ReentrantLock(); //
           ToSync クラスにフィールドを追加
12     before(ToSync t):
13         call(* *.start())&&
14         this(t)
15     {
16             t.lock.lock();
17     }
18     after(ToSync t):
19         call(* *.end())&&
20         this(t)
21     {
22             t.lock.unlock();
23     }
24 }
```

図 2.16: AspectJ による同期のモジュール化

と，ロックが解放されない．このため，AspectJ では同期を行う場合にも不十分である．

2.3.3　LoopsAJ

LoopsAJ[9] は，AspectJ を拡張した言語であり，ループを join point としてポイントカットすることが出来る．これにより，ループを並列化させることが可能である．

図 2.17 のようなアドバイスを記述することで，整数の範囲を繰り返すループをポイントカット出来る．この時，アドバイスの引数 *min* にはループカウンタの整数の最小値が，*max* には最大値が，*step* にはカウンタがいくつずつ増加していくか，が渡される．

図 2.18 のアドバイスでは，配列の各要素に対して繰り返すループもポイントカット出来る．このように記述することで，*min*，*max*，*step* に加え，引数 *array* が利用でき，*array* には，繰り返している配列が渡される．

このポイントカット機構を利用し，図 2.19 のアドバイスを適用すると，ループにおける各繰り返しの偶数番目と奇数番目をそれぞれ別のスレッド

```
1 void around(int min,int max,int step) :
2       loop() && args(min, max, step)
3 {
4   //some adivce
5 }
```

図 2.17: 整数の範囲を繰り返すループに対するアドバイス

```
1 void around(int min,int max,int step,Object[] array) :
2       loop() && args(min, max, step, array)
3 {
4   //some adivce
5 }
```

図 2.18: 整数の範囲を繰り返すループに対するアドバイス

で実行させることができる．

LoopsAJ でポイントカット可能なのはループ構造のみであるため，同期処理や例外処理を行うには対象のソースコードを書き換えて 1 度しか実行しないループを形成する，等の方法をとるしかない．また，Java にはループに対して名前などのメタデータを与えることかできないため，1 つのメソッド内に存在する複数のループを区別することができない．

LoopsAJ では，ループ構造の中でも，図 2.20 のような Iterator を利用したループをポイントカットすることはできない．また，ラベル付の break や continue を利用した大域的なジャンプを持つループに対して around アドバイスを適用することも出来ない．このような制限のため，ループを並列化するにあたってもポイントカットできないループが存在する．

2.3.4　Explicit joinpoints

Explicit joinpoints [10] は，AspectJ を拡張した言語である．この言語では，join point を定義し，プログラム中に join point 名と，ブロックを記述することで，そのブロックを join point としてポイントカットすることが可能である．

図 2.21 は，Explicit joinpoints を用いて同期処理を記述したアスペクトの例である．このように，*scoped joinpoint* キーワードを用いてブロックを対象とした join point を定義し，

```
1  void around(int min , int max , int step ):
2    loop() && args (min , max , step , ..)
3  {
4      for (int i = 0 ; i< 2 ; i ++) {
5          final int t_min = min +i ;
6          final int t_max = max ;
7          final int t_step = 2 * step ;
8          new Thread(new Runnable () {
9              public void run () {
10                 proceed(t_min , t_max , t_step ) ;
11             }
12         }).start() ;
13     }
14 }
```

図 2.19: ループを並列化するアドバイス

```
1  Iterator it = collection.iterator();
2  while(it.hasNext()){
3      System.out.println(it.next());
4  }
```

図 2.20: Iterator を利用したループ

```
call(ejpscope(* Sync.sync(..)))
```

という指定子でブロックをポイントカットする．
　この時，

```
Sync.sync(toSync){
}
```

というブロックで囲まれた範囲がポイントカットされる (toSync は同期を行いたいオブジェクト)．例えば，図 2.10 のプログラムは，図 2.22 のように書き換えることで，アスペクトを用いて同期処理を行うことが出来る．

　このように，Explicit joinpoints を用いれば，コードブロックを join point として，ポイントカットでき，join point にパラメータを渡すことで，ローカル変数を取得できる．また，ソースコード上でブロックを挿入可能な場所ならば自由にポイントカットできる．そのため，同期処理，例外処理，ループの並列化などをアスペクトを用いて行うことが出来る．しかし，Explicit joinpoints を利用するためには，必ず対象のソースコードを変更しなければならない，という欠点がある．

```
1  aspect Sync {
2      scoped joinpoint void sync(Object toSync);
3      void around(Object toSync) :
4          call(ejpscope(* Sync.sync(..))) &&
5          this(toSync)
6      {
7          synchronized(toSync){
8              proceed(toSync);
9          }
10     }
11 }
```

図 2.21: Explicit joinpoints による同期処理を行うアスペクト

```
1  static Map dataMap;
2  public void addData(String key,int i){
3      SomeObject obj=(SomeObject)dataMap.get(key);
4      Sync.sync(obj){
5          obj.number+=i;
6      };
7  }
```

図 2.22: Explicit joinpoints による同期処理を行うアスペクト

2.3.5　GluonJ/R

GluonJ/R [15] は，アスペクト指向言語である GluonJ[3] を拡張した言語である．この言語は，ブロックを join point としてポイントカットし，そのブロックに対して例外処理を追加することが出来る．

GluonJ/R では，ブロックをポイントカットするために，block ポイントカットというポイントカット指定子を用いる．ブロックポイントカットは，

```
Pcd.block(Pcd.call("A#start(..)"), Pcd.call("B#end(..)"))
```

のように，2 つのポイントカットのペアを指定して記述する．最初に指定されるポイントカットが始点となり，後に指定されるものが終点となる．この場合はメソッド内での，A.start() メソッドの呼び出しから，B.end() メソッドの呼び出しまでが join point となり，ポイントカットされる．

ポイントカットされたブロックに例外処理を織り込むには，recover アドバイスを用いる．recover アドバイスは図 2.23 のように，@Recover というアノテーションで記述される．*etype* で捕捉する例外クラスを，*advice* で例外が発生したときに行いたい処理を指定し，その下に記述されているポイントカットに例外処理を織り込む．

```
1 @Glue
2 class PrintStackTrace {
3     @Recover(
4         etype = "java.lang.Exception",
5         advice = "{$e.printStackTrace();}")
6     Pointcut p = Pcd.block(
7         Pcd.call("A#start(..)"),
8         Pcd.call("B#end(..)" ));
9 }
```

図 2.23: GluonJ/R による例外処理

図 2.24 のように，1 つのメソッド内に，start() と end() が複数存在し，

```
Pcd.block(Pcd.call("A#start(..)"),Pcd.call("A#end(..)" ))
```

という範囲を指定したとする．このように，1 つのメソッド内に始点/終点にマッチする join point が複数存在する場合，GluonJ/R では，最も近いペアが join point として選択される．この例では，6 行目から 8 行目が選択される．この時に外側の start() と end() をポイントカットする必要があっても，この選択ルールのためにそのような指定はできない．

選択したいブロックの前後に，

```
@Line(begin)
...
@Line(end)
```

といった @Line アノテーションを記述し，

```
Pcd.block(Pcd.line("begin"), Pcd.line("end"))
```

というポイントカットを行うことで，明示的に選択するブロックを指定する，行アノテーションという機能が GluonJ/R には存在するが，この機能を利用するためには対象となるソースコードの編集が必要となる．

GluonJ/R では，対象としたソースコードに存在するループなどの制御構造を無視したブロックの指定行うことが出来る．そのため，意図しないプログラムの破壊を招いてしまう恐れがある．

このように，GluonJ/R でのブロックの指定には問題があり，ブロックに対して織り込める処理も例外処理のみであるため，ブロックの振る舞いを変更するということには不十分である．

```
1  class A{
2      void manyStartAndEnd(){
3          ...
4          start();
5          ...
6          start();
7          ...
8          end();
9          ...
10         end();
11         ...
12     }
13 }
```

図 2.24: 複数の始点/終点が存在する場合

第3章　ブロックを選択するポイントカット

2章で説明した，ブロックの振る舞いを変更する能力を備えたプログラミング言語として，AspectJ を拡張した言語設計した．この章では，その言語が持つ，ブロックを選択可能なポイントカット機構である，ブロックポイントカットについて述べる．

3.1　begin/end ポイントカット

ブロックポイントカットでは，ブロックを始点と終点となる文を指定して選択する．その指定のために，begin ポイントカット，及び end ポイントカットを提供する．

3.1.1　単純な指定

begin/end ポイントカットでは，

```
begin(call(* A.startBlock())) &&
end(set(* A.endField))
```

のように，始点/終点となるポイントカットを与え，begin と end の論理積によってブロックを指定する．

このように begin と end による指定が行われると，begin によって指定された join point を含む文 (statement) の直前から，end によって指定された join point を含む文の直後までがポイントカットされる．

例えば，図 3.1 では，includeStart() メソッドの呼び出しの直前から，endBlock() メソッドの呼び出しの直後までがポイントカットされる．

begin と end の指定では，

```
begin(call(* A.startBlock())) &&
end(call(* A.endBlock1())) &&
end(call(* A.endBlock2()))
```

```
1  class A{
2      void test(){
3          ...
4          includeStart(startBlock());
5          ...
6          endBlock();
7          ...
8      }
9  }
10
11 aspect B{
12
13     void around() :
14         begin(* *.startBlock()) && end(* *.endBlock())
15     {
16         proceed();
17     }
18 }
```

図 3.1: begin/end ポイントカット

のように，1 つの begin(end) に対し，複数の end(begin) の論理積を指定することは出来ない．ただし，

```
begin(call(* A.startBlock())) &&
(
   end(set(* A.endBlock1())) ||
   end(set(* A.endBlock2()))
)
```

のような，複数の end(begin) の論理和を指定することは可能である．
また，

```
begin( call(* A.startBlock()) && within(A) &&args(x) )
```

のように，begin/end ポイントカットに与えられたポイントカットの中に，within，args 等，join point を表さないポイントカットを記述してもそれは無視される．上の例は，

```
begin( call(* A.startBlock()) )
```

と等価であるとみなされる．

3.1.2　ソースコード上のブロックをまたぐ指定

始点と終点として指定された join point がブロックをまたいで存在する，つまり始点と終点が存在するブロックの深さが異なる場合の，ブロックの選択ルールについて述べる．

```
1 aspect A{
2     void around() :
3         begin(* *.startBlock()) &&
4         end(* *.endBlock())
5     {
6         proceed();
7     }
8 }
```

図 3.2: Aspect A

```
1   void a(){
2       ...
3       //ここから
4       for(int i=0;i<10;i++){
5           startBlock();
6           ...
7       }
8       ...
9       endBlock();
10      //ここまで
11      ...
12  }
```

図 3.3: for ブロックにまたがる指定

ブロックポイントカットでは，始点と終点のブロックの深さが異なる場合，それぞれが所属するブロックを，深さが同じブロックが見つかるまで遡る．その後，同じブロック内に存在する，始点を含むブロックの直前から，終点を含むブロックの直後までを選択する．

例えば，図 3.2 のアスペクト A を，図 3.3 のメソッドに織り込むと，for 文の直前，つまり変数 i の初期化の直前から，endBlock() の呼び出し直後までがポイントカットされる．

また，アスペクト A を図 3.4 のような，1 つの制御構造の異なるブロックに始点・終点が存在するメソッドに織り込んだ場合，その制御構造全体をポイントカットする．この例の場合は，if 文全体が選択される．

3.1.3　対となる始点・終点が複数存在する場合

図 3.5 のメソッドに，図 3.2 のアスペクト A を織り込んだ場合を考える．このように，対となる始点・終点が複数存在する場合，最初に見つかる始点からそれ以降の最初に見つかる終点までがポイントカットされる．また，すでに選択された範囲に，同じアドバイスによって選択され得るブ

```
1  void b(){
2      ...
3      //ここから
4      if(isTrue()){
5          ...
6          try{
7              startBlock();
8          }finally{
9              ...
10         }
11         ...
12     }else{
13       endBlock();
14       ...
15     }
16     //ここまで
17     ...
18 }
```

図 3.4: for ブロックにまたがる指定

ロックが存在しても，そのブロックは選択しない．この例の場合，最初の startBlock() から最初の endBlock() までが選択され，2 番目の startBlock() はどの終点ともペアにならない．

図 3.6 のように，選択されたブロックの外にさらに選択可能な始点・終点のペアが存在する場合，後に存在するペアもブロックとして選択される．

3.1.4　対となる始点・終点が複数存在する場合

begin ポイントカットと，end ポイントカットは，通常ペアで利用される．しかし，begin，end の片方を省略して利用することが可能である．

begin を省略した場合

begin ポイントカットが省略され，end ポイントカットのみが用いられた場合，ブロックの始点は end による終点に最も近いブロックの先頭となる．

例えば，図 3.7 のメソッドに対し，

```
end(call(* *.endBlock1()))
```

というポイントカットを指定した場合，”n+=1;” から，”endBlock();” までがポイントカットされる．

```
1   void c(){
2       ...
3       //ここから
4       startBlock();
5       ...
6       startBlock();
7       ...
8       endBlock();
9       //ここまで
10      ...
11      endBlock();
12      ...
13  }
```

図 3.5: 複数の始点・終点

```
1   void d(){
2       ...
3       //ここから
4       startBlock();
5       ...
6       endBlock();
7       //ここまで
8       ...
9       //ここから
10      startBlock();
11      ...
12      endBlock();
13      //ここまで
14      ...
15  }
```

図 3.6: 複数の始点・終点 2

end を省略した場合

end ポイントカットが省略され，begin ポイントカット単独が用いられると，ブロックの終点は始点以降で最も近いブロックの末尾となる．

図 3.7 のメソッドに対し，

```
begin(call(* *.startBlock1()))
```

というポイントカットを指定した場合，”startBlock1;” から，”n+=2;” までがポイントカットされる．

```
1     void e(){
2         int n=0;
3         for(;;){
4             n+=1;
5             startBlock();
6             ...
7             endBlock();
8             n+=2;
9         }
10    }
```

図 3.7: begin/end の省略

```
1  class ConflictClass{
2      void conflict(){
3          ...
4          startBlock1();
5          ...
6          startBlock2();
7          ...
8          endBlock1();
9          ...
10         endBlock2();
11         ...
12     }
13 }
14 aspect ConflictAspect{
15     after(): begin(call(* *.startBlock1())) && end(call(* *.endBlock1
           ())){}
16     after(): begin(call(* *.startBlock2())) && end(call(* *.endBlock2
           ())){}
17 }
```

図 3.8: 異なるアドバイスによる衝突

3.1.5　異なるアドバイスによるブロックの衝突

図 3.8 のアスペクト ConflictAspect は，startBlock1() から endBlock1()，startBlock2() から endBlock2() までの 2 箇所をポイントカットする．ところが，これら 2 つのブロックは startBlock2() から endBlock1() の範囲で重なってしまっている．この 2 つのブロックは，異なるアドバイスによってポイントカットされているため，片方かのポイントカットを無視する，もしくは片方のブロックの範囲を広げて衝突を避ける，といった事を行うための優先順位を決めることが出来ない．

そのため，このようなブロックの衝突が異なるアドバイス同士で発生した場合，ブロックポイントカットではエラーとなる．

3.2　include/exclude ポイントカット

ブロックは，3.1 節で述べたルールに従って選択される．しかし，図 3.6 において最初の startBlock() から最後の endBlock() までをポイントカットしたい等，基本となるルールと異なる指定を行いたい場合が存在する．そのような指定を可能にするため，本研究では include ポイントカット，exclude ポイントカットと呼ばれる 2 つのポイントカット指定子を提供する．

include/exclude ポイントカットは，

```
begin(call(* *.startBlock())) &&
end(call(* *.endBlock())) &&
include(call(* *.includedMethod())) &&
exclude(call(* *.excludedMethod()))
```

のように，begin/end ポイントカットに追加して指定する．include/exclude ポイントカットは，選択されるブロックの中に include で指定された join point が存在しなければならず，exclude で指定された join point が存在してはいけない，という条件を begin/end ポイントカットに与える．

include/exclude ポイントカットが用いられた場合，まず 3.1 節と同じルールで選択可能な始点・終点のペアによるブロックの候補を探す．そのブロック候補の中に，include で指定された全ての join point が含まれていて，かつ，exclude で指定された全ての join point が含まれていない場合，そのブロックが選択される．条件を満たしていない場合，その候補となっている終点を無視し再びブロックの候補を探す．

図 3.9 のメソッドでは，

```
begin(call(* *.startBlock())) &&
end(call(* *.endBlock()))
```

というポイントカットを行うと，3 行目から 7 行目までがポイントカットされる．ここで，3 行目から 9 行目までをポイントカットするには，

```
begin(call(* *.startBlock())) &&
end(call(* *.endBlock())) &&
include(call(* *.method3()))
```

5 行目から 7 行目までをポイントカットするには，

```
begin(call(* *.startBlock())) &&
end(call(* *.endBlock())) &&
exclude(call(* *.method1()))
```

5 行目から，9 行目をポイントカットするには，

```
1  void f(){
2      ...
3      startBlock();
4        method1();
5      startBlock();
6        method2();
7      endBlock();
8        method3();
9      endBlock();
10     ...
11 }
```

図 3.9: 複数の始点・終点 3

```
begin(call(* *.startBlock())) &&
end(call(* *.endBlock())) &&
exclude(call(* *.method1())) &&
include(call(* *.method3()))
```

と書けばよい.

include/exclude ポイントカットもまた，begin/end と同様に与えられたポイントカットの中に，within，args 等，join point を表さないポイントカットを記述してもそれは無視される.

3.3　コンテキスト渡し

ブロックポイントカットでは，ポイントカットしたブロックにおけるコンテキストをアドバイスに渡すことが出来る.

3.3.1　this ポイントカット

this ポイントカットを用いることで，ブロックの存在するメソッドにおいて *this* で取得できるオブジェクトをアドバイスに渡すことが出来る．(図 3.10)

3.3.2　args ポイントカット

ブロックポイントカットでは，args ポイントカットを利用することで，選択されたブロックの開始地点において有効なローカル変数 (メソッドの引数を含む) をアドバイスに渡すことが出来る.

```
1  public class ThisSampleClass{
2    void startBlock(){}
3    void endBlock(){}
4    void sample(){
5        startBlock();
6        System.out.println("inside␣block");
7        endBlock();
8    }
9  }
10
11 aspect ThisSampleAspect{
12     void around(ThisSampleClass tsc):
13         begin(call(* *.startBlock())) &&
14         end(call(* *.endBlock())) &&
15         this(tsc)
16     {
17         System.out.println(tsc.getClass());
18         //=>"class ThisSampleClass"
19         proceed(tsc);
20     }
21 }
```

図 3.10: this ポイントカットによるコンテキスト渡し

ローカル変数をそのメソッドの外側から利用する場合，名前を指定してローカル変数を選択することは，良い方法ではない．それは，ローカル変数の名前というものはメソッドの外部には公開されていないため，そのメソッドを編集しているプログラマがいつでも名前を変更する可能性があるためである．そのため，リファクタリングの為にローカル変数の名前を変更しただけで，アスペクトが織り込めなくなってしまう．

また，”3 番目に宣言されているローカル変数”，のようにローカル変数の宣言されている順番を指定して選択する方法も得策ではない．ローカル変数の順番も変更されてしまう可能性があるためである．

このような問題が存在するため，ブロックポイントカットでは，ローカル変数の型を利用してアドバイスに渡すローカル変数を指定する．これは，ある程度対象のメソッドが完成に近づいた状態ならば，ローカル変数の型が変更される可能性は低いと思われるためである．

args ポイントカットを利用するには，

```
begin(call(* *.startBlock())) &&
end(call(* *.endBlock())) &&
args(str,n) &&
args(String,int)
```

```
1  void e(String str1){
2      String str2="2";
3      String str3="3";
4      startBlock();
5          ...
6      endBlock();
7  }
```

図 3.11: 複数の同じ型のローカル変数

のように，識別子が与えられた args ポイントカットと，`args(String,int)` のような型名のみが渡されている args ポイントカットを用いて記述する．この時，`args(String,int)` のような，引数として型名のみが渡されている args ポイントカットが存在する場合，その型 (サブクラスは含まない) で宣言されているローカル変数をブロックの開始地点で有効なローカル変数から抽出し，args ポイントカットに与えられた識別子に束縛する．

また，引数として型名のみが渡されている args ポイントカットが存在しない場合，ブロックの開始地点で有効なローカル変数全てが宣言されている順番で，識別子の与えられた args ポイントカットに割り当てられる．

図 3.11 のように，複数の同じ型のローカル変数が存在する場合を考える．このような場合，メソッド内の前方で宣言されているローカル変数から優先的に選ばれる．図の場合，`args(a) && args(String)` と指定した場合，*str1* が *a* に割り当てられ，`args(a,b) && args(String,String)` と指定すると，*str1* が *a* に，*str2* が *b* に割り当てられる．

第4章　実装

本研究では，前章において述べたブロックポイントカットを，AspectJ コンパイラの実装の 1 つである *abc(The AspectBench Compiler)* [1] [7] を拡張することで実装した．abc は AspectJ に新たな機能の追加を行いやすくするための，拡張性に優れた実装である．拡張性のため，abc ではコンパイラのフロントエンドとして Polyglot[5]，バックエンドとして Soot[6] というフレームワークを採用している．

abc でのコンパイル処理の流れは以下のようになる．

1. 構文解析
2. アスペクトの解析
3. 中間表現への変換
4. アドバイスの織り込み
5. 最適化
6. クラスファイルの生成

4.1　ブロックの選択

abc では，アドバイスの織り込みは中間表現である Jimple 上で行う．Jimple は Java バイトコードに対応する 3 番地表現である．

Jimple は Java バイトコードに近い表現であるため，Java のソースコード上に存在するブロックや文の開始/終了位置の情報は残っていない．これに対応するため，本システムでは Jimple に対し，ブロック・文・制御構造のそれぞれの開始/終了位置を表す新たな命令を導入する．

4.1.1　中間表現コンパイラの改造

新たに導入した命令を利用するために，Polyglot によって生成された抽象構文木を Jimple に変換するための中間表現コンパイラ (soot.javaToJimple.JimpleBodyBuilder クラス) を改造する．

文を生成するメソッド (createStmt) の前後で，文の開始/終了位置を示す命令を，ブロックを生成するメソッド (createBlock) の前後でブロックの開始/終了位置を示す命令を命令列に挿入する．また，制御構造 (if,while,do,for,switch,try,synchronized) を生成するメソッドの前後では，制御構造の開始/終了位置を示す命令を命令列に挿入する．

4.1.2　ブロック構造の解析

ブロック・文・制御構造の開始/終了位置を示す命令を基に，各メソッドの Jimple の命令列をトップダウンに解析し，ブロック構造の木構造を作成する．

作成された木構造において，文は命令列を保持する葉ノードとなり，ブロック・制御構造は，子ノードに文・ブロック・制御構造持つ内部ノードとなる．

4.1.3　条件に合うブロックの選択

4.1.2 で作成された木構造を用いて，ポイントカットするブロックを選択する．

ブロックの選択の前処理として，begin・end・include・exclude で指定されたポイントカットの内，各ブロック・文・制御構造の内部においてマッチするものを列挙する．

文においては，文が持つ命令列に対してポイントカットがマッチするかどうかを調べ，マッチしたポイントカットを記録する．ブロック・制御構造では，まず子のブロック・文・制御構造でマッチするポイントカットを再帰的に求める．その後，全ての子にマッチしたポイントカットの和を，そのブロック・制御構造にマッチしたポイントカットとする．

ここでは，各 begin,end,include,exclude の組において，ブロックとして選択する命令列の範囲を決定するアルゴリズムを述べる．

ブロック (制御構造) が，begin，end，include で指定されたポイントカット全てにマッチするか調べる．マッチしなければこのブロック内で選択できるブロックは存在しない．

先頭の子ノードを c とする．c にこのアルゴリズムを再帰的に適用し選択可能なブロックが存在するか調べ，そのノード c 内で選択可能なブロックが存在するか調べる．選択可能なブロックが存在せず，かつ，c が begin のポイントカットにマッチするならば，c を始点とする．c 以降のノード

で，c からそのノードの間のノードに，exclude にマッチするノードが無く，かつ，include で指定されたポイントカットが全て存在し，end にマッチする最初のノードを終点し，終点の次のノードを c としてこの処理を繰り返す．終点となるノードが見つからなければ，c の次のノードを新たな c とし繰り返す．

最後の子ノードまで上の処理を繰り返し，子ノードで見つかった始点・終点のペアと，このブロック (制御構造) で見つかった始点・終点のペアとを合わせたものが，このブロック内で選択可能なブロックの範囲のリストとなる．

4.2　選択範囲の命令列の再構築

4.1.3 の処理を行うことで，ポイントカットすべき命令列の始点と終点となる命令の位置が分かる．そして，始点となる命令の直前と終点となる命令の直後に *nop* 命令を挿入し，その位置を abc に指示することで，その範囲をポイントカットし，abc がアドバイスの織り込みを行う．

始点と終点を指示するだけでは上手くアドバイスの織り込みが行えない場合が存在する．それに対処するため，選択された範囲の命令列の書き換えを行う．

4.2.1　break,continue 文

after アドバイスと around アドバイスを織り込む場合，break 文や continue 文を用いた，選択された命令列の範囲 (ジョインポイントシャドウ，以下シャドウ) の中から外へのジャンプを行う命令が存在すると正しく動かない (図 4.1)．after アドバイスでは，シャドウの後にアドバイスが挿入されるため，シャドウから抜ける際は，終点の直後に挿入した nop 命令を必ず実行するようになっていなければアドバイスが実行されない．また，abc では，around アドバイスを織り込む場合，シャドウが元々存在するメソッドから別の static メソッドに分離されるため，シャドウの外へのジャンプを行うことが出来ない．

この問題に対処するために，まず，シャドウの中から外へのジャンプ全てに番号を振る．そして，ジャンプ命令の直後に，ジャンプの番号を格納するローカル変数にそのジャンプの番号を代入する命令と，シャドウの末尾の nop にジャンプする goto 命令を挿入する．そして，元のジャンプ命令のジャンプ先を，挿入した代入命令に変更する．

シャドウの直後には，tableswitch 命令を挿入し，ジャンプの番号を見て，そのジャンプ命令の元々のジャンプ先にジャンプするように命令列を

```
1  class A{
2      void test(){
3          ...
4          for(int i=0;i<10;i++){
5              ...
6              startBlock();
7              if(isTrue()){
8                  break;
9              }
10             endBlock();
11             ...
12         }
13         ...
14     }
15 }
16 aspect B{
17     void around() :
18         begin(* *.startBlock()) && end(* *.endBlock())
19     {
20         proceed();
21     }
22 }
```

図 4.1: break を含んだブロックの選択

書き換える．

書き換えの様子を疑似コードで表すと，図 4.2 のようになる．

これにより，break 文や continue 文が存在しても，正しくプログラムが動くようになる．

4.2.2　return 文

return 文がシャドウ内に存在する場合も，break 文や continue 文と同様に正しく動かない．この問題は abc では，return 文をメソッドの末尾に移動し，元の return 文をメソッド末尾の return 文へのジャンプに置き換えることで対処されている．

そのため，return 文の移動を行った後に break 文や continue 文と同様に命令列の書き換えを行うことで，ブロックに対してアドバイスを織り込んだ場合でも正しく動くようになる．

4.3　around アドバイスへの対応

4.2 で述べたように，around アドバイスでは，シャドウが別のメソッドへ分離される．この時，シャドウの前で宣言されているローカル変数は，

```
1  //書き換え前
2      nop; //シャドウの先頭
3      startBlock();
4      ...
5      goto jump_target;
6      ...
7      endBlock();
8      nop; //シャドウの末尾
9      ...
10 jump_target:
```

```
1  //書き換え後
2      int jump_number=−1;
3      nop; //シャドウの先頭
4      startBlock();
5      ...
6      goto assign_jump_number;
7  assign_jump_number:
8      jump_number=0;
9      goto end_nop;
10     ...
11     endBlock();
12 end_nop:
13     nop; //シャドウの末尾
14     switch(jump_number){
15     case 0:
16         goto jump_target;
17     }
18     ...
19 jump_target:
```

図 4.2: break を含む命令列の書き換えの様子

abc により自動的に，分離されたメソッドの引数となる．そのため，シャドウの外で宣言されているローカル変数を読み込むことが出来る．しかし，シャドウの前で宣言されているローカル変数に対しシャドウの内部で代入を行っても，その代入は分離される前の元のメソッドには反映されない．例えば，図 4.3 では，アスペクトの織り込みを行わなければ，”assined”と表示されるが，アスペクトを織り込むと，”init”と表示されてしまう．

シャドウの外のローカル変数への代入を反映させるために，ローカル変数を保持するオブジェクトを，シャドウを分離したメソッドの引数として渡すように命令列を書き換える．

まず，シャドウの前までに宣言されているローカル変数の内，シャドウの内部で代入されているローカル変数をリストアップする．そして，そのローカル変数を保存するため，それぞれのローカル変数と同じ型のフィールドを持ったクラスを生成する．

メソッドの先頭に，そのローカル変数を保存するクラスをインスタンス

```
1  class A{
2      void test(){
3          ...
4          String s="init";
5          startBlock();
6          s="assigned";
7          endBlock();
8          System.out.println(i);
9          ...
10     }
11 }
12 aspect B{
13     void around() :
14         begin(* *.startBlock()) && end(* *.endBlock())
15     {
16         proceed();
17     }
18 }
```

図 4.3: シャドウの外のローカル変数への代入

化し適当なローカル変数に代入する命令を挿入する．シャドウの前に，そのインスタンスの各フィールドに対応するローカル変数を代入する命令を挿入する．また，シャドウの末尾にも，インスタンスの各フィールドの値を対応するローカル変数に代入する命令を挿入する．そして，シャドウから抜けた後の場所に，ローカル変数へインスタンスの各フィールドの値を書き戻す命令を挿入する．

この書き換えは，図 4.4 のように行われる．この時，変数 *localStorage* は，abc の機能により，シャドウを分離したメソッドの引数となる．

このように，命令列を書き換えることで，シャドウの外のローカル変数へシャドウ内での代入が反映されるようになる．

図 4.5 のように，around アドバイスを織り込んだシャドウ内で宣言されたローカル変数を，シャドウの外で利用しようとした場合を考える．この場合，宣言が別のメソッドに移動してしまい，一度も代入されていないローカル変数を利用しようとするためにコンパイル時にエラーが起きる．

この問題に対処するため，シャドウの開始前の位置にそのローカル変数の最初の代入を挿入する．この時，変数の型が参照型は null で，boolean 型は flase で，数の型は 0 で初期化しておく．こうすることで，上で述べた方法により，シャドウの内での本当の初期化が外へ反映されるようになり，シャドウ内で宣言されているローカル変数を，シャドウの外で利用することが出来る．

ところが，初期化の行われないローカル変数宣言は，中間表現上には現

```
1  //書き換え前
2      int a;
3      double b;
4      startBlock();
5          a=100;
6          b=10.0;
7      endBlock();
```

```
1  //書き換え後
2      /*生成されるクラス
3       *class LocalStorageClass{
4       * int a;
5       * double b;
6       *}
7       */
8      int a;
9      double b;
10     localStorage=new LocalStorageClass();
11     startBlock();
12         a=100;
13         b=10.0;
14     localStorage.a=a;
15     localStorage.b=b;
16     endBlock();
17     a=localStorage.a;
18     b=localStorage.b;
```

図 4.4: シャドウの外のローカル変数への代入を含む命令列の書き換えの様子

れないため，移動することが出来ない．この場合は，Jimple に初期化のないローカル変数宣言を表現する新たな命令を追加する．そして，それを元にシャドウの前にローカル変数の初期化を挿入する．

4.4　独自の Jimple 命令の除去

4.1.1 では，ブロック・文・制御構造のそれぞれの開始/終了位置を表す命令を，4.3 では，初期化のないローカル変数宣言を表す命令を，Jimple に追加した．このような独自の Jimple 命令は，Java バイトコードへのコンパイル時に nop 命令へとコンパイルされる．このような無駄な nop 命令を無くすため，abc の最適化フェイズにおいて，これらの独自の Jimple 命令を取り除く処理を行う．

```
1  class A{
2      void test(){
3          ...
4          startBlock();
5          String s="assigned";
6          endBlock();
7          System.out.println(s);
8          ...
9      }
10 }
11 aspect B{
12     void around() :
13         begin(* *.startBlock()) && end(* *.endBlock())
14     {
15         proceed();
16     }
17 }
```

図 4.5: シャドウの内で宣言されているローカル変数

第5章　実験

5.1　マイクロベンチマーク

ブロック・ポイントカットを織り込んだ場合に，織り込まなかった場合に比べてどのくらいオーバーヘッドが発生するかを計測するため，以下のような実験を行った．
実験環境は以下通り．

CPU Intel Xeon CPU 5160 3.00GHz 4 コア

メモリ 2GB

OS Ubuntu Linux 7.10 Server Edition x86_64

JVM 1.6.0_03

5.1.1　単純なブロックの場合

10 億回のループを行う，図 5.1 のクラスの a() メソッドのに対し以下の 3 通りの処理を行い，実行時間を計測した．

- アスペクトを織り込まない
- startBlock() への call ポイントカットを行うアスペクト (図 5.2) を織り込む
- startBlock() から endBlock() までのブロックをポイントカットするアスペクト (図 5.3) を織り込む

その結果は，表 5.1 のようになった．
この結果から，アスペクトの織り込みを行わない場合に比べ，アドバイスの 10 億回の実行あたり，11 ミリ秒ほどのオーバーヘッドが存在することが分かった．また，call ポイントカットの実行時間とほぼ同程度であることから，既存の AspectJ のポイントカットを利用したアドバイスと，遜色ない性能を発揮できることが分かった．

```
1  public class Test{
2      void startBlock(){}
3      void endBlock(){}
4      public void a(){
5          for(int i=0;i<1000000000;i++){
6              startBlock();
7              endBlock();
8          }
9      }
10 }
```

図 5.1: アスペクトが織り込まれるクラス

```
1  aspect Call{
2      void around():
3          call(* *.startBlock())
4      {
5          proceed();
6      }
7  }
```

図 5.2: 1 つのメソッド呼び出しをポイントカットするアスペクト

5.1.2　ブロックの外のローカル変数への代入が存在する場合

選択されるブロック内に，ブロックの外で宣言されたローカル変数への代入が存在する図 5.4 のクラスに対し，5.1.1 と同様の，3 パターンの実行速度を計測した．

その結果は，表 5.2 のように得られた．

アスペクトを織り込まなかった場合に比べ，ブロックポイントカットを用いてアスペクトを織り込んだ場合の実行時間は 17 倍以上で，ループ 1 回あたりのオーバーヘッドは 0.083 ナノ秒となっている．これは，ブロッ

```
1  aspect Block{
2      void around():
3          begin(call(* *.startBlock())) &&
4          end(call(* *.endBlock()))
5      {
6          proceed();
7      }
8  }
```

図 5.3: ブロックをポイントカットするアスペクト

	平均実行時間 (ミリ秒)	標準偏差
アスペクト無し	5.06	0.49
call ポイントカット	15.53	0.46
ブロックポイントカット	15.76	0.58

表 5.1: 単純なブロックの場合の実行時間の比較

```
1  public class Test2{
2      void startBlock(){}
3      void endBlock(){}
4      public void a(){
5          int x=0;
6          for(int i=0;i<1000000000;i++){
7              startBlock();
8              x+=1;
9              endBlock();
10         }
11     }
12 }
```

図 5.4: ブロック外への代入がある場合

クの外で宣言されているローカル変数の値の保存・復帰が，アドバイスが実行されるたびに行われているために遅くなっていると考えられる．

このように，オーバーヘッドが大きくなっているのは，ブロックの中やアドバイス内で何の処理も行っていないため，相対的にアドバイス呼び出しの実行時間の影響が大きくなっているものと考えられる．

	実行時間 (ミリ秒)	標準偏差
アスペクト無し	5.17	0.48
call ポイントカット	15.84	0.60
ブロックポイントカット	87.92	0.84

表 5.2: ブロックの外のローカル変数が存在する場合の実行時間の比較

5.1.3　アドバイスで同期を行う場合

アドバイス内で同期処理を行った場合の，アドバイスのオーバーヘッドを計測する．

ブロックの外で宣言されているローカル変数への代入が存在し，1000

```
1  public class TestForLock{
2      void startBlock(){}
3      void endBlock(){}
4      public void a(){
5          int x=0;
6          for(int i=0;i<10000000;i++){
7              startBlock();
8              x+=1;
9              endBlock();
10         }
11     }
12 }
```

図 5.5: 同期が織り込まれるクラス

万回のループを行う，図 5.5 の TestForLock クラスの a() メソッドに対し．以下の 4 種類の処理を行い，実行時間を計測する．

- synchronized ブロックを書き加えて同期を行う (図 5.6)
- ブロックポイントカットと synchronized ブロックを利用して同期するアスペクト (図 5.7) を織り込む
- ReentrantLock を用いるコードを書き加えて同期を行う (図 5.8)
- before，after アドバイスと ReentrantLock 用いて同期を行うアスペクト (図 5.9) を織り込む
- ブロックポイントカットと ReentrantLock を利用して同期するアスペクト (図 5.10) を織り込む

計測した結果は，表 5.3 である．

この結果では，ブロックポイントカットを利用した場合のオーバーヘッドは，synchronized ブロックを利用した場合は，ループ 1 回あたり 6.46 ナノ秒で 7.27%，ReentrantLock を利用した場合は，ループ 1 回あたり 2.86 ナノ秒で 9.57%となっている．

選択されたブロックの内部や，アドバイス中でさらなる処理を行うと，オーバーヘッドの影響は小さくなる．そのため，本システムのオーバーヘッドは，実用上問題の無いレベルであると考えられる．

5.2　同期処理への応用

本研究で作成したブロックポイントカット機構を用いて，Javassist [8] [14] における同期処理のモジュール化を行い，同期処理の粒度の違いによ

```
1  public class ExplicitSync{
2      void startBlock(){}
3      void endBlock(){}
4      public void a(){
5          int x=0;
6          for(int i=0;i<10000000;i++){
7              synchronized(this){
8                  startBlock();
9                  x+=1;
10                 endBlock();
11             }
12         }
13     }
14 }
```

図 5.6: synchronized ブロックを書き加えて同期

```
1  aspect Sync{
2      void around(TestForLock t):
3          begin(call(* *.startBlock())) &&
4          end(call(* *.endBlock())) && this(t)
5      {
6          synchronized(t){
7              proceed(t);
8          }
9      }
10 }
```

図 5.7: ブロックポイントカットと synchronized ブロックを利用した同期を行うアスペクト

る性能の比較を行った．Javassist は，Java バイトコードの読み書きや，クラスの実行時の変更や生成を行うためのライブラリである．

Javassist の javassist.util.proxy.ProxyFactory クラスは，プロキシークラスを動的に生成するクラスである．ProxyFactory は生成したプロキシークラスをキャッシュする機構を備えており，キャッシュを操作する際に，キャッシュの不整合を防ぐために同期処理を行っている．

プロキシークラスの生成は，図 5.11 の ProxyFactory.createClass()，図 5.12 の ProxyFactory.createClass2() で行っており，粗い粒度 (coarse-grained) の同期の場合は createClass() の 4 行目から 9 行目の範囲で，細かい粒度 (fine-grained) の同期は createClass2() の 3 行目から 23 行目と，25 行目から 33 行目の範囲で，同期を行う．

この同期処理の粒度をブロックポイントカットによって切り替えて，粒度の違いによる性能の差を検証する．fine-grained な同期を行う場合は，

```
1  public class ExplicitLock{
2      final ReentrantLock lock=new ReentrantLock();
3      void startBlock(){}
4      void endBlock(){}
5      public void a(){
6          int x=0;
7          for(int i=0;i<10000000;i++){
8              lock.lock();
9              try{
10                 startBlock();
11                 x+=1;
12                 endBlock();
13             }finally{
14                 lock.unlock();
15             }
16         }
17     }
18 }
```

図 5.8: ReentrantLock を用いるコードをを書き加えて同期

	実行時間 (ミリ秒)	標準偏差
synchronized	888.7	5.10
ブロックポイントカット,synchronized	953.3	4.32
ReentrantLock	298.7	2.61
before,after,ReentrantLock	311.8	1.77
ブロックポイントカット,ReentrantLock	327.3	1.11

表 5.3: 同期を行った場合の実行時間

図 5.13 のアスペクトを，coarse-grained な同期を行う場合は，図 5.14 のアスペクトを織り込む.

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 [4] 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は，サーバー側のプログラムでシリアライズしたオブジェクトを，クライアント側で受け取り，それをデシリアライズする際に，クライアント側でプロキシークラスを生成している．クライアント側がサーバーからシリアライズしたオブジェクトを受け取る処理を様々なスレッド数で実行し，実行時間を測定する.

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，クライアントの CPU のコア数を変えて，テストプログラムを実行する.

```
1  aspect Lock{
2      final ReentrantLock TestForLock.lock=new ReentrantLock();
3      before(TestForLock t):
4          call(* *.startBlock())&&
5          this(t)
6      {
7              t.lock.lock();
8      }
9      after(TestForLock t):
10         call(* *.endBlock())&&
11         this(t)
12     {
13             t.lock.unlock();
14     }
15 }
```

図 5.9: before,after と ReentrantLock を利用した同期を行うアスペクト

CPU Intel Xeon CPU 5160 3.00GHz 4 コア

メモリ 2GB

OS Ubuntu Linux 7.10 Server Edition x86_64

JVM 1.6.0_03

クライアント側の計算機の CPU を 2 コアに減らし，計 10000 回プロキシークラスを生成・取得する処理を，スレッド数，及び同期の粒度 (fine-grained と coarse-grained の 2 種類) を変えて，それぞれ 1000 回試行し，実行時間を計測した．

5,10，20，40，80，100 スレッドで動かした時のそれぞれの平均値・中央値・標準偏差は表 5.4 のように得られ，それぞれの度数分布は表 5.5，5.6，表 5.7，表 5.8，表 5.9，表 5.10 のようになった．

クライアント側の計算機の CPU を 4 コアにし，同様に実験を行うと，5,10，20，40，80，100 スレッドで動かした時のそれぞれの平均値・中央値・標準偏差は表 5.11 のように得られ，それぞれの度数分布は表 5.12，表 5.13，表 5.14，表 5.15，表 5.16，表 5.17 のようになった．

この結果より，2 コアの場合は，fine-grained の同期と coarse-grained の同期とでは，平均実行時間を同じ程度だが，fine-grained の方がバラつきが大きいことが分かる．また，度数分布を見ると，coarse-grained では，実行時間が 14 秒未満である割合がほぼ 100%であるのに対し，fine-grained では，約 90%以下と，実行時間が 14 秒以上かかる割合が大きいことが分かる．しかし，4 コアの場合は，圧倒的に fine-grained 方が性能が良い．

```
1  import java.util.concurrent.locks.ReentrantLock;
2  aspect BlockLock{
3      final ReentrantLock TestForLock.lock=new ReentrantLock();
4      void around(TestForLock t):
5          begin(call(* *.startBlock())) &&
6          end(call(* *.endBlock())) && this(t)
7      {
8          t.lock.lock();
9          try{
10             proceed(t);
11         }finally{
12             t.lock.unlock();
13         }
14     }
15 }
```

図 5.10: ブロックポイントカットと ReentrantLock を利用した同期を行うアスペクト

```
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         }
11         return thisClass;
12     }
```

図 5.11: ProxyFactory.createClass()

これより，ProxyFactory においては，2 コアで 10 スレッド以上での利用が見込まれ，実行時間が安定していて遅延が発生しにくいことが好ましい場合は，coarse-grained な同期を，4 コアの場合は fine-grained な同期を行った方が良いということが分かった.

	fine-grained			coarse-grained		
スレッド数	平均 (秒)	中央値 (秒)	標準偏差	平均 (秒)	中央値 (秒)	標準偏差
5	12.43	12.47	0.51	12.94	12.93	0.25
10	13.19	13.19	0.59	12.90	12.92	0.23
20	13.01	12.99	0.92	12.99	12.92	0.23
40	13.31	13.20	1.02	13.08	13.07	0.26
80	13.36	13.36	1.00	13.10	13.11	0.276
100	12.88	12.91	1.73	13.13	13.11	1.39

表 5.4: 2 コア時の平均実行時間・標準偏差

	fine-grained		coarse-grained	
実行時間 (秒)	度数	累積	度数	累積
9 ∼ 10	6	6	0	0
10 ∼ 11	13	19	0	0
11 ∼ 12	129	148	2	2
12 ∼ 13	768	916	615	617
13 ∼ 14	84	1000	383	1000

表 5.5: 度数分布：2 コア, 5 スレッド

	fine-grained		coarse-grained	
実行時間 (秒)	度数	累積	度数	累積
9 ∼ 10	1	1	0	0
10 ∼ 11	3	4	0	0
11 ∼ 12	14	18	0	0
12 ∼ 13	339	357	661	661
13 ∼ 14	565	922	339	1000
14 ∼ 15	66	988	0	1000
15 ∼ 16	12	1000	0	1000

表 5.6: 度数分布：2 コア, 10 スレッド

	fine-grained		coarse-grained	
実行時間 (秒)	度数	累積	度数	累積
10 ∼ 11	10	10	0	0
11 ∼ 12	111	121	0	0
12 ∼ 13	387	508	466	466
13 ∼ 14	369	877	534	1000
14 ∼ 15	101	978	0	1000
15 ∼ 16	15	993	0	1000
16 ∼ 17	5	998	0	1000
17 ∼ 18	2	1000	0	1000

表 5.7: 度数分布：2 コア, 20 スレッド

	fine-grained		coarse-grained	
実行時間 (秒)	度数	累積	度数	累積
10 ∼ 11	11	11	0	0
11 ∼ 12	51	62	0	0
12 ∼ 13	334	396	364	364
13 ∼ 14	396	792	634	998
14 ∼ 15	156	948	1	999
15 ∼ 16	37	985	1	1000
16 ∼ 17	9	994	0	1000
17 ∼ 18	5	999	0	1000
18 ∼ 19	0	999	0	1000
19 ∼ 20	0	999	0	1000
20 ∼ 21	0	999	0	1000
21 ∼ 22	1	1000	0	1000

表 5.8: 度数分布：2 コア, 40 スレッド

	fine-grained		coarse-grained	
実行時間 (秒)	度数	累積	度数	累積
10 ∼ 11	9	9	0	0
11 ∼ 12	63	72	0	0
12 ∼ 13	246	318	336	336
13 ∼ 14	476	794	663	999
14 ∼ 15	162	956	1	1000
15 ∼ 16	36	992	0	1000
16 ∼ 17	5	997	0	1000
17 ∼ 18	1	998	0	1000
18 ∼ 19	1	999	0	1000
19 ∼ 26	0	999	0	1000
26 ∼ 27	1	1000	0	1000

表 5.9: 度数分布：2 コア, 80 スレッド

	fine-grained		coarse-grained	
実行時間 (秒)	度数	累積	度数	累積
9 ∼ 10	2	2	0	0
10 ∼ 11	67	69	0	0
11 ∼ 12	136	205	0	0
12 ∼ 13	336	541	340	340
13 ∼ 14	346	887	658	998
14 ∼ 15	96	983	1	999
15 ∼ 16	10	993	0	999
16 ∼ 17	4	997	0	999
17 ∼ 18	1	998	0	999
18 ∼ 34	0	998	0	999
34 ∼ 35	1	999	0	999
35 ∼ 51	0	999	0	999
50 ∼ 51	1	1000	0	999
51 ∼ 56	0	1000	0	999
56 ∼ 57	0	1000	1	1000

表 5.10: 度数分布：2 コア, 100 スレッド

```
1   private void createClass2(ClassLoader cl) {
2       CacheKey key = new CacheKey(superClass, interfaces,
            methodFilter, handler);
3       // synchronized (proxyCache) {
4           HashMap cacheForTheLoader = (HashMap)proxyCache.
                get(cl);
5           if (cacheForTheLoader == null) {
6               cacheForTheLoader = new HashMap();
7               proxyCache.put(cl, cacheForTheLoader);
8               cacheForTheLoader.put(key, key);
9           }
10          else {
11              CacheKey found = (CacheKey)cacheForTheLoader.get
                    (key);
12              if (found == null)
13                  cacheForTheLoader.put(key, key);
14              else {
15                  key = found;
16                  Class c = isValidEntry(key); // no need to
                        synchronize
17                  if (c != null) {
18                      thisClass = c;
19                      return;
20                  }
21              }
22          }
23      // }
24
25      // synchronized (key) {
26          Class c = isValidEntry(key);
27          if (c == null) {
28              createClass3(cl);
29              key.proxyClass = new WeakReference(thisClass);
30          }
31          else
32              thisClass = c;
33      // }
```

図 5.12: ProxyFactory.createClass2()

```
1  package javassist.util.proxy;
2  import java.lang.ref.WeakReference;
3  import java.util.*;
4  import javassist.bytecode.*;
5  public aspect FineGrainedSync{
6      void around():
7          begin(call(* WeakHashMap.get(..))) &&
8          end(call(* WeakHashMap.put(..))) &&
9          withincode(* ProxyFactory.createClass2(..))
10     {
11         synchronized(ProxyFactory.class){
12             proceed();
13         }
14     }
15
16     void around(ProxyFactory.CacheKey key):
17         begin(call(* *.isValidEntry(..))) &&
18         end(call(WeakReference.new(..))) &&
19         exclude(call(* HashMap.put(..))) &&
20         args(key) && args(ProxyFactory.CacheKey)&&
21         withincode(* ProxyFactory.createClass2(..))
22     {
23         synchronized(key){
24             proceed(key);
25           }
26     }
27 }
```

図 5.13: fine-grained な同期を行うアスペクト

```
1  package javassist.util.proxy;
2  import java.lang.ref.WeakReference;
3  import java.util.*;
4  import javassist.bytecode.*;
5  public aspect CoarseGrainedSync{
6  package javassist.util.proxy;
7  import java.lang.ref.WeakReference;
8  import java.util.*;
9  import javassist.bytecode.*;
10 public aspect CoarseGrainedSync{
11     void around():
12         begin(call(* *.createClass2(..)))&&
13         end(call(* *.createClass3(..))) &&
14         withincode(* ProxyFactory.createClass(..))
15     {
16         synchronized(ProxyFactory.class){
17             proceed();
18         }
19     }
20 }
```

図 5.14: coarse-grained な同期を行うアスペクト

	fine-grained			coarse-grained		
スレッド数	平均 (秒)	中央値 (秒)	標準偏差	平均 (秒)	中央値 (秒)	標準偏差
5	8.57	8.58	0.16	11.89	11.91	0.18
10	8.55	8.54	0.23	11.92	11.95	0.19
20	8.62	8.63	0.34	11.94	11.96	0.18
40	8.97	8.96	0.36	11.94	11.95	0.17
80	9.24	9.17	0.59	12.03	12.04	0.21
100	9.22	9.19	0.52	12.09	12.09	0.19

表 5.11: 4 コア時の平均実行時間・標準偏差

	fine-grained		coarse-grained	
実行時間 (秒)	度数	累積	度数	累積
7 ∼ 8	1	1	0	0
8 ∼ 9	998	999	0	0
9 ∼ 10	1	1000	0	0
10 ∼ 11	0	1000	2	2
11 ∼ 12	0	1000	723	725
12 ∼ 13	0	1000	275	1000

表 5.12: 度数分布：4 コア, 5 スレッド

	fine-grained		coarse-grained	
実行時間 (秒)	度数	累積	度数	累積
1 ∼ 2	0	0	1	1
2 ∼ 3	0	0	0	1
3 ∼ 4	0	0	0	1
4 ∼ 5	0	0	0	1
5 ∼ 6	0	0	0	1
6 ∼ 7	0	0	0	1
7 ∼ 8	6	6	0	1
8 ∼ 9	961	967	0	1
9 ∼ 10	33	1000	0	1
10 ∼ 11	0	1000	1	2
11 ∼ 12	0	1000	624	626
12 ∼ 13	0	1000	374	1000

表 5.13: 度数分布：4 コア, 10 スレッド

	fine-grained		coarse-grained	
実行時間 (秒)	度数	累積	度数	累積
7 ∼ 8	22	22	0	0
8 ∼ 9	896	918	0	0
9 ∼ 10	81	999	0	0
10 ∼ 11	0	999	0	0
11 ∼ 12	0	999	616	616
12 ∼ 13	0	999	384	1000
13 ∼ 14	0	999	0	1000
14 ∼ 15	1	1000	0	1000

表 5.14: 度数分布：4 コア, 20 スレッド

	fine-grained		coarse-grained	
実行時間 (秒)	度数	累積	度数	累積
8 ∼ 9	563	563	0	0
9 ∼ 10	431	994	0	0
10 ∼ 11	6	1000	0	0
11 ∼ 12	0	1000	632	632
12 ∼ 13	0	1000	368	1000

表 5.15: 度数分布：4 コア, 40 スレッド

	fine-grained		coarse-grained	
実行時間 (秒)	度数	累積	度数	累積
8 ∼ 9	271	271	0	0
9 ∼ 10	698	969	0	0
10 ∼ 11	24	993	1	1
11 ∼ 12	2	995	404	405
12 ∼ 13	0	995	595	1000
13 ∼ 14	0	995	0	1000
14 ∼ 15	4	999	0	1000
15 ∼ 20	0	999	0	1000
19 ∼ 20	1	1000	0	1000

表 5.16: 度数分布：4 コア, 80 スレッド

	fine-grained		coarse-grained	
実行時間 (秒)	度数	累積	度数	累積
8 ∼ 9	275	275	0	0
9 ∼ 10	700	975	0	0
10 ∼ 11	24	999	0	0
11 ∼ 12	0	999	285	285
12 ∼ 13	0	999	715	1000
13 ∼ 22	0	999	0	1000
22 ∼ 23	1	1000	0	1000

表 5.17: 度数分布：4 コア, 100 スレッド

第6章　まとめ

本研究では，コードブロックの振る舞いを変更することができるアスペクト指向言語を提案し，実装した．本システムでは，コードブロックを，プログラムの意味や構造を破壊することなくポイントカットすることが出来る．これにより，同期処理，例外処理，ループの並列化などが，アスペクトとしてモジュール化できるようになった．

本システムでは，AspectJ コンパイラの実装の 1 つである abc(The AspectBench Compiler) を改造し，実装した．通常は消えてしまうソースコード上の文・ブロック・制御構造の開始位置・終了位置を中間表現上に記録し，それを解析することでポイントカットするブロックを選択する．また，アドバイスを正しく織り込むために，選択された範囲の命令列の書き換えを行う．

実験により，本システムを用いた場合，用いなかった場合に比べてオーバーヘッドが存在するが，実用上問題のないと思われる範囲であることを確認した．また，Javassist における同期処理の粒度の切り替えに本システムを適用し，粗い同期を行う方が良いの思われる状況が存在することを確認した．

6.1　今後の課題

6.1.1　指定可能なポイントカットの多様性

本システムではブロックを選択するために，begin,end,include,exclude, それぞれのポイントカットに，AspectJ の既存のポイントカットを与えて指定している．この時に指定出来るのは，call,set,get の 3 種類のポイントカットのみであり，ローカル変数への代入や，四則演算・比較など演算子を用いた演算などは指定できない．そのため，指定できない，若しくは指定しづらいブロックが存在する．これを解決し，より柔軟なブロックのポイントカットを行うことの出来るシステムが必要である．

6.1.2　ローカル変数の取得

本システムでは，ブロックの開始時点で有効なローカル変数をアドバイスで利用するために，変数の型を指定する．しかし，同じ型のローカル変数が複数宣言している場合，その宣言されている順番に依存してしまう．そのため，型以外の条件も指定できるようし，宣言されている順番に依存せずに，取得したいローカル変数を指定する方法を用意しなければならない．

参考文献

[1] : abc: The AspectBench Compiler for AspectJ, http://abc.comlab.ox.ac.uk/introduction.

[2] : The AspectJ Project, http://www.eclipse.org/aspectj/.

[3] : GluonJ, http://www.csg.is.titech.ac.jp/projects/gluonj/.

[4] : [#JASSIST-28] javassist enhancement failed on deserializing hibernate proxies - jboss.org JIRA, http://jira.jboss.org/jira/browse/JASSIST-28.

[5] : Polyglot extensible compiler framework, http://www.cs.cornell.edu/Projects/polyglot/.

[6] : Soot: a Java Optimization Framework, http://www.sable.mcgill.ca/soot/.

[7] 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: Proceedings of the 4th international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp. 87–98 (2005).

[8] Chiba, S.: Load-Time Structural Reflection in Java, *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, London, UK, Springer-Verlag, pp. 313–336 (2000).

[9] Harbulot, B. and Gurd, J. R.: A join point for loops in AspectJ, *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp. 63–74 (2006).

[10] Hoffman, K. and Eugster, P.: Bridging Java and AspectJ through explicit join points, *PPPJ '07: Proceedings of the 5th international*

symposium on Principles and practice of programming in Java, New York, NY, USA, ACM, pp. 63–72 (2007).

[11] IBM developerWorks: Java の理論と実践: JDK 5.0 における、より柔軟でスケーラブルなロック, http://www.ibm.com/developerworks/jp/java/library/j-jtp10264/.

[12] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: An Overview of AspectJ, *ECOOP '01*, LNCS, Vol. 2072, pp. 327–355 (2001).

[13] ティム・リンドホルム, フランク・イェリン: Java 仮想マシン仕様 第 2 版, ピアソン・エデュケーション (2001).

[14] 千葉滋: Javassist Home Page, http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html.

[15] 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).