# 分散 Java プログラムのためのアスペクト指向言語

西 澤 無 我† 千 葉 滋†

本発表では、分散プログラム内に存在する横断的な関心事を、他のモジュールから分離して記述することが可能なアスペクト指向言語 DJcutter を提案する。横断的な関心事は、複数のクラスまたはモジュールにまたがってしまう処理の事を指し、既存のオブジェクト指向プログラミングではうまく扱えない。DJcutter は、AspectJ のサブセットをもとに、遠隔ホスト上の join point も pointcut で抽出できるように拡張した言語である。これにより DJcutter では、異なるホストにまたがる observer パターンのような、分散プログラム内に存在する横断的な関心事を、アスペクトと呼ばれる一つのモジュールにまとめて記述できる。汎用的なアスペクト指向言語である AspectJ などは、ネットワーク越しの横断的関心事を直接的にサポートしていない。一方、DJcutter は、AspectJ で解決可能なクラス間にまたがる横断的関心事はもちろん、異なるホスト間にまたがる横断的関心事も一つにまとめて記述できる。

# An Aspect-Oriented Programming Language for Distributed Programming in Java

Muga Nishizawa† and Shigeru Chiba†

We present an Aspect-Oriented Programming (AOP) language, which is called *DJcutter*. It allows developers to separate crosscutting concerns in distributed Java programs. Crosscutting concerns are ones cutting across several classes or modules and thus object-oriented languages cannot separate them into an independent module. DJcutter is a subset of AspectJ but, unlike AspectJ, it enables pointcuts to identify join points on different hosts. Thereby, DJcutter can separate distributed crosscutting concerns, such as the Observer pattern involving several hosts, into a module called an aspect. AspectJ, which is a general-purpose AOP language, cannot deal with these distributed crosscutting concerns. On the other hand, DJcutter can deal with not only local crosscutting concerns, which AspectJ can handle, but also distributed crosscutting concerns.

## 1. はじめに

今日、分散ソフトウェアの開発にかかるコストが問題となっている。分散ソフトウェアの開発は、ネットワーク処理などの分散特有の問題に対処しなくてはならず、通常のソフトウェアに比べ、可読性が低く、修正・変更が困難になりがちである。このため、非分散プログラムの作成に比べて、分散プログラムの作成や維持にかかる人的コストは飛躍的大きくなりがちである。

このため、様々な分散プログラミング支援のツールやミドルウェア[9)10)12)] が開発され、利用者は分散プログラムを以前よりも容易に記述することが可能になった。例えば、Java 言語であれば標準で用意されている *Java RMI*[12)] が利用できる。Java RMI は、遠隔ホスト上に生成したオブジェクトのメソッドを、同一ホスト上のメソッド呼び出しと同様な方法で呼び出せるようにするためのミドルウェアである。これにより、利用者はプログラムを煩雑にしがちだったネットワーク処理をあまり意識する事なく、分散プログラムを開発できるようになる。

ところが最近、異なるクラス間にまたがってしまいモジュール化しきれない処理、すなわち横断的関心事がプログラムに存在する事が問題となっている[1)]。この横断的関心事はオブジェクト指向プログラミング (OOP) を用いて開発されたプログラムの多くに存在し、プログラムを煩雑で見にくいものにしてしまう。プログラムの可読性が低下する事で、ソフトウェアの拡張・維持が困難になる。

この横断的関心事をモジュール化する技術として、アスペクト指向プログラミングが注目を浴びている[1)4)5)]。その中でも代表的なのが、横断的関心事をアスペクトと呼ばれるモジュールに分離して記述でき

† 東京工業大学大学院 数理・計算科学専攻
Tokyo Institute of Technology
Dept. of Mathematical and Computing Sciences

る Java の拡張言語、*AspectJ*[1] である。このような AOP 言語により、多くの非分散プログラムの可読性は向上したが、このような言語では、分散プログラム内に存在する横断的関心事のモジュール化は容易ではない[6]。

本発表では、分散プログラム内に存在する横断的関心事をモジュール化するためのアスペクト指向言語 DJcutter を提案する。この言語では、遠隔クラス内の join point を抽出することが可能である。これは AspectJ 言語のサブセットを拡張した言語であり、分散プログラム内の横断的関心事を AspectJ のアスペクトと同様の記述で分離する事が可能である。

以後、2 章では分散プログラム内の横断的関心事の例を示す。3 章では、我々の提案する AOP 言語 DJcutter の特徴、言語仕様そして実装方法を説明する。4 章で我々は DJcutter を用いた分散プログラム内に存在する横断的関心事のモジュール化として、2 章で説明する分散プログラムを例に出す。5 章では DJcutter の性能測定実験の結果を示し、6 章で関連研究について述べ、そして最後に 7 章で本発表をまとめる。

## 2. 分散プログラム内に存在する横断的関心事

### 2.1 横断的関心事とは

最近、異なるクラス間にまたがってしまう処理、すなわち横断的関心事 (*crosscutting concern*) のモジュール化技術が提案されている。横断的関心事はオブジェクト指向プログラミング (*OOP*) を用いて開発されたプログラムの多くに存在し、プログラムを煩雑で見にくいものにしてしまう。ここで横断的関心事の例として、*observer* パターンを用いて実装されている図形エディタのプログラム[2] を考える (**図 1**)。描こうとしている図形要素の抽象クラスを `FigureElement` クラス、ユーザが操作するスクリーンを表すクラスを `Screen` とする。点や線を表す `Point`, `Line` クラスは `FiguareElement` のサブクラスである。Observer パターンは、エディタの利用者によって各図形要素の位置が変更されたときに、連動してスクリーンを再描画する処理に使われる。ここで subject は `Point`, `Line` クラスであり、observer は `Screen` クラスである。例えば、点の位置を変更しようとして `Point` オブジェクトの `setX, setY` メソッドが呼ばれると、それらのメソッドは位置変化を observer に通知するため、`FigureElement` クラスの `notify` メソッドを呼ぶ。`notify` は、登録されている `Screen` オブジェクトの `update` メソッドを呼び、スクリーンを再描画させるので、これによって図形の位置とスクリーンの内容を連動させることができる。

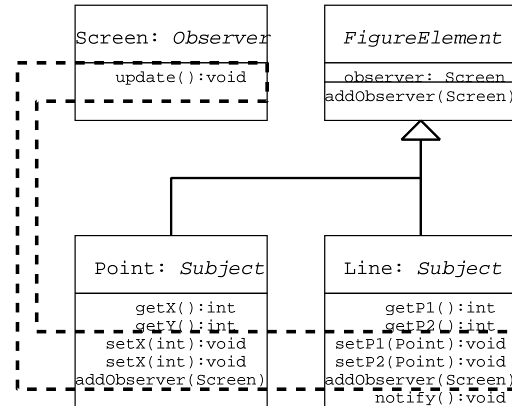


**図 1** 図形エディタのモデル

各図形要素の位置の変化とスクリーンの再描画を連動させる処理は、開発上のひとつの関心事であるが、関連するメソッドが多数のクラスに散らばっている横断的関心事である。本来、ひとつの関心事は、ひとつのモジュール、つまりクラスにまとめられているべきだが、関心事が横断的であると、プログラムの可読性が下がる。処理の流れを把握するためには、多数のクラスの定義を読まなければならないからである。例えば、図形要素の位置を変更する全てのメソッドが、正しく `notify` を読んでいるか確認する作業も容易ではない。

### 2.2 AspectJ: 汎用的な AOP 言語

現在、OOP 技術ではモジュール化しきれない横断的関心事のモジュール化の技術として、アスペクト指向プログラミング (*AOP*) が注目を浴びている。その中でも代表的なのが、横断的関心事をアスペクト (*aspect*) と呼ばれるモジュールに分離して記述できる Java の拡張言語、*AspectJ* である。

AspectJ 言語の基本要素は *join point*、*pointcut*、そして *advice* である。Join point とはメソッド呼び出し、メソッド実行、フィールドアクセスなどのプログラム実行の基本要素である。advice とは、非 AOP では異なるクラスのメソッドとして散らばってしまうコードの断片のことである。AspectJ では、関連する advice をひとつのモジュールにまとめた上で、それぞれの advice がどの join point で実行されるか指定できる。このモジュールをアスペクトといい、各 advice を実行する join point の指定を pointcut という。pointcut はまた join point の情報を引数として advice に渡すことができる。この引数を pointcut 引

数という。アスペクトとクラスをまとめてコンパイルすることを weave と呼ぶ。

この AspectJ 言語を利用して、先述した図形エディタ内の横断的関心事をモジュール化したアスペクトを**図 2** に示す。このアスペクトは `subjectChange` という pointcut を定義する。これは subject の位置を変更する全てのメソッド呼び出しの join point を指定する pointcut である。`call`, `target` は AspectJ のプリミティブであり、それぞれプログラム中の join point を指定するための述語である。`after` から始まる部分は advice で、これを実行すると subject に登録されている observer の `update` メソッドを呼び出す (説明を簡単にするため observer は 1 つと仮定する)。この adivce は `after` から始まるので、`subjectChange` で指定される join point の直後に実行される。advice は join point の直後 (`before`) に実行したり、join point 自体の代わり (`around`) に実行することもできる。

```
aspect CoodinationAspect {
  pointcut subjectChange(Subject s):
    (call(void Point.setX(int))
    || call(void Point.setY(int))
    || call(void Line.setP1(Point))
    || call(void Line.setP2(Point)))
    && target(s);

  after(Subject s): subjectChange(s) {
     s.observer.update();
  }
}
```

**図 2** 図形エディタ内の横断的関心事をモジュール化する AspectJ のアスペクト

この AspectJ のアスペクト記述は、横断的関心事をモジュール化しているといえる。非 AOP では `Point` や `Line` に散らばってしまう `update` メソッドの呼び出しが、AspectJ では、ひとつのアスペクト記述の中にまとめられる。`Point` や `Line` には、`update` の呼び出しに関係するコードは一切書かなくてよい。

### 2.3 分散プログラム内の横断的関心事

このような observer パターンが当てはまる事例は分散プログラム内にも存在する。しかしそのような場合、分散した observer パターンを AspectJ を利用してモジュール化することは容易ではない。それは AspectJ がローカルホストに存在する join point の抽出しかサポートしていないからである。

ここで、先の図形エディタを分散化してグループで同時に使える図形エディタを例として考える。この分散図形エディタの実装では、`Screen` オブジェクトだけでなく、図形要素を表す `Point` や `Line` オブジェクトも、各利用者のホストごとにコピーが作られるとする。図形の位置が変わった際には、全てのホスト上のコピーの状態が更新され、連動して全てのスクリーンも再描画されるとする。

このようなホスト間の連携も observer パターンを用いて実装できる。しかし AspectJ を利用してこの observer パターンをモジュール化するのは容易ではない。pointcut で抽出する必要のある join point が複数のホスト上にあるが、AspectJ はローカルホスト上の join point しか抽出できない。そこで advice のコードを各ホスト上に配置し、抽出された個々の join point に制御が達したら、同一ホスト上の advice コードを実行するようにしなければならない。Advice の中では、関連する全ての observer の `update` メソッドを、遠隔参照を使って呼び出すことになる。

ところが、advice コードがそれぞれのホスト上に散らばって配置されるのはプログラミング上好ましくない。Advice コードの実行に必要な遠隔参照の値等のデータを、それぞれのホストに配布し、そのデータが変更された場合には一貫性を保つように全体を更新しなければならないからである。このような分散データの管理は困難である。

## 3. DJcutter: 分散プログラム用 AOP 言語

我々は、異なるホスト上の join point を抽出することのできる AOP 言語、DJcutter を提案する。DJcutter は AspectJ のサブセットを拡張した言語であり、ユーザは AspectJ とほぼ同様の構文でアスペクトを記述し、分散プログラム内の横断的関心事をモジュール化することができる。

### 3.1 DJcutter の特徴

DJcutter の特徴は次の 2 点である。

**遠隔ホスト上の join point の抽出**

DJcutter では、advice は指定されたホスト上で動くアスペクト・サーバによって実行される。この advice が実行されるのは、他のホスト上の制御の流れが、その advice に対応する pointcut によって抽出された join point に到達したときである。同一アスペクトに属する advice は同一ホスト上に集中して実行されるので、advice の実行に必要なデータを各ホストに配布し、一貫性を維持する手間が省ける。

**Pointcut 引数の拡張**

advice が join point の存在するホストとは異なるホストで実行されるため、pointcut 引数は advice から見て異なるホスト上のオブジェクトを表すことになる。したがってユーザは、pointcut 引数を advice に渡すとき、オブジェクトのコピーを渡すか、あるいは元のオブジェクトの遠隔参照を作って渡すか、選択することができる。オブジェクトへの遠隔参照はプロキシ・オブジェクトによって実現される。これにより、advice は遠隔ホスト上の join point の実行時情報を得て必要な処理をおこなうことができる。

### 3.2 アスペクトの定義

DJcutter のアスペクトの定義およびメンバの記述方法は、AspectJ の核となる機能を取り出したサブセットを元にしている。現在の所、DJcutter はまだ AspectJ の抽象アスペクトや、per-object アスペクト、introduction などに対応していない。

**Pointcut**

現在 DJcutter が提供する pointcut 指定子は **図 3** のとおりである。これらの中に、ホストを特定するための指定子は用意されていない。ホスト名を直接 pointcut 指定子の中に埋め込むと、プログラムを動かすホストを変えるたびにアスペクトを修正しなければならなくなり、アスペクトの再利用性が著しく低下してしまう。これを避けるには、pointcut 指定子の中でホスト名を表す変数を利用できるようにすればよいが、他の pointcut 指定子の中では変数を利用できないので、設計の整合性をとるため、我々はこの方針を採用しなかった。代わりに次に説明する組み込み変数を利用する。

**Advice**

DJcutter では、AspectJ 同様、before, after, around の 3 種類の advice が利用可能である。Advice の定義には AspectJ と同じ構文を使う。

DJcutter の advice 内では、AspectJ でも利用できる `this` や `target` の他に、組み込み変数 `$remotehost` を利用できる。この変数は join point が存在するホストの名前を表す `String` 型の変数である。Join point が存在するホストに応じて advice が実行する処理の内容を変える場合は、この変数を使って処理を切り替える。

**アスペクト・フィールドとメソッド**

AspectJ 同様、DJcutter はアスペクト内にフィールドやメソッドを定義し、advice からそれらを利用することが可能である。アスペクト内で宣言したフィールドやメソッドを普通のクラスから呼び出したい場合、ユーザはアスペクトが実装している interface を介して呼び出す。例えば、アスペクトの定義が以下のようであるとする。

```
aspect LoggingAspect implements Logger {
  void displayLog(Point p, int x) {
    System.out.println("set x: " + x);
  }
}
interface Logger extends AspectInterface{
  void displayLog(Point p, int x);
}
```

`LoggingAspect` アスペクト内部で定義されている `displayLog` メソッドを呼び出しする場合は、`LoggingAspect` アスペクトが実装している Logger インタフェースを使って次のようにする。

```
Logger logger = (Logger)
  Aspect.get("LoggingAspect", this);
logger.displayLog();
```

`Aspect` は、DJcutter が用意するクラスライブラリが提供するクラスである。このクラスの `static` メソッド である `get` を利用すると、目的のアスペクトへの遠隔参照 (プロキシ・オブジェクト) が得られる。この遠隔参照を利用すると、アスペクトに定義されたメソッドを遠隔呼び出しすることができる。遠隔呼び出しの引数は、pointcut 引数と同様に、指定すると遠隔参照に変換されて渡される。

### 3.3 実　　装

DJcutter は大まかにアスペクトのソースファイルをコンパイルする処理系と、分散プログラムを実行する実行時ライブラリから構成されている。アスペクトはコンパイルされると、通常の Java のクラスに変換されてコンパイルされる。このとき、アスペクトのメンバであるアスペクト・フィールドやメソッドはそのクラスのフィールドやメソッドへ、また advice は、`static` メソッドへと変換される。

**アスペクト・サーバ**

プログラムの実行時に最初に起動されるのは、アスペクト・サーバである。このサーバはアスペクトを変換して得られたクラスをロードし、advice やメソッドの実行に備える。また各 advice に対応する pointcut の定義もアスペクト・サーバに読みこまれ、他のホストからの参照に答える準備をおこなう。

**遠隔クラス内の join point の抽出**

各ホストの上では、クラスが DJcutter 専用のクラスローダ[8] によって JVM に読み込まれる。このクラスローダは、アスペクト・サーバに問い合わせをおこな

| pointcut 指定子 | 説明 |
|---|---|
| within(*Type*) | それぞれののホスト上に存在する *Type* クラス内の全ての join point を抽出 |
| target(*Id*) | *Id* と同じ型のオブジェクトを実行している全てのホスト上の join point を抽出 |
| args(*Id*, ...) | *Ids* と同じ型を引数にとる join point を全て抽出 |
| call(*Signature*) | *Signature* に一致しているメソッドの呼び出し join point を抽出 |
| execution(*Signature*) | *Signature* に一致しているメソッドの実行 join point を抽出 |

**図 3** DJcutter の pointcut 指定子

い、現在ロードしようとしてるクラス内の join point が、いずれかの advice の pointcut で抽出されているか検査する。もし抽出されているときは、bytecode レベルの変換により、アスペクト・サーバ内の advice を呼び出すコードをその join point に埋め込む。本システムの bytecode 変換には、Javassist[7] を利用している。Advice は、通常の Java のクラスの `static` メソッドに変換されるので、これを遠隔呼び出しするコードが bytecode 変換により join point に埋め込まれる。

**遠隔参照の実装**

DJcutter ではオブジェクトの遠隔参照を利用して、任意のホスト間で遠隔メソッド呼び出しをおこなうことができる。オブジェクトの遠隔参照は、*remote proxy* パターン[13] としても知られるプロキシ・マスタ方式を用いて実装している。プロキシ・オブジェクトは、自分に対するメソッド呼び出しを全て、遠隔ホスト上のマスタ・オブジェクトに中継するオブジェクトである。これにより、遠隔ホスト上のオブジェクトのメソッドを、同一ホスト上のオブジェクトのメソッド呼び出しと同様に、透過的におこなうことができる。

DJcutter が、オブジェクトへの参照を遠隔参照に変換するのは以下の場合である。

- Pointcut によって抽出された join point に制御の流れが到達したとき。Pointcut 引数がオブジェクトへの参照であれば、そのオブジェクトへの遠隔参照に変換され、advice に渡される。
- `Aspect.get` メソッドによってアスペクトの参照を入手したとき。アスペクトへの遠隔参照に変換され、get の戻り値となる。
- アスペクトのメンバであるメソッドを引数つきで呼んだとき、引数がオブジェクトへの参照であれば、そのオブジェクトへの遠隔参照に変換され、呼び出されたメソッドに渡される。
- 遠隔参照がさすオブジェクトのメソッドを遠隔メソッド呼び出ししたとき。遠隔メソッド呼び出しの引数がオブジェクトへの参照であれば、そのオブジェクトへの遠隔参照に変換され、呼び出されたメソッドに渡される。遠隔参照があれば、任意のホスト間で遠隔メソッド呼び出しが可能である。アドバイス・サーバと通常の Java オブジェクトを実行しているホストの間だけでなく、後者のホスト同士の間でも、遠隔メソッド呼び出しは可能である。

上記の場合であっても、引数を遠隔参照に変換するか、引数のオブジェクトをコピー渡しするかを、クラスごとにユーザが指定することができる。例えば String クラスであれば、遠隔参照に変換するよりも、オブジェクトのコピーを渡した方がよい。DJcutter は、特に指定しなければ引数のオブジェクトをコピー渡しする。引数を遠隔参照に変換して渡したいときは、次のような記述を含んだ XML のコンフィギュレーション・ファイルを用意する。

```
<class name="Point" proxy="replace">
```

この例は Point クラスの引数を遠隔参照に変換して渡すことを意味する。`Replace` とは、遠隔参照の実装方法である。我々は過去に、自動分散化ツール Addistant[9] のために 3 種類の遠隔参照の実装方法 replace, subclass, rename を開発した。replace は、一般的なプロキシ・オブジェクトを使って遠隔参照を実現する方法である。現在のところ、DJcutter のユーザは replace しか選択できないが、将来的には他の実装方法も選択できるように DJcutter を拡張する予定である。

## 4. アプリケーション例 - 分散化した observer パターン

2 章で説明した分散化した observer パターンを、DJcutter のアスペクトを用いてモジュール化する例を示す。DJcutter は AspectJ の構文を踏襲しているので、DJcutter で記述したアスペクトは AspectJ で記述したものと、ほぼ同等である。違いは DJcutter ではオブジェクトの遠隔参照を考慮しなければならない点だけである。**図 4** に DJcutter で記述したアスペクトの概要を示す。

Pointcut や advice の定義は AspectJ と同じであ

```
aspect CoodinationAspect
         implements CoodinateProtocol {
  static Hashtable observers;
  static Hashtable subjects;
  void addObserver(String id, Observer ob) {
    // id をキーに ob をフィールド
    // observers に登録
  }
  void addSubject(String id, Subject s) {
    // id が一致する observers と subject
    // の組を s をキーにフィールド subjects
    // に登録
  }
  ArrayList getObserver(Subject s) {
    // フィールド subjects を調べて
    // s に対応する observers を返す
  }

  pointcut subjectChange(Subject s):
    (call(void Point.setX(int))
    || call(void Point.setY(int))
    || call(void Line.setP1(Point))
    || call(void Line.setP2(Point)))
    && target(s);

  after(Subject s): subjectChange(s) {
    Iterator iter
      = getObserver(s).iterator();
    while (iter.hasNext()) {
      // 各 observer に subject が状態を変
      // 化したことを通知する。
    }
  }
}
```

**図 4** DJcutter で記述した observer パターン

る。pointcut の定義で、Point や Line クラスのメソッドの呼び出しを抽出しているが、DJcutter の場合、全てのホスト上の該当する呼び出しがこの定義で抽出される。また advice が受け取る pointcut 引数 s は join point のホスト上の subject への遠隔参照となる。

一方、対応する subject と observers の関係の登録には工夫が必要である。それぞれが異なるホスト上に存在する可能性があるので、登録する際にはそれぞれの遠隔参照を入手する必要がある。DJcutter では、Java RMI のレジストリ・サーバに相当する役割をアスペクトにおわせる。まず、observer を登録するには、observer が存在するホストから、次のようにアスペクトの addObserver メソッドを遠隔で呼び出す。

```
Screen s = ... ;
CoodinateProtocol cp = (CoodinateProtocol)
    Aspect.get("CoodinationAspect", s);
cp.addObserver("editor", (Subject)s);
```

この例では Screen オブジェクトを observer として登録している。addObserver メソッドの第 1 引数 "editor" は対応する subject を識別するためのキーである。第 2 引数 s は、オブジェクトへの参照なので、遠隔参照に変換されてから addObserver へ渡される。

次に subject を登録する。具体的には、次のようにアスペクトの addSubject メソッドを呼び出す。

```
Point p = ... ;
CoodinateProtocol cp = (CoodinateProtocol)
    Aspect.get("CoodinationAspect", p);
cp.addSubject("editor", (Observer)p);
```

addSubject メソッド呼び出されると、第 1 引数 "editor" を使って、登録ずみの対応する observers への遠隔参照を取得する。第 2 引数 p は、オブジェクトへの参照なので、遠隔参照に変換されてから addSubject へ渡されている。したがって、subject と observers の遠隔参照が得られるので、その組を登録できる。登録された組は advice によって利用され、subject の状態変化を observers に通知するために使われる。

このように DJcutter では、アスペクトをレジストリ・サーバの代わりに使って遠隔参照の管理ができる。上の例では簡単のため、Point や Line オブジェクトの複製を各ホストに配置する処理は省略した。しかし DJcutter では、遠隔参照が取得できれば、任意のホスト間で遠隔メソッド呼び出しができるので、そのような複製の作成も、DJcutter の機能を使えば実現可能である。

## 5. 実　験

DJcutter では、advice が join point とは異なるホスト上の JVM で実行される。我々はこのオーバーヘッドを実験で測定した。実験に用いた計算機は、Sun Blade 1000 (Ultra SPARC III 750MHz x 2 CPU, 1 GB メモリ) と Sun Enterprise 450 (Ultra SPARC II 300MHz x 4 CPU, 1GB メモリ) である。Java 言語の処理系には Sun JDK1.4.0 を用いた。2 つのホスト間をつなぐネットワークは 1000baseFX である。

実験では、空 (null) の advice を join point に weave した。advice を実行する JVM が、join point が存在する JVM と同一ホスト上にある場合と、異なるホストにある場合とで、advice の実行時間の計測をおこなった。この実行時間にはネットワーク通信の時間が含まれる。advice を実行するプロセスは常に Blade 1000 上に、join point が存在する JVM は Blade 1000 または Enterprise 450 上に配置した。また比較のため、Java RMI を使って異なる JVM 上の null メソッドを呼び出した場合の実行時間も計測した。JavaRMI の計測では、引数なしの場合と Object 型の引数 1 個の場合を計測した。

| | 同一ホスト | 異なるホスト |
|---|---|---|
| DJcutter | 1.3 | 2.3 |
| Java RMI (0 引数) | 0.7 | 1.4 |
| Java RMI (1 引数) | 1.3 | 2.3 |

**表 1** null advice/method の実行時間 (msec.)

**表 1** に測定結果を示す。この測定結果は、1000 回の繰り返しの平均値である。DJcutter の実装では、advice の 1 回の実行ごとに遠隔メソッド呼び出し 1 回分相当の時間がかかることがわかる。DJcutter は pointcut 引数が使われていないときも、join point が存在する JVM の情報等を advice へ送るので、引数ありの場合の遠隔メソッド呼び出しと同じ程度の時間がかかっている。引数の直列化が必要な分、引数なしの場合より余分に時間がかかっている。

DJcutter は全ての advice を独立した JVM で動かすので、実行には必ず JVM 間通信をともなう。異なるホストにまたがらない横断的関心事の場合は、AspectJ の実装のように同一 JVM で advice を動かせば、大幅な高速化が可能である。我々の実験では AspectJ での null advice の実行は 10 nsec. 程度である。Advice を同一 JVM で実行するか、独立した JVM で実行するか、アスペクトごとに指定できるように DJcutter を拡張することは、我々の今後の課題である。

## 6. 関 連 研 究

本研究以外にも、分散プログラム内の横断的関心事をモジュール化する試みは以前から行なわれている。例えば Soares[3] らは、JavaRMI を用いて書かれた分散プログラムを、AspectJ を使ってモジュール性が高まるように書き直す研究をおこなった。D 言語[11] は DJcutter と同様、アスペクト指向の分散プログラミングを支援する処理系であり、並列に動作するスレッド間の協調動作や遠隔メソッド呼び出しを独自のアスペクトを利用してモジュール化することができる。我々の過去の研究である Addistant[9] では、オブジェクトの分散配置と遠隔参照の実装方法を独立したモジュールに記述することができる。これらの研究は、分散処理に関係する、いわゆる非機能的関心事 (non-functional concern) を、アスペクト指向技術を使って独立したモジュールで記述できるようにしようとするものである。DJcutter のように、分散している join point をひとつのアスペクトの中で取り扱えるようにしようとした研究は、我々の知る限りおこなわれていない。

## 7. ま と め

本発表では、分散プログラムのためのアスペクト指向言語 DJcutter を提案した。DJcutter は、汎用的な AOP 言語として知られる AspectJ のサブセットを拡張し、遠隔ホスト上の join point も抽出できるようにした言語である。Join point が遠隔ホスト上にある場合、join point の状態を表す pointcut 変数は、遠隔参照を可能にするためのプロキシに必要に応じて変換されて advice に渡される。これらの機能により、例えば、AspectJ ではモジュール化が困難である分散化した observer パターンのプログラムをうまくモジュール化することができる。また実験により、advice を遠隔ホスト上の join point と weave した場合の advice の実行オーバーヘッドが、遠隔手続き呼び出しのオーバーヘッドとほぼ同等であることを確かめた。現在の DJcutter の実装では、join point が同一ホスト上にある場合も advice は常に別 JVM で実行される。そのような場合は同一 JVM 内で実行するようにして性能を改善することは今後の課題である。

**謝辞** 本稿を作成するにあたり、我々は科学技術振興事業団の皆様に多大な援助を受けました。謹んで感謝の意を表します。

## 参 考 文 献

1) Kiczales, G., Lamping, J., Mendhekar, A., Cristina V., Lopes, J.-M.L. and Irwin, J.: Aspect-Oriented Programming, *ECOOP '97 – Object Oriented Programming* LNCS, 1241, Springer, pp. 220–242 (1997).
2) Hannemann, J. and Kiczales, G.: Design Pattern Implementation in Java and AspectJ, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications* ACM SIGPLAN Notices, Vol.37, pp. 161-173 (2002).
3) Soares, S., Laureano, E., and Borba, P.: Implementing Distribution and Persistence As-

pects with AspectJ, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications* ACM SIGPLAN Notices, Vol.37, pp. 174-190 (2003).

4) Clarke, S., Harrison, W., Ossher, H. and Tarr, P. Subject-oriented design: Towards improved alignment of requirements, design, and code. *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, (Oct. 1999).
5) Harrison, W. and Ossher, H.: Subject-oriented programming (a critique of pure objects). *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, (Sept. 1993).
6) Kevin, S. and Lin, G.: Non-modularity in aspect-oriented languages: integration as a crosscutting concern for Aspectj, *Proc. ACM Conf. on Aspect-Oriented software development*, (2002).
7) Chiba, S.: Load-time Structural Reflection in Java, *ECOOP 2000 – Object Oriented Programming* LNCS, 1850, Springer, pp. 313-336 (2000).
8) Liang, S. and Bracha, G.: Dynamic Class Loading in the Java Virtual Machine, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications* ACM SIGPLAN Notices, Vol.33, No.10, pp. 36-44 (1998).
9) Tatsubori, M., Sasaki, T., Chiba, S. and Itano, K.: A Bytecode Translator for Distributed Execution of "Legacy" Java Software, *ECOOP 2001 – Object-Oriented Programming* LNCS, 2071, Springer, pp. 236-255 (2001).
10) Tilevich, E. and Smaragdakis, Y.: J-Orchestra: Automatic Java Application Partitioning, *ECOOP 2001 – Object-Oriented Programming* LNCS, 2071, Springer, (2002).
11) Lopes, C, V. and Kiczales, G.: D: A Language Framework for Distributed Programming, Technical Report SPL97-010, Xerox Palo Alto Research Center, Palo Alto, CA, USA (1997).
12) Sun Microsystems: The Java Remote Method Invocation Specification (1997). http://java.sun.com/products/jdk/rmi/.
13) Rohnert, H.: The Proxy Design Pattern Revisited, *Pattern Languages of Program Design 2*, chapter 7, pp. 105-118, Book published by Addison-Wesley (1995).
14) Gamma E., Helm R., et al., *Design patterns – elements of reusable object-oriented software.* Book published by Addison-Wesley. (1994).