社団法人 情報処理学会　研究報告
IPSJ SIG Technical Report

2004－SE－144　(28)
2004／3／19

連想アスペクト

櫻井 孝平[1]　増原 英彦[2]　鵜林 尚靖[3]

松浦 佐江子[1]　古宮 誠一[4]

[1] 芝浦工業大学　[2] 東京大学大学院　[3] 九州工業大学　[4] 芝浦工業大学大学院

AspectJ のような言語に対して, **連想アスペクト** *(association aspects)* と呼ぶ言語の機構を提案する. 連想アスペクトは AspectJ のオブジェクトごとのアスペクトを拡張して, オブジェクトのグループにアスペクトのインスタンスを関連づけることを可能にする. またアドバイスの実行文脈としてアスペクトのインスタンスを特定するための, 原始ポイントカットを新たに提供する. 連想アスペクトはオブジェクトの特定のグループに関連づけることができるため, 状態による振る舞いの横断的関心事を直接実装することが可能になる. 新しい原始ポイントカットは既存の暗黙的な機構と比べ, より柔軟なアスペクトインスタンスの特定を可能にする. そして, 連想アスペクトと標準の AspectJ のアスペクトによる実行時間の比較を行い, 中規模の利用の場合, それらの実行時間はほぼ同等であることを示す.

Association Aspects

Kouhei Sakurai[1]　Hidehiko Masuhara[2]　Naoyasu Ubayashi[3]

Saeko Matsuura[1]　Seiichi Komiya[1]

[1] Shibaura Institute of Technology　[2] University of Tokyo　[3] Kyushu Institute of Technology

We propose a linguistic mechanism for AspectJ-like languages that concisely associates aspect instances to object groups. The mechanism, which supports *association aspects*, extends the per-object aspects in AspectJ by allowing an aspect instance to be associated to a group of objects, and by providing a new pointcut primitive to specify aspect instances as execution contexts of advice. With association aspects, we can straightforwardly implement crosscutting concerns that have stateful behavior related to a particular group of objects. The new pointcut primitive can more flexibly specify aspect instances when compared against previous implicit mechanisms. The comparison of execution times between the programs with association aspects and the ones with regular AspectJ aspects revealed that the association aspects exhibited almost equivalent for the medium-sized configurations.

1　はじめに

アスペクト指向プログラミング (AOP) では, アスペクトは横断的関心事のモジュール単位である. アスペクトは独自のモジュールシステムとして提供される (例えば AspectJ[10]) か, もしくは既存のモジュールシステムを利用することによって定義される (例えば Hyper/J[16]). 両方の場合において, アスペクトは状態と振る舞いのカプセル化として提供される. そして AspectJ のような言語では, インスタンス変数やアドバイス宣言によりカプセル化が表現される.

AspectJ のような言語はアスペクトインスタンスの**文脈で**アドバイス本体を実行するが, 通常, アスペクトインスタンスはプログラムの実行中に明確でない. このため, どのようにアドバイスの実行文脈であるアスペクトインスタンスを決定するかが問題となる. 例えば AspectJ では, この問題に対していくつかの機構を提供している.[1]

- **シングルトン** *(singleton)* アスペクトはその宣言ごとに,1 つのアスペクトインスタンスのみが作られる. この種のアスペクトは, システム全体に影響する振る舞いの関心事を実装する場合に適している.
- **オブジェクトごと** *(per-object)* のアスペクトは 1 つのアスペクトインスタンスを, それぞれオブジェクトごとに**関連づける**. アドバイス実行を引き起こすオブジェクトの処理により, システムは自動的にオブジェクトに関連づけられたアスペクトインスタンスを探し出し, 実行文脈として使用する. この種のアスペクトは, それぞれのオブジェクトごとの状態を持つ関心事の実装に適している.

これらは特定の横断的関心事に対しては有効であるが,Sullivan らが指摘した**振る舞いの関連**を直接実装することはできない. 振る舞いの関連とは, それぞれの振る舞いの拡張もしくは修正により, オブジェクトの集合を統合する関心事である.[21] 上記で示した既存の機構では, このような振る舞いの関連は通常, シングルトンアスペクトを作ることで実装される. そしてそのアスペクトは, 状態を関連づけた表を持つように定義される. その表の値はオブジェクトのグループに対して一意に決まる. 結果としてこの実装は, 単一のアスペクト定義の中に, 中心となる振る舞いのためのコードだけでなく, 関連を管理するコードが必要になる.

Rajan と Sullivan はこの問題の解決として, アスペクトインスタンスによるインスタンスレベルのアドバイスを提案した. また, 実装として AOP 言語である Eos[19] を提供した. Eos では, 振る舞いの関連の表現として, アスペクトインスタンスをプログラマが直接作ることができる. それぞれのアスペクトは, 表現する関係中のオブジェクトに対して関連づけることができる. プログラムの実行中に呼ばれたメソッドに対するアドバイスの実行では, 呼び出しの対象に関連づけられたアスペクトインスタンスが実行の文脈となる. 結果として, この機構は振る舞いの関連を自然に実装できる. しかしながら, この機構は次のような問題があり, 改善の余地がある.(1) アスペクトインスタンスの選択において, 常に対象のオブジェクトが選択されてしまい, 柔軟でない.(2) 関連づけられた複数のオブジェクトの型の間で, 互換性がある場合, それらのオブジェクトを区別するためには, 言語に対する追加構造が必要になる.

この論文は**連想アスペクト** *(association aspects)* と呼ぶ別の機構を提供する. 連想アスペクトはオブジェクトのグループに対して, アスペクトのインスタンスを関連づけることを可能

[1] 制御の流れに基づいた機構もあるが, この論文の議論とは直接関係ない.

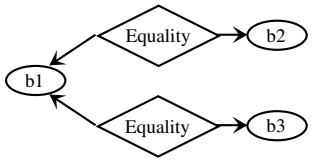

図 1: Bit の統合

にする．この機構は, 新しい原始ポイントカットを提供することで, 柔軟なアスペクトインスタンスの選択ができ, 関連づけた複数のオブジェクトを区別するをことを可能にすることで, 上記の問題を解決する．また連想アスペクトに対し, 手動で表を管理するシングルトンアスペクト, 及びオブジェクトごとのアスペクトとの比較を行った. その結果として, 連想アスペクトが許容範囲のオーバーヘッドで実装可能であることを示す.

この論文は次のような構成になっている． 2 節では振る舞いの関連の例を示す． 3 節では, 連想アスペクトのデザインと提供する機構を説明する． 4 節は, 純粋な AspectJ のプログラムと連想アスペクトとの比較による性能評価を提供する． 5 節は連想アスペクトと類似の手法との比較を行う． 6 節は結論である.

2 動機となる例

この節では連想アスペクトが必要となるシステムの例を提供する． 2.1 節はシステム統合の問題を提供し, それがオブジェクト指向のプログラミングでは横断的関心事になることを説明する． 2.2 節では, そのような関心事を AspectJ で無理矢理実装した場合の例を示す． 2.3 節では, そのような問題が発生する場合に対する解析を述べる.

この問題は最初に Sullivan, Gu, Cai[21] により指摘されたものである. 彼らの研究をよく知っているならば 3 節にスキップできる.

2.1 システム統合

横断的関心事として, 独立して開発されたシステムの統合が挙げられることがある. システムを統合するために, システムの多くの表現を修正しなければならないためである．例えば, 統合開発環境システム (IDE) を構築する場合を考える. これはテキストエディタやコンパイラシステムの統合になる.[20, 22] AOP を適用しない場合, 統合関心事の表現がサブシステムそれぞれに出現する必要がある. 例えば,“save” メソッドはファイルに保存するだけでなく, コンパイラも起動する必要がある.

具体化のために, この論文では Bit オブジェクトを考える.Bit オブジェクトは boolean のインスタンス変数と, 変数値の設定, クリア, 取得のメソッドを持つ.

```
class Bit {
    boolean value = false;
    void set()    { value = true; }
    void clear()  { value = false; }
    boolean get() { return value; }
}
```

この統合は, 特定の Bit ごとのペアの値を同期させるための関連により表現される．関連はその種類 (等価関連またはトリガ関連) と Bit オブジェクトのペアからなる．この関連はプログラムの実行中に, 動的に作られる.

図 1 は 3 つの Bit オブジェクト (図中の丸) と, それらを接続する 2 つの等価関連 (図中の菱形) を表している. 1 つの等価関連は set と get の呼び出しを, 左手側から右手側, もしくはその逆に伝播させる. つまり,set が b_2 に呼ばれたとき, 上の等価関連が b_1 の set を呼び出す. そしてそれに伴って, 下の等価関連が b_3 に対して set を呼ぶ. また無限ループを防ぐ必要がある. 例えば,b_1 が上の等価関連から呼び出されたとき,b_2 には戻って伝播させない.

トリガ関連は, 単に左手側から右手側に呼び出しの伝播を行うものである.

ここで上のシステム統合が, 横断的関心事であるとして議論を続ける. しかし伝統的なオブジェクト指向の実装が 1 つ考えられる. 興味のある読者は Sullivan らの論文 [21] に詳細な議論をみることができる.

2.2 AspectJ による解決

AspectJ ではアスペクトを定義することで, 上で述べた関連を実装できるが, その定義は直感的なものではない.

図 2 は,AspectJ で可能な等価関連の定義を表したものである.[2] それぞれの状態を表現するために Relation と呼ぶ内部クラスを定義している. その内部クラスは, 関連する Bit オブジェクトの参照と busy フラグを持っている．このアスペクトはそれぞれの Bit オブジェクトに Relation のリストを追加し, アドバイスが Bit オブジェクトから Relation を取得できるようにしている.

2 つのアドバイス宣言は, すべての Bit オブジェクトに対する set と clear の呼び出しをとらえるものである．アドバイスの本体では, メソッド呼び出しの対象のオブジェクトから relations のリストを参照する．そのリスト中のそれぞれの Relation について, フラグをチェックし, 同じ名前のメソッドを呼び出す. 同じ Relation において, アドバイスが再帰的でないときにメソッド実行が行われる.

静的なメソッドである associate は関連を作り出す. 2 つの Bit オブジェクトと共にそのメソッドが呼ばれると,Relation オブジェクトを作り出し, 与えられた Bit オブジェクトのそれぞれの relations リストに登録を行う．以下のコード行の実行により, 図 1 で示される Bit の統合システムが構築される.

```
Bit b1 = new Bit(), b2 = new Bit(), b3 = new Bit();
Equality.associate(b1,b2); //b1 と b2 を接続する
Equality.associate(b1,b3); //b1 と b3 を接続する
```

2.3 AspectJ による解決での問題点

図 2 は,AspectJ による Equality アスペクトを表している. これは直接的な等価関連のモデルにはなっていない.

デザインレベルでは, 等価関連は, 状態 (関連するオブジェクトと busy フラグ) と振る舞い (メソッド呼び出しの解析と, 伝播) をカプセル化する実体である．プログラミングレベルでは, インスタンスにより関連がモデル化されるのが自然である．しかし, ここで示した解決手法は, 関連がアスペクトの宣言 (振る舞い) と内部クラスのインスタンス (状態) としてモデル化されている.

[2] この定義は Sullivan らによって提供されたオリジナルのアウトラインに従って, 著者らが書いたものである.

```
aspect Equality {
  static class Relation {
    Bit left, right;
    boolean busy = false;
    Bit getOpponent(Bit b) {
      return b==left ? right : left;
  } }
  private List Bit.relations = new LinkedList();

  static void associate(Bit left, Bit right) {
    Relation r = new Relation();
    r.left = left;
    r.right = right;
    left.relations.add(r);
    right.relations.add(r);
  }
  after(Bit b): call(void Bit.set())
                && target(b) {
    for (Iterator iter=left.relations.iterator();
         iter.hasNext(); ) {
      Relation r = (Relation) iter.next();
      if (!r.busy) {            //無限ループを
        r.busy = true;          //回避する
        r.getOpponent(b).set();
        r.busy = false;
      }
    }
  }

  //clear メソッドのアドバイスが以下に続く
  // ...
}
```

図 2: AspectJ での等価関連の実装

加えて, この解決手法では, 関連づけたオブジェクトを探し出すために, 状態のリストを管理しなければならない. これは冗長であり, 関連を追加するという実際の作業からプログラマの注意を紛らわせるものである.

通常, オブジェクトのグループに影響し, 状態による振る舞いを持つ関心事を直接, 実装することができないため,AspecctJ でのアスペクトのインスタンス化機構は十分ではない. アスペクトインスタンスとして状態と振る舞いをカプセル化する, という考えは自然な発想であり, オブジェクトのグループごとにアスペクトインスタンスが構築可能な機構は有用であると言える.

AspectJ での**シングルトン**アスペクトはインスタンスを 1 つ以上作れないので, 結果として, 実装は異なったオブジェクトの中に状態を割り当てるということになり, 表を使ってそれらのオブジェクトを管理することになる.

AspectJ での**オブジェクトごと**のアスペクトは `pertarget` と `perthis` アスペクトと名付けられている. この種のアスペクトは, それぞれのオブジェクトごとに 1 つのアスペクトインスタンスのみが作られるため, 振る舞いの関連の実装には十分ではない. オブジェクト間の関連を表現するためには,1 つのオブジェクトに対して,1 つ以上のアスペクトインスタンスが存在できなければならない.

この問題は大規模のシステム統合特有の問題でなく, より小規模のシステムにおいてもみられる問題である, と考えられる. 例えば,Hannemann, Kiczales[6] による AspectJ の GoF デザインパターン [5] の実装がある. この実装では,23 パターンのうち 6 パターンが状態間の関連を表を使って管理している.

```
aspect Equality perobjects(Bit, Bit) {
  Bit left, right;
  Equality(Bit l, Bit r) {
    associate(l, r);        //関連づけを
    left = l; right = r;    //構築する
  }
  after(Bit l) : call(void Bit.set())
      && target(l) && associated(l,*){
    propagateSet(right);    //左が呼ばれたら,
  }                         //右をセットする
  after(Bit r) : call(void Bit.set())
      && target(r) && associated(*,r){
    propagateSet(left);     //右が呼ばれたら,
  }                         //左をセットする
  boolean busy = false;     //関連がアクティブで
                            //あるかどうかを示す
  void propagateSet(Bit opp) {
    if (!busy) {            //既に伝播されて
      busy = true;          //いなければ opp に
      opp.set();            //対して set を呼ぶ
      busy = false;
  } }
  // clear メソッドのアドバイスが続く
}
```

図 3: 連想アスペクトによる等価関連

3 連想アスペクト

3.1 概要

AspectJ におけるアスペクトのインスタンス化機構の拡張を提案し, **連想アスペクト**と呼ぶ. 連想アスペクトにより, プログラマはオブジェクトの組にアスペクトのインスタンスを関連づけることができるようになる. 連想アスペクトは, 振る舞いの関連の様な横断的関心事を直接モデル化するためにデザインされており, 特定のオブジェクトのグループ間での振る舞いを表現する.

連想アスペクトは 2 つの基本的な機能をサポートしている. それらは (1) オブジェクトの組にアスペクトのインスタンスを関連づける機能と,(2) アドバイスの実行時に関連づけたアスペクトのインスタンスを選択する機能である.

図 3 は連想アスペクトによる `Bit` の統合の例を示している. 最初の行の `perobjects` 節は,`Bit` オブジェクトの組にインスタンスが関連づけられることを示す. 以下のコードは, 図 1 のアスペクトを利用して, 統合された `Bit` を構築するものである.

```
Bit b1 = new Bit(), b2 = new Bit(), b3 = new Bit();
Equality a1 = new Equality(b1,b2);
Equality a2 = new Equality(b1,b3);
```

`new` 演算子の式は `Equality` アスペクトインスタンスを生成す

る．Equality のコンストラクタは, 与えられた 2 つの Bit オブジェクトに対して, 生成されたインスタンスを関連づける．

アドバイス宣言の中の associated ポイントカットはどのアスペクトインスタンスがアドバイス本体の実行文脈となるかを指定するものである．target(l) && associated(*,l) のポイントカットの組み合わせは, 現在の対象となるオブジェクトに関連づけられたアスペクトインスタンンスを選択する．選択されたアスペクトインスタンスはアドバイスの実行文脈となる．例えば, アドバイスの本体は, 選択されたアスペクトインスタンスのインスタンス変数に対する読み書きを伴って実行する．

例えば,b2.set() が評価された場合, アスペクトインスタンス a1 が 2 番目のアドバイスによって選択される．そして, 選択されたアスペクトインスタンスがアドバイスの本体を実行する．このアドバイスは a1 の flag を確認し,a1 で b1 に割り当てられている left.set() を呼び出す．

ここで, アスペクトのインスタンスを選択し, 選択されたインスタンスをアドバイスの文脈としてアドバイスを実行する処理を, **アスペクトインスタンスに対するアドバイスのディスパッチ**として参照することにする．

以下の節は関連づけとアドバイスのディスパッチの機構を詳しく説明する．

3.2 アスペクトインスタンスの生成と関連づけ

連想アスペクトは perobjects 節により宣言される．perobjects 節は以下の構文からなる．

aspect A perobjects(T,...) { $mdecl$ ... }

A はアスペクトの名前であり,T は関連づけられるオブジェクトの型である．そして mdecl はコンストラクタ, メソッド, 変数, アドバイスをなど含むメンバ宣言である．

連想アスペクトは new A(...) の式の評価によりインスタンス化される．これはオブジェクトと同様である．新しいアスペクトインスタンスは初期化のためにコンストラクタを実行する．新しく作られたアスペクトのインスタンスは, どんなオブジェクトにも関連づけられていない．

perobjects(T_1,T_2,...,T_n) 節は A に associate メソッドを自動的に定義する． そのメソッドは型 $T_1,\ldots,T_n$ の n 個のオブジェクトを引数にとり, 与えられた $o_1,\ldots,o_n$ のオブジェクトにアスペクトインスタンスを関連づける．また void A.delete() メソッドも定義される．このメソッドは関連づけを破棄する．

AspectJ でのオブジェクトごとのアスペクトとは対照的に, アスペクトの生成と関連づけは明示的である．典型的な連想アスペクトの利用方法は Bit 統合の例での Equality 関連のように, 明示的であるためである．連想アスペクトを生成するジョインポイントが特定できるならば, 3.4 節で示すように, この処理を侵入的でなくすることができる．

3.3 アスペクトインスタンスへのディスパッチ

意味的には, アドバイスをアスペクトインスタンスへのディスパッチは, **すべての**アスペクトのインスタンスのコンテキストで同じアドバイスを実行しようとして, **実際には**本体を実行するのはポイントカットにより特定されたインスタンスだけである, ということにより実現することができる．関連づけられたアスペクトのインスタンスを選択するためには,associated 原始ポイントカットが提供される．

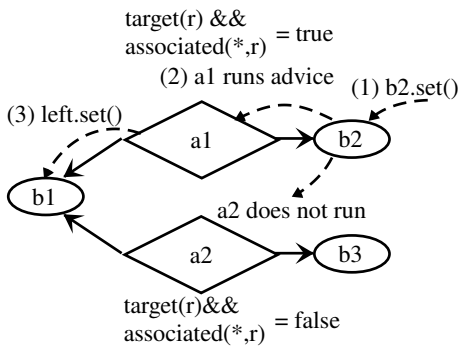

図 4: 関連づけられたアスペクトへのアドバイスのディスパッチ

図 4 はこの節の最初に提示した例での意味を示している．b2.set() の評価は呼び出しのジョインポイントを生成する (1). ここで,2 番目のアドバイス宣言の実行に注目する．それぞれのアスペクトインスタンスはポイントカットをテストする．2 番目の引数の b2 に関連づけられたアスペクトインスタンスが特定され,a1 だけがアドバイスを実行する (2). アドバイスの本体は実行文脈に保持されているインスタンス変数の読み取りにより,a1 に呼び出しを伝達する (3).

複数のアスペクトのインスタンスのアドバイス宣言のテストと実行は, 非決定な順になる．around アドバイスでは, アスペクトインスタンスが proceed 式を実行するときに, 次のアスペクトインスタンスが同じアドバイスのテストと実行を行う．

associated ポイントカットはインスタンスがどのようにオブジェクトに関連づけられたかを決定する． perobjects(T_1,...,T_n) を伴ったアスペクトは,associated(v_1,...,v_n) としてポイントカットが記述される．ここで v_i は,

- 変数であり, 他のポイントカットで束縛されている (例えば target(v_i)). もしくは,
- ワイルドカードとして, アスタリスク (*) で指定される.

追加の制約として,associated ポイントカットはその引数の最低 1 つの変数が束縛されていなければならない．

associated(v_1,...,v_n) ポイントカットは, すべてのアスペクトインスタンス $1 \le i \le n$, v_i が, アスタリスクであるか,v_i が o_i に束縛された変数であり,$\langle o_1,\ldots,o_n\rangle$ に関連づけられていれば, 真と評価される．このアスタリスクは 1 つ以上のアスペクトインスタンスを同一のジョインポイントで適合することを可能にする．

ポイントカットはパラメータの位置を区別する．これは, ある関連の異なった位置における異なったイベントをとらえるような,“方向のある” 関連を定義するのに便利である．

3.3.1 関連づけられたオブジェクトへの束縛

associated ポイントカットは, ワイルドカードを書く替わりに自由変数を記述することで, 関連づけられたオブジェクトを変数に束縛することができる．例えば, 次の宣言は図 3 の最初のアドバイス宣言を少し修正し, ワイルドカードの替わりに自由変数 r を置いている．

```
after(Bit l, Bit r) : call(void Bit.set())
  && target(l) && associated(l,r) {
  propagateSet(r);
}
```

この修正されたアドバイスは,r が本体実行時の 2 番目のパラメータの位置の関連づけられたオブジェクトに束縛されることを除いては, 元のものと同じ振る舞いをする.

束縛の機能により, 相称的な連想アスペクトはより短い定義にできる. 例えば, 以下の単一のアドバイス宣言は, 図 3 の最初の 2 つのアドバイス宣言の代用となる.

```
after(Bit b,Bit o): call(void Bit.set())&&target(b)
  && (associated(b,o) || associated(o,b)) {
  propagateSet(o);
}
```

これは associated ポイントカットが, 対象のオブジェクトに関連づけられた関連づけられたアスペクトインスタンスの特定を, パラメータの位置を気にせずに行うからである.

そして束縛の機能が,o を対象でない関連づけられたオブジェクトに束縛する.

3.4 静的なアドバイス

連想アスペクトは**静的な**アドバイスを宣言することができる. それはシングルトンアスペクトでのアドバイスの宣言と同じような意味を提供する. アドバイスの宣言が static 修飾子をもっている場合, 存在するアスペクトインスタンスの数にはよらず, ポイントカットのマッチと実行が 1 回だけ実行される. 自明なこととして, 静的なアドバイスは associated ポイントカットを使わない. このアドバイスの実行文脈はアスペクトのクラスである. このアドバイスの本体は静的な (クラス) 変数の読み書きのみができる.

典型的な起動のために, 静的なアドバイスは利用される. 最初はアスペクトのインスタンスは存在しないため, 新しいアスペクトインスタンスをアドバイスの機構によりインスタンス化したい場合には, 静的なアドバイス宣言が使われる. 例えば, 以下のコードのアドバイスは Equality インスタンスを callSomeMethod() の発生時に作り出す.

```
aspect Equality perobjects(Bit, Bit) {
  static after(Bit l, Bit r) :
      callSomeMethod() && args(l,r) {
    new Equality(l,r); アスペクトのインスタンスを作り出す
  }
  ...
}
```

3.5 アスペクトインスタンスの検索

アスペクトのインスタンスが特定の組のオブジェクトに関連づけられているかどうか, を確認しなければならない場合がある. または特定のオブジェクトに関連づけられたすべてのアスペクトインスタンスをチェックしたい場合もある (例えば,1 つのオブジェクトからすべてのアスペクトインスタンスを削除する場合). このような操作は,associated ポイントカットを利用したアドバイスの宣言により, 実現できるため, このような目的のために特別な機構は提供しない.

```
aspect Equality perobjects(Bit,Bit) {
 ...
 static void showAll(Bit b) { }    // 空の本体
 after(Bit b) :
    call(void Equality.showAll(Bit))&&args(Bit b)
    && (associated(b,*) || associated(*,b)) {
  System.out.println(this);  //this は関連づけられた
} }                          //インスタンスに束縛される
```

図 5: アスペクトインスタンスを列挙するイディオム

例としては同じオブジェクトのペアに対して,1 つ以上の Equality アスペクトインスタンスを作らないというものが挙げられる. これは次のようなアドバイスの動作として定義される.

```
aspect Equality perobjects(Bit,Bit) {
  ...
  Equality around(Bit l, Bit r) :
      call(new Equality(Bit,Bit)) && args(l,r)
      && (associated(l,r) || associated(r,l)) {
    return this;
} }
```

プログラムが new Equality(b,b') を実行し, アスペクトインスタンス a が $\langle b, b' \rangle$ もしくは $\langle b', b \rangle$ に関連づけられているかどうかを確認する. 既に関連づけられていれば, 上記のアドバイスは新しいオブジェクトではなく a を返す. 関連づけられているアスペクトインスタンスがなければ, アドバイスは実行されず新たな Equality インスタンスが返る.

特定のオブジェクトに関連づけられたすべてのアスペクトインスタンスを列挙することが, アドバイスの宣言を伴った空の静的なメソッドにより実現可能である. 例えば, 図 5 は Equality.showAll(b) の実行により,b が左もしくは右として関連づけられた, すべてのアスペクトインスタンスを表示する.

3.6 実装

連想アスペクトの機構は,AspectJ バージョン 1.0.6 のコンパイラを改変することで実装された.[3]

連想アスペクトのコンパイルによって, 関連づけるオブジェクトの型にはマップが追加される. マップのキーはそのオブジェクトとペアとして関連づけられるオブジェクトであり, 値は連想アスペクトのインスタンスを格納する. 2 つ以上の型に対して関連づけを行う場合は, さらに複雑になる. (詳細は [23] を参照) アドバイスのディスパッチは, マップ上のすべてのキーと値の組による繰り返しに変換される.

4 パフォーマンス評価

小規模の性能評価テストを実施し,(1) 連想アスペクトのプログラム,(2) 手動で, 関連づけられた状態を管理するシングルト

[3]実装は http://www.komiya.ise.shibaura-it.ac.jp/~sakurai/から入手できる. 現在 associated ポイントカットはアドバイス定義の修飾節となっている. これは柔軟さは低いが, 大きな問題にはなっていない.

ンアスペクトのプログラム,(3)AspectJ での, オブジェクトごとのアスペクトのプログラムの実行時の比較を行った.

すべてのベンチマークテストは Sun HotSpot Client Java VM version 1.4.2 beta で,Celeron 800MHz の Windows XP Professional, メモリ 512MB のマシンで実行された. それぞれの実行時間は, 実行時間の平均によって計測され, ループを 1 秒以上実行して `currentTimeMillis` で計測された値である.

4.1 基本操作のパフォーマンス

基本的な操作として, オブジェクトの生成, アスペクトのインスタンス化と関連づけ, アドバイスの実行を伴ったメソッドの実行を定義し, それぞれのコストを計測した. これらは以下の操作のプログラムの実行により計測される.

1. (**OBJ:**) n 個の空のメソッドとインスタンス変数を持つ,n 個のオブジェクトを作る.
2. (**ASSOC:**) アスペクトインスタンスを 1 つ作り n 個のオブジェクトと関連づけ, そして,
3. (**ADV:**) オブジェクトの空のメソッドを実行する.

ADV において, アスペクトは m $(1 \leq m \leq n)$ 個のオブジェクトを引数から取り出し, それらのオブジェクトを使うことで, アスペクトのインスタンスを探し出すアドバイスを実行する. アドバイスの本体は単純に 5 つの整数のインスタンス変数の値を増加させるものである. アスペクトの定義は以下のようになる.

```
aspect Test perobjects(C,...,C) {
   int x1, x2, x3, x4, x5;
   Test(C o1,...,C on) {
      associate(o1,...,on);
   }
   before() :  callEmptyMethod()
         && args(o1,...,om,*,...,*)
         && associated(o1,...,om,*,...,*) {
      x1++; x2++; x3++; x4++; x5++;
} }
```

3 つの実際のアスペクトの実装がある.

AA: (上で示した) 連想アスペクトを利用.

SNG: AspectJ でシングルトンアスペクトを使い, 内部クラスのオブジェクトを `HashMap` に状態として入れた.

PO: AspectJ で (`pertarget` と名づけられている) オブジェクトごとのアスペクトを使ったもの. これは n が 1 のみの場合である.

表 1 は基本的な 3 つの操作を, 異なった n, m の組み合わせで行った場合の実行時間を示している. OBJ は 1 つのオブジェクトの生成の時間を示す. OBJ と ASSOC の時間は m には影響を受けない. 右端の列は SNG に対する AA の実行時間である.

表を見る限り,AA は手動の場合の SNG に比べて, ほぼ 14%のオーバーヘッドである. これらの数値はコンパイルされた AA コードの, 基本的な SNG と等価なコードとして妥当な結果である.

4.2 Performance of Bit Integration

`Bit` の統合の例として,AA と SNG(図 2 と 3 を参照) の実装として, それらの実行時間による比較も行った. 計測プログラムは最初に 100 個の `Bit` オブジェクトを作成し,n の等価関連とトリガ関連をランダムに接続する. そして `set` と `clear` メソッドをランダムに選択されたオブジェクトに対して 1000 回実行した.

	n	m	AA	SNG	PO	AA/SNG
OBJ	1		0.951	0.891	0.901	1.07
	2		1.35	1.33		1.02
	3		1.36	1.32		1.03
ASSOC	1		0.320	2.15	0.373	0.15
	2		4.21	7.42		0.57
	3		11.5	16.1		0.71
ADV	1	1	0.0781	0.144	0.137	0.54
	2	1	0.840	0.831		1.01
	2	2	0.194	0.250		0.78
	3	1	1.81	1.59		1.14
	3	2	0.844	0.941		0.90
	3	3	0.310	0.360		0.86

表 1: 基本操作の実行時間 (μsec.)

n	AA	SNG	$\frac{\text{AA}}{\text{SNG}}$	n	AA	SNG	$\frac{\text{AA}}{\text{SNG}}$
10	1.32	0.760	1.7	60	9.95	13.7	0.7
20	1.98	1.32	1.5	70	16.1	15.6	1.0
30	3.00	2.20	1.4	80	24.5	24.7	1.0
40	4.50	3.31	1.4	90	53.6	43.1	1.2
50	6.31	12.9	0.5	100	356	161	2.2

表 2: n 個の関連による Bit 統合の実行時間 (msec)

この全体の実行時間は, 表 2 と図 6 で示される. この表の右端の列を見ると,AA の実行時間は SNG に対して 0.5 から 2.2 の範囲であり, 関連の比重に依存する. AA と SNG でのコレクションライブラリの使用の違いにより, 違いが出ると推定している. AA はハッシュ表を関連づけの管理のための実装で使用していて, 一方 SNG はリンクリストを使っている. 通常ハッシュ表は中間のサイズの要素に対して最適化されており, より小さな場合と大きな関連づけの場合にはパフォーマンスが低下する.

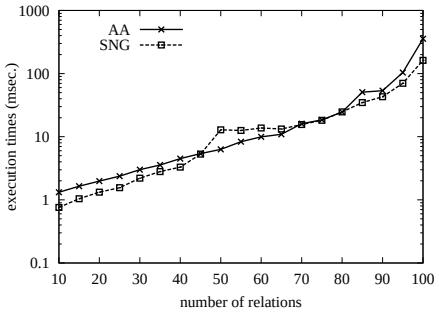

図 6: Bit 統合の実行時間 (mesc)

5 議論

5.1 Eos との比較

連想アスペクトの研究は Eos[19] に基づいているとして, ここで詳細な違いを議論する.

最も顕著な違いは,Eos はアドバイスの実行時にアスペクトインスタンスを選択するとき, 暗黙のうちに現在の対象のオブジェクトを利用するということである. 対照的に, 連想アスペクトは, ポイントカットによって明示的に指定された任意のオブジェクトを使うことができる. この機構が Eos では, 次の場合における柔軟性を低下させている.(1) アスペクトインスタンスが, 対象のオブジェクトでないオブジェクトを利用することで選択される必要がある場合. 例えば, アドバイスがクラスメソッドを呼ぶ場合. (2) そしてアスペクトインスタンスが 1 つ以上のオブジェクトを利用して選択される場合. 例えば, セキュリティの関心事に置いて, オブジェクト A から B に対するメソッドの呼び出しを防ぐ場合は,A と B に関連づけられたアスペクトインスタンスにより実現できる. A から B に対する呼び出しが発生したときに,Eos では B に関連づけられたすべてのアスペクトインスタンスがアドバイスの本体を実行し, そして, おそらく呼び出し元のオブジェクト A がアスペクトインスタンスの選択のために利用されることになる.

連想アスペクトと Eos の両方は, 関連づけられたオブジェクトの役割を区別することができる. しかしながら,Eos は追加的な役割の構造をアドバイスの宣言の周辺に導入することで区別を行う. これはアスペクトの再利用を困難にするだろう. 例えば,`Trigger` と `Equality` アスペクトは, 2.1 節ではアドバイスのディスパッチだけが異なっていて, これらの宣言は Eos では異なったプログラムの構造を持つ必要がある. つまり前者はアドバイスの宣言を, 役割の構造の中に入れなければならない. 一方, 連想アスペクトでは, 役割のオブジェクトを,`associated` ポイントカットの引数の位置よって区別することができる. このようなアドバイスのディスパッチが, ポイントカットによって管理されることで,AspectJ の他の言語機構にうまく適合するだろう. 例えば, ポイントカットの抽象化機構 (すなわち, 名前付きポイントカットや抽象ポイントカット) を通してアスペクトを再利用する場合に利用できる.

最初に目にしたときは, 暗黙のディスパッチ機構が総称的な関連づけに対して, 明示的なものよりも便利に見えるが, 実際には最初に考えたようにはならない. 例えば, 図 3 の 2 つのアドバイス宣言は以下の暗黙の機構で記述できる.

```
after(Bit b): call(void Bit.set())&&target(b) {
  if (!busy) {
    busy = true;
    if (b == left) left.set(); else right.set();
    busy = false;
} }
```

アドバイスはより短いポイントカットになっているが, 本体は対象となるオブジェクトのもう一方に対する呼び出しを伝播させるために, 対象が `left` もしくは `right` であるかの判定を持たなければならない. 連想アスペクトでは単にポイントカットを利用することで, 3.3.1 節で見たように, もう一方を決定することができるので, 結果としてより簡潔なアドバイスの本体になる.

Eos と連想アスペクトの両方とも, オブジェクトに関連づけられたアスペクトのインスタンスが無い場合の, 性能低下について注意を払うべきだろう. `Bit` 統合の例では,`Equality` インスタンスが関連づけられてない `Bit` オブジェクトに対する `set` の呼び出しは, 速度低下を被るべきではない. ここには速度低下に対する 2 つの次元が存在する.

最初はアスペクトインスタンスの数である. 通常の AspectJ による実装 ([19] では first work-around と呼ばれている) では, アスペクトインスタンスのシステム単位でのテーブルに対する検索のために, 性能がかなり低下する. Eos と連想アスペクトでは, この問題をそれぞれのオブジェクトが関連づけられたアスペクトのリストを持つことで回避している.

2 番目は, 呼び出しに対して静的にマッチするアドバイス宣言の数である. 連想アスペクトでは, それぞれのアドバイス宣言が, メソッド呼び出し式に空のリストのループを追加していくため, 性能が線形に低下していくだろう. Eos では, それぞれのメソッド呼び出し式に対するリストを持つことで回避している. しかしながら, この Eos での手法はより多くのメモリと, アスペクトインスタンスの関連づけ/関連づけ削除により多くの操作を要求する. これは, トレードオフの評価のために調査が必要である.

5.2 他の関連研究

アスペクトインスタンスをオブジェクトに関連づけるという, よく似た機構が AspectJ の古いバージョンに存在した.[4]

Mezini と Ostermann は,AOP のモデルのために Caesar を提案した.[13, 14] Caesar では wrapper のインスタンスが,AspectJ でのアスペクトのインスタンスにほぼ相当する. 手動でインスタンス化することで, オブジェクトに関連づけることができる. また,wrapper の再利用の機構は, オブジェクトから関連づけられた wrapper を回復する助けとなる. しかしながら, 著者の理解する限り, それぞれの wrapper1 つのオブジェクトにしか関連づけることができない. 連想アスペクトはこれらの実装と複数のオブジェクトへの関連づけの機能を,AspectJ のアスペクトとして統合している.

PROSE[17, 18] と Handi-Wrap[2],JBoss[4],AspectWerkz[3] などの, いくつかの動的なウィービング機構を持つ AOP のシステムは, アスペクトの定義から複数のアスペクトインスタンスを作り出すことができる. もし, これらの機構がアスペクトインスタンスを複数のオブジェクトに関連づけることができたなら, 振る舞いの関連が直接実装できる. しかし, 著者の理解する限り, これらの機構はアスペクトのインスタンスを実行環境のシステムごとに追加するものである. 例えば, アスペクトインスタンスの追加により, 指定されたクラスのすべてのオブジェクトが影響を受ける. Handi-Wrap は, オブジェクトと wrapper のインスタンス間の関連づけを行うライブラリを定義することはできる.

これらの機構は連想アスペクトが行うことを実装することができる. しかし連想アスペクトが多くのアプリケーションでより宣言的な定義を与えることができると考えられる. 動的なウィービング機構では, 関連づけとアドバイスのディスパッチは, ウィービングの副作用によって理解されるだろう.

6 結論

連想アスペクトを AspectJ での簡潔なアスペクトのインスタンス化機構の拡張として提案した. インスタンス化とポイントカットに基づいたのアドバイスのディスパッチ機構は, 柔軟

[4] この AspectJ の 1.0 以前のバージョンはもはや公開されていない. いくつかの文献 [7, 8] はこのような機能に言及した 0.6 までのバージョンを使っている.

で簡明であり,1 つ以上のオブジェクトに関連づけられたインスタンスを持つアスペクトの表現を可能にする. 結果として, 連想アスペクトは特定のオブジェクトのグループに関する状態を持ち, その状態による振る舞いの関連を持つ横断的関心事の自然な表現を提供する.

連想アスペクトのコンパイラを AspectJ のコンパイラの改良により開発した. 性能評価テストでは, 連想アスペクトを使ったプログラムは正規の AspectJ を使った場合に対して 0.5 から 2.2 倍の速度低下であることを示した. また最適化によりオーバーヘッドを減らすことが可能であると考えられる. 例えば, 関連づけを管理するマップを減らすことで, メモリの性能を向上させることができる.

将来的な研究は, 連想アスペクトを実践的なアプリケーションの横断的関心事に適用することである. 我々は現在, 元のコードを変更しない複数のビューの GUI アプリケーションや,GoF のデザインパターンをテストしている.

連想アスペクトにより適切にモデル化することができるような, デザインレベルの概念を調査することも将来の研究である. UML での “関連オブジェクト” やコラボレーションデザインでの “役割” のような概念において, 多くの可能性があると思われる. 例えば,AspectJ[7] の役割のモデルの実装で, 非本質的なインスタンスを含むものは, 連想アスペクトで実装できる. これは Eos と同様に, 連想アスペクトにおいても, 関連づけられたオブジェクトの区別のために役割の構造をもつような, アスペクトインスタンスのデザインの助けになるだろう.

謝辞

Kevin Sullivan と Hridesh Rajan による,Eos の詳細な情報と論文に対するコメントに大変感謝します. また Tetsuo Tamai,Tomoyuki Kaneko,Etsuya Shibayama, ほか匿名のレビュアーや東京大学の PoPL と Kumiki ミーティングのメンバーによる, 初期の論文のコメントに対しても感謝します.

参考文献

[1] M. Akşit, ed. *Proc. of AOSD'03*. 2003.

[2] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In [9], pp.86–98.

[3] J. Bonér and A. Vasseur. AspectWerkz. `http://aspectwerkz.codehaus.org/`.

[4] B. Burke and A. Brok. Aspect-oriented programming and JBoss. O'Reilly Network, May 2003. `http://www.oreillynet.com/pub/a/onjava/2003/05/28/aop_jboss.html`.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[6] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In [12], pp.161–173.

[7] E. A. Kendall. Role model designs and implementations with aspect-oriented programming. In [15], pp.353–369.

[8] M. Kersten and G. C. Murphy. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. In [15], pp.340–352.

[9] G. Kiczales, ed. *Proc. of AOSD'02*. 2002.

[10] G. Kiczales, et al. An overview of AspectJ. In *ECOOP 2001*, pp.327–353, 2001.

[11] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proc. of Compiler Construction (CC2003)*, pp.46–60, 2003.

[12] S. Matsuoka, ed. *Proc. of OOPSLA2002*. Nov. 2002.

[13] M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In [12], pp.52–67.

[14] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In [1].

[15] L. M. Northrop, ed. *Proc. of OOPSLA'99*, Oct. 1999.

[16] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proc. of the Symposium on Software Architectures and Component Technology*. 2000.

[17] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for Java. In [1], pp.100–109.

[18] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In [9], pp.141–147.

[19] H. Rajan and K. Sullivan. Eos: Instance-level aspects for integrated system design. In *Proc. of ESEC/FSE*, pp.297–306, 2003.

[20] K. Sullivan. Mediators: Easing the design and evolution of integrated systems. PhD Thesis, Dept. of Computer Science, University of Washington, published as TR UW-CSE-94-08-01, 1994.

[21] K. Sullivan, L. Gu, and Y. Cai. Non-modularity in aspect-oriented languages. In [9], pp.19–27, 2002

[22] K. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM TOSEM,*, 1(3):229–268, July 1992.

[23] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura and S. Komiya. Association Aspects. In *Proc. of AOSD'04*. 2004.