Vol. 47 No. 4 情報処理学会論文誌 Apr. 2006

例外処理のためのアスペクト指向言語

熊原 奈津子[†] 光来 健一[†] 千葉 滋[†]

プログラムの実行環境の問題によって例外が発生した場合、その例外に対して適切な処理をすべきである。例外が発生したことを見落として正常時の動作を継続してしまうとより深刻で致命的な異常事態を招いてしまう恐れがある。しかし、プログラムを記述する際には、例外処理に関してより、ロジックを書くのに集中できた方がよい。また、ロジックを記述した後に必要に応じて例外処理を記述したい場合もある。例えば、サーバに負荷をかけて性能を測定するという大規模な実験を行うために複数クライアントを起動させる制御プログラムを作成する場合、必要に応じて例外処理を変更できるとよい。

このような例外処理記述を可能にするために、我々はアスペクト指向システム GluonJ/R を提案する。GluonJ/R がもつ `block` ポイントカット指定子を用いることでプログラム中の範囲を指定することができ、`recover` アドバイスを用いて指定した範囲内で例外が発生した場合の処理を記述できる。これにより、例外処理をプログラムロジックから分離して記述することができ、後から容易に追加削除できるようになる。また、GluonJ/R は指定した範囲の先頭に戻ってその範囲の処理を再実行することができるという、アドバイスの中で使える特殊なメソッドも提供している。

An Aspect-Oriented Language for Exception Handling

Natsuko Kumahara,[†] Kenichi Kourai[†] and Shigeru Chiba[†]

We must often handle exceptions raised due to a problem of the operation environment. If the exceptions are not handled, they will cause more serious problems. However, when developers are writing a main part of their program, they want to focus on the program logic rather than the exceptions. They may also want to add the code for handling exceptions after they finish writing the program logic. For example, they will want to do that when they are writing a program for measuring execution performance of some server software. To address this problem, we propose an aspect-oriented system named GluonJ/R. GluonJ/R provides a block pointcut for selecting a range of a given program. The selected range is associated with a recover advice, which can handle an exception raised in the range. Developers can thereby describe exception handling separately from the program logic and they can add and remove it on demand. This enables the programming style we mentioned above. Furthermore, a recover advice can call a special method, which reexecutes the range of the program associated with that advice.

1. はじめに

プログラムの実行環境の問題によって例外が発生した場合、その例外に対して適切な処理をすべきである。例外が発生したことを見落として正常時の動作を継続してしまうとより深刻で致命的な異常事態を招いてしまう恐れがある。

しかし、プログラムを記述する際には、例外処理に関してより、ロジックを書くのに集中できた方がよい。また、ロジックを記述した後に必要に応じて例外処理を記述したい場合もある。本稿ではこの問題点の例として、サーバに負荷をかけて性能を測定するという大規模な実験を行うために複数クライアントを起動させる制御プログラム作成する場合を例にとって考えていく。

このような例外処理に関する問題点を解決するために、我々は例外処理を行うのに特化したアスペクト指向システム GluonJ/R を提案する。GluonJ/R がもつ `block` ポイントカットを用いてプログラム中の範囲を指定し、その指定した範囲内で例外が発生した場合の処理を `recover` アドバイスを用いて記述する。これにより、例外処理をプログラムロジックから分離して記述することができるようになる。また、指定した範囲の先頭に戻ってその範囲の処理を再実行することが

† 東京工業大学 大学院 情報理工学研究科 数理・計算科学専攻
Dept. of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology

できるという、アドバイスの中で使える特殊なメソッドも提供している。

以下、2 章では例外処理を行う際の問題点とその具体例を示し、3 章ではこの問題を解決するために我々が提案する GluonJ/R について述べる。4 章では GluonJ/R の実装について述べ、5 章で GluonJ/R が出力したコードの性能を調べた実験について報告する。6 章では GluonJ/R の関連研究を取り上げ、7 章で本稿をまとめる。

2. 例外処理の記述

プログラムの実行環境の問題によって例外が発生した場合、その例外に対して適切な処理をすべきである。例として、分散環境上で動くサーバマシンの負荷テストをしており、サーバに対して負荷を発生させるプログラムを複数のクライアント上で起動する制御プログラムを作成しているとする。各クライアントに必要なファイルを送信する部分のプログラムは次のようになるだろう。

```
 1 class Sender{
 2   public void sendFile(String host,
 3       String fileName) throws Exception {
 4     int n;
 5     int port = 9000;
 6     byte[] buff = new byte[1024];
 7
 8     Socket s = new Socket(host, port);
 9     DataOutputStream out
10         = new DataOutputStream(
11             s.getOutputStream());
12     RandomAccessFile file
13         = new RandomAccessFile(fileName, "r");
14
15     while((n = file.read(buff)) > 0){
16       out.write(buff, 0, n);
17     }
18
19     file.close();
20     out.close();
21     s.close();
22     System.out.println(fileName
23             + " has been sent to " + host);
24   }
25 }
```

この場合、サーバとクライアントをつなぐネットワーク障害に障害が発生したときや、クライアントマシンがそもそも起動していないときに、例外が発生する可能性がある。例えば、8 行目で `UnknownHostException` や `IOException`、11 行目で `IOException`、13 行目で `FileNotFoundException` など、起こり得る例外はたくさんある。これらの例外は、もし発生すると、`sendFile()` メソッドの実行を中断し、`sendFile()` メソッドの呼び出し側に投げられる。

```
class Sender{
  public void sendFile(String host,
      String fileName) throws Exception {
    int n;
    int port = 9000;
    byte[] buff = new byte[1024];

        (1)

        Socket s = new Socket(host, port);
          :
        s.close();

        (2)

    System.out.println(fileName
              + " has been sent to " + host);
  }
}
```

図 1 実験プログラム

このような実験プログラムのロジックを記述する際には、例外の処理に関しては後で記述したいことが多い。なぜなら、例外処理を書かなくても多くの場合はうまく動くので、最初の段階ではロジックを書く方に集中できた方がよいからである。例外処理は、後で必要になったとき、はじめて実験プログラムに追加できると望ましい。

しかし、プログラムに例外処理を不用意に後から加えると、既に動いているプログラムを変更することになるため、そのロジックを壊してしまう危険性がある。また、追加した例外に修正や変更が加えられた場合には、対象となる例外処理を全て探し出して逐一変更しなければならなくなる。そのようなプログラムの追加は、面倒であるだけでなく、1 つでも変更し忘れるとプログラム全体の整合性がとれなくなる危険性がある。

追加する例外処理は、実験プログラムの例の場合、図 1 の四角で囲んだ部分に書くことになる。図 1 は例外処理について記述されていない、プログラムのロジックだけが書かれた実験プログラムの断片である。例えば図 1 の (1) に

```
try{
```

を追加し、(2) に

```
}catch(IOException e){
  e.printStackTrace();
}
```

などと追加することになるだろう。

ところが実験プログラムの場合、実験の内容によって発生する例外をどのように処理したいかが変わる可能性がある。例えば、例外処理の内容を、エラーログを画面に出力することから、実験者にメールを送信す

るように変えるとする。その場合、プログラム中の全ての catch 節を修正しなければならないが、修正もれがあると、全ての例外がメールによって送信されず、例外に気づくのに遅れてしまう恐れがある。

さらには、例外が起こった時点でプログラムの実行を止めずに、処理全体をやり直したいときもある。大きな実験プログラムを実行する場合、全体の処理時間が長くなるので、途中で例外が発生したからといって、実験を途中で止めて最初からやり直すのは好ましいとはいえない。例えば、サーバとクライアント間でネットワーク障害が起こった場合、時間をおいて再試行すれば障害が解決してうまくいく場合がある。また、クライアントマシンからの応答がなかった場合は代替のマシンに換えて再試行することで実験を続行できる場合もある。このように、プログラム全体を止めずに適切にリカバリ処理ができれば、それまでの実験の結果を無駄にせずに済む。

しかし、従来の Java 言語の範囲内ではリカバリ処理を記述するのは困難であった。つまり、プログラムの状態を、例外が起こらないような設定に変えて、もう一度同じ処理を繰り返させるような記述は、必ずしも容易ではなかった。例えば、try-catch 文の try ブロックの部分を catch 節の中から再試行したくても、そのような機能は Java 言語にはない。try ブロックの部分をメソッドにして catch 節の中でそのメソッドを呼ぶようにすれば、目的は達成できるが、catch 節の中にまた try-catch 文を書かなければならず、プログラムが見づらくなる。例えば図 1 の場合、

```
try {
  sendFileBody(host, fileName, port,
               buff, n);
} catch (IOException e) {
  host = getAnotherHost();
  try {
    sendFileBody(host, fileName, port,
                 buff, n);
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

のようになり、見づらい。なお sendFileBody() は元の try ブロックの中身を実行するメソッドであり、setAnotherHost() は代替マシンが存在すればそのホスト名を返すメソッドである。

try-catch 文を do-while 文で囲むことでリカバリ処理を実現する方法もある。つまり、図 1 の (1) に

```
Exception e = null;
do {
  try{
```

```
import javassist.gluonj.Glue;
import javassist.gluonj.Pcd;
import javassist.gluonj.Pointcut;
import javassist.gluonj.plugin.Block.Recover;

@Glue
class FileSenderRecovery {
  @Recover(etype = "java.io.IOException",
           advice = "{ $1 = getAnotherHost();"
         + " javassist.gluonj.GluonJR.retry(); }")
  Pointcut p = Pcd.block(
      Pcd.call("java.io.FileReader#new(..)"),
      Pcd.call("java.io.PrintStream#println(..)"));
}
```

図 2 アスペクトの記述例

を追加し、(2) に

```
    } catch (IOException err) {
      e = err;
      host = getAnotherHost();
    }
  } while (e != null);
```

を追加すれば、リカバリ処理を実装できる。しかし、先の方法と同様、プログラムが見づらくなる。

3. GluonJ/R

前章の問題点を解決するため、例外処理をアスペクトとして記述できるようにしたアスペクト指向システム *GluonJ/R* (GluonJ with Recovery) を提案する。例外処理をアスペクトとすることで、プログラムロジックから、その中で起こった例外の処理を分離して記述することができるようになる。

3.1 block ポイントカットと recover アドバイス

GluonJ/R は、AspectJ[2)3)] のような一般的なアスペクト指向システムがもつ機能に加えて、block **ポイントカット** と recover **アドバイス**を提供している。block ポイントカットは、2 つのジョインポイントの組を選択するためのポイントカット指定子である。block ポイントカットによって選ばれたジョインポイントの組で囲まれた範囲で例外が発生したときに実行されるコードが recover アドバイスである。なお block ポイントカットで指定される範囲は、同一メソッドの中に含まれていなければならない。

例えば、2 章で示したファイルを送信するプログラムの例外処理を、block ポイントカットと recover アドバイスを使ってアスペクトとして書くと、図 2 のようになる。GluonJ/R は 我々の研究室で開発したアスペクト指向システム GluonJ[1)] を拡張したものなので、GluonJ の文法に従って書くことになる。GluonJ では @Glue で注釈されたクラスがアスペクトになる。

そして、そのアスペクト内で Pointcut 型のフィールド（ポイントカット・フィールドという）を宣言することでポイントカットを指定することができる。具体的には Pcd クラスが持つメソッドを利用してポイントカットを指定する。

block ポイントカットは Pcd クラスに存在するメソッド block を用いて指定し、メソッド block の引数には2つのポイントカットのペアを渡す。このポイントカットのペアで例外処理を追加したい範囲の始点と終点を宣言する。図 2 では、範囲の始点と終点を call メソッドを用いて宣言しているが、この call メソッドの引数となる文字列は

（クラス名）#（メソッド名（引数））

で記述する。このとき、new というメソッドはコンストラクタ（オブジェクト生成時）をポイントカットし、メソッド名の引数は .. で省略することができる。call の他にも set や get といったメソッドも存在し、

（クラス名）#（フィールド名）

と記述することでフィールドの値を読み込むときや書き込むときを指定することができる。

recover アドバイスを追加したい場合は、宣言されたポイントカット・フィールドを @Recover アノテーションで注釈する。そして図 2 の様に etype に処理したい例外の型を、advice にアドバイスとして例外処理の内容を記述する。このアドバイスの中では、変数 $1 を sendFile() メソッドの第一引数を表す変数として利用している。また、recover アドバイスの中で GluonJR.retry() が呼ばれているが、これは block ポイントカットで指定された範囲の先頭に戻って、その範囲の処理を再実行するためのメソッドで、リカバリ処理の記述のために、GluonJ/R が提供する特殊なメソッドである。

図 2 のアスペクトを、図 1 のプログラムに織り込む (weave する) と、以下の try-catch 文を使ったプログラムと同等の振る舞いをするプログラムが得られる。

```
class Sender{
  public void sendFile(String host,
      String fileName) throws Exception {
    int n;
    int port = 9000;
    byte[] buff = new byte[1024];

    again:
    try{
      Socket s = new Socket(host, port);
        :
      s.close();
    }catch(IOException e){
      host = getAnotherHost();
      goto again;
    }
    System.out.println(fileName
              + " has been sent to " + host);
  }
}
```

ただし、catch 節の中で呼ばれている goto は Java の文法ではない。

3.2 行アノテーション

現実には指定したい範囲の直前や直後に適当なジョインポイントがない場合がある。例えば、下のコードにあるように if 文の直後に for 文が書かれてある場合、if 文の直後すなわち for 文の直前には適当なジョインポイントは存在しないので、for 文の前後を範囲とする block ポイントカットはそのままでは定義できない。

```
if(){
    :
}else {
    :
}
for(){
    :
}
    :
```

このような場合は、GluonJ/R が提供する**行アノテーション**を用いて指定する。行アノテーションとはユーザ定義のジョインポイントであり、メソッド中の特定の行に対してつけられるアノテーションのことを指す。

将来例外を捕まえる範囲として指定されそうな箇所にあらかじめユーザが目印として行アノテーションを記述しておくことで後でジョインポイントとして利用できる。以下に、上の文の for 文の前後に行アノテーションを付加した例を示す。

```
if(){
    :
}else {
    :
}
@Line(begin)
for(){
    :
}
@Line(end)
    :
```

block ポイントカットを

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

のように記述することでこの行アノテーションで囲まれた範囲をポイントカットすることができる。

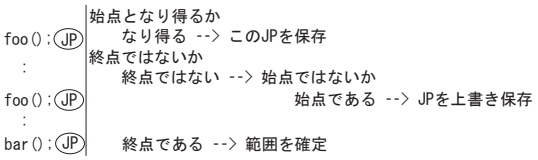

図 3 foo() から bar() までの範囲が選択されるアルゴリズム

4. 実　　装

我々は我々の研究室で開発したアスペクト指向システム GluonJ を拡張して GluonJ/R を実装した。block ポイントカットで指定された 2 つのジョインポイントで囲まれた範囲を try ブロックとし、アドバイスとして書かれたコードを catch 節の中身とした try-catch 文を元のプログラムにバイトコード変換で埋め込む。バイトコード変換には Javassist[6] を利用した。

4.1 block ポイントカット

GluonJ では、ユーザによって宣言されたポイントカットを見つけるとその宣言されたポイントカットを表現する抽象構文木をポイントカットノードを組み合わせて生成する。ポイントカットノードの種類には、クラス名やメソッド名を表現する文字列をフィールドに持ち、その文字列に一致するメソッド呼び出しをポイントカットする call ポイントカット等がある。そして、ソースコード内のジョインポイントにぶつかる度に生成された構文木を巡回し、ポイントカットすべきジョインポイントなのかを判定するという仕組みになっている。抽象構文木を構成するポイントカットノードの種類に、2 つのポイントカットのペアをフィールドとして持つ Block ポイントカットノードを追加して block ポイントカットを実装している。この 2 つのフィールドとなるジョインポイントは GluonJ に既にあるポイントカット（call ポイントカット等）で表現される。

次に、block ポイントカットによって選択されるべき範囲を見つけるアルゴリズムについて述べる。このアルゴリズムは始点と終点となり得るジョインポイントが複数存在する場合はそれらのうちで一番近いペア同士が範囲として選択されるように設計されている。図 3 に示す様に、まず、ジョインポイントが範囲の始点となり得るかをチェックしていく。始点の候補となるジョインポイント（1 番目の foo() の呼び出し）が現れた場合、そのジョインポイントを一時的に保存しておく。始点の候補となるジョインポイントが現れた後は、ジョインポイントを見つけると、保存してある始点と対となる終点のジョインポイントにならないかをチェックする。終点ではないと判定されると、始点のジョインポイントとならないかをチェックする。そこでもし始点となるジョインポイントだと判定された場合は保存しておいたジョインポイントを上書きして保存する。（2 番目の foo() の呼び出し）始点の候補が存在している間に、終点のジョインポイント（bar() の呼び出し）が見つかるとその時点で範囲を確定し、始点と終点を対にして保存する。そして、他にもポイントカットすべき範囲がないかをチェックするために、始点となり得るかをチェックする段階から新たな範囲を探していく。

4.2 recover アドバイス

クラスファイルの中には各メソッドの情報が含まれており、そのメソッドの情報の中にはメソッドの属性が含まれている。メソッドの属性の中にはそのメソッドを実装しているバイトコードのほかに Exception Table という表が書かれている。この表に含まれる情報としては、

- 例外ハンドラがアクティブとなるバイトコードの始点と終点
- 例外が生じた場合に実行するバイトコードの先頭
- 例外ハンドラがキャッチする例外のクラス

がある。

まず、recover アドバイスで指定された例外が生じた場合に実行したいコードを、ポイントカットされた範囲の始点と終点が存在するメソッドのバイトコードの末尾に追加する。そして、追加したバイトコードの先頭のインデックスを Exception Table に追加する。次に、block ポイントカットで指定された例外処理を追加したいソースコードの始点と終点の情報をもとに例外ハンドラがアクティブとなるバイトコードの始点と終点を求める。そして、etype で指定された処理したい例外の型と共に Exception Table に追加する。例外が発生した場合、当てはまるかどうかは表の順番通りに検査されるため、アスペクトで追加された例外処理の優先順位が高くなるように表の最初に追加している。

4.3 特殊メソッド GluonJR.retry()

3.1 章でも述べたように、GluonJR.retry() はリカバリ処理が容易に記述できるアドバイス内で利用可能な特殊メソッドである。この static メソッドをコンパイルすると invokestatic という命令長が 3 バイトのバイトコードに変換される。この命令を同じ 3 バイトの命令長の goto 命令に置換することで block ポイントカットで指定した範囲の先頭に戻って再試行することを実現している。

4.4 アドバイスの最後で呼ばれるメソッド

アドバイスの最後には自動的に GluonJR.doNext() メソッドを追加している。このメソッドは GluonJR.retry() と同じく、static メソッドを goto 命令に変換することで実現している。このメソッドはソースコードに try-catch 文を追加したときと同様、catch 節を実行し終わった後でその下のコードを実行するようにするために必要となる。つまり、goto の飛び先は catch 節のすぐ下の、block ポイントカットで指定した終点に相当するコードである。このように goto で飛ぶ先を明確に示さないとバイトコード検査器によって VerifyError☆が投げられてしまう。

4.5 バイトコード変換

GluonJ/R はバイトコード変換によって try-catch 文を挿入することで recover アドバイスを実現するが、単純な変換ではうまくいかない。3 章で示した図 2 の FileSenderRecovery アスペクトでは、call ポイントカットによって、Scoket オブジェクトの生成時を表すジョインポイントを選択していた。これはプログラムの次の行に該当する。

```
Socket s = new Socket(host, port);
```

プログラムを Java コンパイラでコンパイルして得られるバイトコードのうち、上の行に対応するバイトコードは以下のようになる。

```
new Socket
dup
aload_1
iload 4
invokespecial Socket()
astore 5
   :
```

aload_1 と iload 4 は、コンストラクタの引数をスタックに積むための命令である。コンストラクタの呼び出しは invokespecial 命令である。

AspectJ に代表される通常のアスペクト指向システムでは、一般に、ジョインポイントを invokespecial 命令の実行時と解釈する。ところが、この命令を try ブロックの始点と考えると、retry() 命令の実現が困難になる。単純に goto 命令などで、invokespecial 命令から実行を再開しようとすると、コンストラクタの引数がスタックに積まれていない状態で、コンストラクタを呼ぼうとしてしまう。これは不正な実行なので、バイトコードを Java 仮想機械にロードする段階で、バイトコード検査器が不正なコードとしてロードを拒否し、VerifyError が投げられてしまう。

この問題を回避するため、GluonJ/R では、ジョインポイントに該当するソースプログラムの行を構成するバイトコード列の先頭を、block ポイントカットによって選択される範囲の境界として用いる。先の例では、先頭の new 命令を境界として用いる。これによって、Java 仮想機械のスタックの状態が常に整合性に保った状態であることを保障する。一般的な Java コンパイラが生成するバイトコードでは、行の境界では必ずスタックが空になるので、整合性が保障できる。なお、実現にあたっては、Java クラスファイル内に記録されているソースプログラムの行と各バイトコード命令との対応関係を表す情報を利用している。

4.6 行アノテーション

行アノテーションに関しては、ソースプログラムを読み込み、行アノテーションを空の static メソッド呼び出しに置換するプリプロセッサによって実現している。呼ばれる static メソッドとして、引数も返り値もないメソッドを選べば、メソッドが呼ばれた場所（call ポイントカット）で自由にジョインポイントを指定することができる。また、メソッドの中身は空のためプログラムの実行には影響を与えない。

プリプロセッサは、ソースコード中の

```
@Line(begin)
```

という文字列を

```
LineAnnotation.begin();
```

という static メソッド呼び出しに置換する。次に、LineAnnotation クラスに

```
public static void begin(){}
```

というメソッドを追加した後、LineAnnotation クラスを再コンパイルするという実装方法である。

5. 実　　験

GluonJ バージョン 1.3 を拡張して開発した GluonJ/R で例外処理をアスペクトとして追加した場合のオーバーヘッドを測定するのを目的として以下のような実験を行った。実験環境は、CPU は Intel® Pentium® 4 CPU 2.8GHz、メモリは 1 GB、OS は Microsoft Windows XP Professional Service Pack 2、JVM のバージョンは 1.5.0_06 である。

5.1 try-catch 文との実行速度の比較

図 4 のメソッド m() を 10 億回呼んで時間を測定するというマイクロベンチマークを走らせて、ソースコードに直接 try-catch 文を書いたものと GluonJ/R で例外処理を追加したものの実行時間を比較してみた。

このベンチマークを走らせたものと、メソッド m() の中身を

```
try{
  a();
```

☆ バイトコードが不正であると検証された場合に投げられるエラー

```
public class Test {
  public void m() throws Exception{
    a();
    b();
  }

  public void a() throws Exception{}
  public void b() {}

}
```

図 4 マイクロベンチマーク

```
@Glue
class InsertTryCatch {
  @Recover(etype = "java.lang.Exception",
           advice = "")
  Pointcut p = Pcd.block(
      Pcd.call("test.Test#a(..)"),
      Pcd.call("test.Test#b(..)"));
}
```

図 5 織り込んだアスペクト

	実行時間 (秒)
元のプログラム（例外処理なし）	2.638
try-catch 文を追加	17.064
GluonJ/R で追加	17.046

表 1 try-catch 文と GluonJ/R の実行時間の比較

```
  } catch(Exception e){}
  b();
```

のように try-catch 文を追加して書き換えたもの、GluonJ/R で書いた図 5 のようなアスペクトを織り込んだ場合の実行時間を比較してみた。これにより得られた結果は表 1 の通りである。

この結果を見てわかるように、ソースコードに直接 try-catch 文を記述するより GluonJ/R を用いて例外処理を追加した方が実行時間は若干短い。

その原因を探るために、try-catch 文を直接ソースコードに追加したものと GluonJ/R を用いて例外処理を追加したもののメソッド m() のバイトコードを比較してみた。それぞれのバイトコードは図 6 と 7 の通りである。

```
   0: aload_0
   1: invokevirtual #20; //Method a:()V
   4: goto 8
   7: astore_1
   8: aload_0
   9: invokevirtual #23; //Method b:()V
   12: return
  Exception table:
   from   to  target type
     0     7     7   Class java/lang/Exception
```

図 6 try-catch 文を直接ソースコードに追加した場合のメソッド m() を実装するバイトコード

```
   0: aload_0
   1: invokevirtual #20; //Method a:()V
   4: aload_0
   5: invokevirtual #23; //Method b:()V
   8: return
   9: astore_1
   10: goto 4
  Exception table:
   from   to  target type
     0     4     9   Class java/lang/Exception
```

図 7 GluonJ/R を用いて例外処理を追加した場合のメソッド m() を実装するバイトコード

```
@Glue
class InsertTryCatch {
  @Recover(etype = "java.lang.Exception",
           advice = "")
  Pointcut p = Pcd.block(Pcd.line("begin"),
                         Pcd.line("end"));
}
```

図 8 行アノテーションを範囲の始点・終点として例外処理を追加するアスペクト

	実行時間 (秒)
行アノテーション追加（例外処理なし）	2.652
行アノテーション追加（例外処理追加）	17.121

表 2 行アノテーション利用時の実行時間の比較

これらを見て分かるように、例外が生じない場合、GluonJ/R で例外処理を追加した場合では 0 から 8 番目のバイトコードを実行するのに対してソースコードに直接 try-catch 文を追加した場合では 0,1 番目のバイトコードを実行した後、goto 命令で 8 に飛び、12 番目まで実行する。つまり GluonJ/R で追加した方が goto 命令一つ分実行せずに済んでいることが分かる。これらが GluonJ/R を用いたときの方が実行時間が短くなった原因と考えられる。

5.2 行アノテーションの性能実験

4.6 章で行アノテーションはプログラムの実行には影響を与えないと述べたが、実行速度にはどのくらい影響するかについて調べてみた。

図 4 のプログラムに対して、メソッド m() の中身を

```
@Line(begin)
a();
@Line(end)
b();
```

のように行アノテーションを追加したものと図 8 のようなアスペクトを織り込んで、行アノテーションを始点・終点とする範囲に例外処理を追加したものの実行速度を測定してみた。得られた結果は表 2 の通りである。

表 1 と表 2 の結果を比較すると、例外処理を追加しない場合の行アノテーションのオーバーヘッドは 0.02

秒以内で、例外処理を追加した場合で 0.08 秒程度遅くなっただけであることが分かる。これは JIT により最適化が行われたためと考えられる。行アノテーション、つまり変換後の static メソッドがプログラム全体において占める割合がこのように高い場合でもこの程度のオーバーヘッドしかないので、実際のプログラムでは行アノテーションのオーバーヘッドはほぼないと考えられる。

6. 関連研究

6.1 AspectJ

AspectJ を利用して、プログラム中の例外処理の分離を試みた研究がいくつか提案されている[4)5)]。特に Lippert らは、既存のソフトウェア JWAM[7)] 中に記述されている例外処理を AspectJ 0.4 で分離することを試みた。JWAM 内に記述されている例外処理はソフトウェア全体に散らばっており、それらの例外処理のコードは互いに似ている。Lippert らは散らばっているこれらの例外処理を、AspectJ を利用して一箇所にまとめることにより、ソフトウェア全体のコードサイズを減らすことができたと報告した。しかし、既存の AspectJ では、`block` ポイントカットのように任意の範囲をポイントカットし、その範囲の例外処理を追加することはできない。また、例外処理を分離して記述することは可能でも、GluonJ/R のようなリカバリー処理 (retry) を実現することは困難である。このため、本論文の 2 章で取り上げた実験プログラムの例外処理を AspectJ で記述するのは適切ではない。

また AspectJ は、バージョン 0.6 以降から、例外処理に関連する言語機構が 2 つ提供された。それらは handler ポイントカット指定子と after throwing アドバイスである。handler は、例外ハンドラの実行時、つまり catch 節の実行時のみをジョインポイントとして選択する。このため、今回の実験プログラムのように予め `try-catch` 文が使われていない場合、このポイントカット指定子は有効ではない。一方、after throwing アドバイスは、選択されたジョインポイントが例外を投げて異常終了したときに実行される。ただし、このアドバイスは catch 節のように例外を捕まえるわけではなく、暗黙のうちに実行され、例外によるメソッドの異常終了の連鎖を止めるわけではない。それゆえ、after throwing を利用してリカバリ処理を実装することは困難である。

6.2 Eiffel[8)9)10)11)] や Ruby[12)] の retry 機構

オブジェクト指向言語である Eiffel や Ruby には GluonJ/R が提供する特殊メソッド `GluonJR.retry()` と同様の機構が存在する。これらの言語には再試行を実現する機構がもともと組み込まれているが Java には存在しない。GluonJ/R ではバイトコード変換により Java で再試行を実現する機構を実現している。

6.3 ループのためのジョインポイント

3.2 章で行アノテーションを用いてソースコードの任意の範囲をポイントカットする方法について述べたが、ループをポイントカットする研究は存在している。[13)] この研究では、バイトコードからループを見つけ出し、ジョインポイントを提供する。しかしこのアプローチでは、ポイントカットできないループも数多く存在している。それゆえ、ループについても、`block` ポイントカットと行アノテーションは有用であると考えられる。

7. まとめ・今後の課題

本稿において我々は例外処理をアドバイスとして扱えるようにしたアスペクト指向システム GluonJ/R を提案した。プログラムロジックから例外処理を分離して記述でき、例外処理の中で再試行できるように書ける事を示した。

GluonJ/R はバイトコード変換で例外処理をプログラムロジックに埋め込む。その時、バイトコード上のジョインポイントを選択するのではなくクラスファイルに含まれる行番号を利用して、ソースコード中のジョインポイントを選択する。このようにすることで、`VerifyError` が起こるのを回避することが出来ることを示した。また、例外処理をアスペクトとして分離して記述してもソースコードに直接 `try-catch` 文を記述した場合とほぼ実行時間は変わらないことが分かった。

今後は、既存のアプリケーションを GluonJ/R を用いて例外処理を記述した場合、うまく記述できるか、コードサイズをどのくらい減らすことができるかについて検証したいと考えている。

参考文献

1) GluonJ Web Site. `http://www.csg.is.titech.ac.jp/projects/gluonj/`.
2) AspectJ Web Site. `http://www.eclipse.org/aspectj/`.
3) Gregor Kiczales and Erik Hilsdale and Jim Hugunin and Mik Kersten and Jerey Palm and William G. Griswold. An Overview of AspectJ. *European Conference on Object Oriented Programming (ECOOP)*, pages 327-353, Jun 2001.
4) Martin Lippert and Cristina Videira Lopes.

A study on exception detection and handling using aspect-oriented programming. *International Conference on Software Engineering (ICSE)*, pages 418-427, Jun 2000.
5) F. Filho, C. Rubira, A. Garci. A Quantitative Study on the Aspectization of Exception Handling. Workshop on Exception Handling in OO Systems. *International Conference on Software Engineering (ICSE)*, pages 418-427, July 2005.
6) Shigeru Chiba. Load-Time Structural Reflection in Java. *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP '00)*, pages 313-336, London, UK, 2000. Springer-Verlag.
7) JWAM framework Web Site. `http://www.jwam.de/`.
8) Bertrand Meyer. Eiffel: the Language. Object-Oriented. Upper Saddle River, NJ, USA, 1992, ISBN 0-13-247925-7. Prentice-Hall.
9) Eiffel Software Web Site. `http://www.eiffel.com/`.
10) the NICE (the Nonprofit International Consortium for Eiffel) Web Site. `http://www.eiffel-nice.org/`.
11) Standard ECMA-367 Web Site. `http://www.ecma-international.org/publications/standards/Ecma-367.htm`.
12) Ruby Web Site. `http://www.ruby-lang.org/`.
13) Bruno Harbulot and John R. Gurd. A join point for loops in AspectJ. *Proceedings of the 5th international conference on Aspect-oriented software development (AOSD '06)*, pages 63-74, Bonn, Germany, 2006.