活性化のタイミング制御の実装を分離記述可能な分散動的アスペクト指向言語

森田 悟史 † 堀江 倫大 † 千葉 滋 †
Satoshi MORITA Michihiro HORIE Shigeru CHIBA

† 東京工業大学 情報理工学研究科 数理・計算科学専攻
{morita, horie, chiba}@csg.is.titech.ac.jp

分散動的アスペクト指向言語では、アスペクトの活性化のタイミング制御が重要である。しかし、これを考慮してアスペクトを実装すると、このためのコードがアスペクトの中に入り込んでしまうため、タイミング制御用のコードの再利用が困難である。そこで、本論文では新しいアスペクト指向言語を提案する。この言語では、アスペクトの活性化のタイミング制御を別なアスペクトとして定義できる。これにより、タイミング制御の抽象化も可能になり、複雑なタイミング制御処理をライブラリ化することができるようになる。

1 はじめに

プログラムの実行を止めずに振る舞いを変更する技術として、動的アスペクト指向がある。動的アスペクト指向では、アスペクトを動的に織り込んだり取り除いたりすることができる。この技術を分散環境でも使用できるのが分散動的アスペクト指向である。これを用いることで、離れたノードで動いているプログラムに対してアスペクトを動的に織り込むことができる。

動的アスペクト指向は動的に織り込みを行うため、織り込まれたアスペクトが有効になる瞬間である、活性化のタイミングが存在する。分散環境で動的アスペクト指向を用いる場合、この活性化のタイミングを意識しないと、うまくアスペクトを織り込むことができない。しかし、活性化のタイミングを考慮してアスペクトを実装すると、タイミング制御のための実装がアスペクトの中に混ざってしまう。そのため、タイミング制御のコードだけを取り出して再利用することは困難である。

この問題を解決するために、我々は新しい分散動的アスペクト指向言語を提案する。この言語を利用することで、分散動的アスペクト指向におけるアスペクトのタイミング制御を別のアスペクトとして記述することが可能となる。

以下に本稿の残りの構成について説明する。まず、2 章で分散動的アスペクト指向におけるアスペクトの活性化のタイミング制御について述べる。次に、3 章で我々が提案する言語について説明する。4 章で我々が提案する言語を利用した活性化タイミング制御の実装例を示す。そして関連研究について 5 章で述べ、6 章でまとめと今後の課題について述べる。

2 分散動的アスペクト指向の活性化のタイミング制御

分散動的アスペクト指向でアスペクトを織り込む場合、活性化のタイミングを制御することが重要である。織り込まれるとすぐに実行される状態になるアスペクトを動的に織り込む場合、活性化のタイミングは織り込みのタイミングと同じになる。分散動的アスペクト指向では、各ノードに対して瞬間的に織り込みを行うことはできない。これは、ネットワークの速度の違いや、各ノードの CPU 使用率の違いなどが原因となる。このことから、織り込み後すぐに実行される状態になるアスペクトは、織り込みのタイミングのずれにより、活性化のタイミングもずれることがある。このように、分散動的アスペクト指向では、織り込まれたアスペクトをすぐに実行せず、同期をとって活性化することが多い。以下ではそのようなアスペクトの例を示す。

2.1 N 体問題の視覚化アスペクト

図 1 のように、N 体問題を分散環境で計算しているとする。各ノードに質点の座標などのデータを分配し、各ノードは質点間の相互作用を計算し、計算したデータを交換する処理を繰り返す。各ノードは同一ステップの質点の動きを計算している必要があるため、ステップごとにバリア同期を行う。

この N 体問題の計算プログラムに、分散動的アス

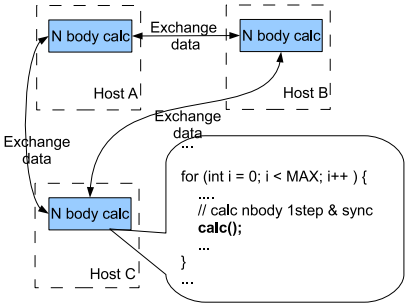

図 1: 分散環境での N 体問題計算

ペクト指向を用いて計算過程の視覚化機能をアスペクトとして追加するとすると、このアスペクトの活性化はタイミング制御が必要である。このアスペクトは、各ノードから各時刻の質点の座標データを受け取り、キャンバスに描画する。このアスペクトを織り込む場合、アスペクトの活性化のタイミングがずれてしまうと、描画開始がずれてしまい、ずれた時刻の質点を描画することになってしまう。

これを避けるためには、N 体問題の計算を行っている各ノードで、ある決まった時刻の計算をしているときに同時に視覚化アスペクトを活性化する必要がある。そのため、計算プログラムが各ステップごとにバリア同期を行っていることを利用し、そのバリア同期と同じタイミングで視覚化アスペクトを活性化するなどの制御が必要である。

2.2 サーバ・クライアント間のメッセージ暗号化アスペクト [6]

次に、サーバ・クライアント間でメッセージのやり取りをするアプリケーションを考える。クライアント側がメッセージを送信すると、サーバ側がそのメッセージを受信する。

このアプリケーションに、分散動的アスペクト指向を用いてメッセージの暗号化機能をアスペクトとして追加するとする。このアスペクトは、クライアント側のメッセージの暗号化と、サーバ側の暗号文の復号化の 2 つのアスペクトから成っている。このアスペクトを織り込むとき、サーバとクライアントで活性化のタイミングがずれてしまい、クライアント側のアスペクトのみ活性化されたとすると、暗号化したメッセージが復号されない不具合が生じてしまう。

この場合は、サーバとクライアントの間のメッセージのやり取りの各セッションごとに、両方のアスペクトが活性化されている、もしくは、両方活性化されていない状態のどちらかになっていないといけない。そのため、クライアントでのメッセージ送信を一時ストップし、サーバで全てのメッセージを受け取り終わった後で双方でアスペクトを活性化し、アプリケーションを再開するなどの制御が必要である。

2.3 活性化のタイミング制御の実装の問題点

上記のように、活性化のタイミングを制御するためには、2 つの機能が必要である。まず、織り込み先の各ノードの処理の進行状態を把握する必要がある。N 体問題の視覚化アスペクトでは、各ノードがバリア同期をしようとしているか、サーバ・クライアント間のメッセージ暗号化アスペクトでは、メッセージのやりとりをしている最中なのかを把握する必要がある。次に、目的のアスペクトが各ノードで準備できているのかを把握する必要がある。N 体問題の例で、バリア同期のタイミングでアスペクトをうまく活性化しようとしても、その時にまだ視覚化アスペクトが準備できていないノードがあるときは活性化してはいけない。また、アスペクトを同時に活性化するための機能も必要である。たとえ上記の機能を利用して活性化すべきタイミングを把握できたとしても、活性化する処理がずれてしまっては意味がない。

素朴な実装で活性化のタイミングを制御しようとすると、各ノードの処理の進行状況の把握と、各ノードの織り込み状況の確認を、織り込みたいアスペクトの実装内に追加のアドバイスを書いて行うことになる。N 体問題の例の視覚化アスペクトを AspectJ [1] で記述すると 図 2 のようになるだろう。しかし、このままでは活性化のタイミング制御ができない。このアスペクトに活性化のタイミング制御処理を加えると図 3 のようになる。まず、新たに after アドバイスを加え、バリア時を実行するメソッド呼び出し Nbody.sync の際に、それを集中管理サーバに通知するようにする。集中管理サーバは、その通知を元に視覚化アスペクトを活性化するか否かを判断し、活性化する場合は各ノードに通知する。通知を受けた各ノードは、視覚化アスペクトによって追加される flag フィールドの値を true に変更し、活性化をおこなう。また、視覚化をおこなう before アドバイスも変更し、flag の値が true のときだけ、視覚化処理をおこなうようにする。

しかし、図 2, 3 を比較すると、活性化のタイミング制御の実装が織り込みたいアスペクトの中に混ざっていることが分かる。また、バリア同期のタイミングを通知する after アドバイスは、視覚化アスペクトを活性化した後では必要がない。このように、必要ないコードが残り続けるという問題もある。しかし図 3 のアスペクトでは、そのように不要となったコードが削除されず残ってしまう。

```
1 /* N 体問題の視覚化 */
2 aspect DrawAspect {
3   before(Satellite s): // 星のデータ
4           execution(void Nbody.calc(Satellite))
5           && args(s) {
6     // 視覚化処理
7 }}
```

図 2: 活性化のタイミング制御しない視覚化アスペクト

```
1  /* N 体問題の視覚化 */
2  aspect DrawAspect {
3    boolean flag = false; // 共有フラグ
4    before(Satellite s): // 星のデータ
5            execution(void Nbody.calc(Satellite))
6            && args(s) {
7      if (flag) {
8        // 視覚化処理
9      }
10   }
11   /* バリア同期のタイミングで実行 */
12   after(): execution(void Nbody.sync()) {
13     // 各遠隔ノードとバリア同期のタイミング
14     // を確認し、なんらかの方法でフラグを
15     // オンにする。
16 }}
```

図 3: 活性化のタイミング制御を行っている視覚化アスペクト

3 活性化のタイミング制御をアスペクトとして分離記述できる言語の提案

前章で指摘した問題点を解決するために、本論文はアスペクトの活性化のタイミング制御の実装を別のアスペクトとして分離記述し、各遠隔ノードに対する織り込みの集中制御を可能にする言語を提案する。

本言語はまず、遠隔ノードの処理の進行状況を把握しやすくするために remote pointcut [4] を提供する。また、あるアスペクトから他のアスペクトを織り込んだり活性化したりする言語機構を備えている。また、目的のアスペクトの織り込み状況を把握することも可能である。これらにより、各遠隔ノードの処理の進行状況をもとに、目的のアスペクトを織り込んで活性化するタイミング制御用アスペクトを容易に記述することが可能となる。

3.1 Remote pointcut

本言語では、 remote pointcut を使用することができる。remote pointcut は、遠隔ノード上で動作しているプログラム内のジョインポイントを指定するための機能である。これを用いることで、遠隔ノード上のプログラムの実行が remote pointcut で指定したジョインポイントに到達したときに、そのジョインポイントが存在するノードとは異なるノードに存在するアドバイスを実行することができる。

remote pointcut を利用することで、各遠隔ノードの処理の進行状況を単一ノード上のアスペクトから集中的に把握することが可能となる。例えば、N 体問題に視覚化アスペクトを織り込む場合を考える。このとき、2.1 章で述べたように、活性化のタイミング制御のアスペクトでは、バリア同期をしているタイミングを把握する必要があった。N 体問題のバリア同期を行っているメソッドを remote pointcut を用いてポイントカットとして選択すると、各ノードでバリア同期に達する毎に、そのアスペクトが存在する特定のノード上でアドバイスが実行される。これにより、単一ノード上のアスペクトによりバリア同期のタイミングを集中管理できる。

本言語が提供する remote pointcut 用のポイントカット指定子を表 1 に示す。これを利用することで、例えば次のようなポイントカットを定義できる。

```
pointcut remote(String h):
            execution(void Nbody.sync())
            && hosts("csg000", "csg001")
            && hostName(h);
```

このポイントカットは、hosts ポイントカットで指定したノード上の Nbody オブジェクトの sync() の実行をジョインポイントとして抽出する。変数 h の値は、実際に sync が実行されたノードの名前となる。このようにポイントカットを定義することで、変数 h の値からどのノードでジョインポイントに達したかをアドバイスで判断して適切な処理を実行することができる。

表 1: 提供する Remote Pointcut

指定子	ジョインポイント
hosts(*hostNames or hostName1, hostName2 ..*)	String 配列、もしくは直接名前で指定したホスト上のジョインポイントを選択
hostName(*hostName*)	ホスト名が *hostName* に一致するジョインポイント

3.2 dynamic アスペクト

本言語では、アスペクトに dynamic 修飾子を付けることで、動的に織り込みが可能なアスペクトを定義することができる。このアスペクトは静的には織り込まれない。このアスペクトはインスタンス化することができ、提供している特別なオペレータを用いることで織り込みや活性化が行われる。このアスペクトを dynamic アスペクトと呼ぶ。

dynamic アスペクトは、他のアスペクトやクラスから織り込んだり、活性化したり、取り除いたりすることができる。このために提供しているオペレータを表 2 に示す。これらのオペレータを用いることで、活性化のタイミング制御用のアスペクトに、目的とするアスペクトを織り込む処理を記述できる。また、提供しているオペレータの中には、織り込み状況の確認のために使う isWoven() や isDeployed() がある。これらを用いることで、アスペクトの織り込み状況を把握することができる。

以下に例を示す。

```
/* dynamic アスペクト */
DrawAspect draw = new DrawAspect();
draw.weave();
while (!draw.isWoven())
  Thread.sleep(10);
draw.deploy();
...
draw.unweave();
```

この例では、DrawAspect アスペクトをインスタンス化し、まず weave() で非活性化状態のまま織り込みを行っている。つまり、 DrawAspect が選択しているジョインポイントでアスペクトを実行する準備をしている。次に isWoven() で準備完了の確認が取れるまで待つ。そしてジョインポイントにアスペクトを実行する準備ができていることが確認できた後で、deploy() でアスペクトを活性化している。最後に、unweave() でアスペクトを取り除いている。このように、タイミング制御用のアスペクトが不要になった後、それを取り除くことができる。

3.3 タイミング制御の実装のライブラリ化

本言語を用いることで、タイミング制御の実装を抽象アスペクトとしてライブラリ化することが可能である。すべての dynamic アスペクトは DAspect クラスを継承している。そのため、dynamic アスペクトをパラメタとしてタイミング制御用のアスペクトに渡すことが可能である。活性化のタイミングを制御するアスペクトを、DAspect 型のパラメタとして dynamic アスペクトを受け取り、受け取った dynamic アスペクトを織り込むように記述することで、再利用可能なライブラリとして実現できる。

例えば、以下のような実装が可能である。

```
abstract aspect BarrierSync {
  private DAspect dAspect;
  public BarrierSync(DAspect dAspect) {
    this.dAspect = dAspect;
    dAspect.weave();
  }
  abstract pointcut remote(String s);
  after(String h): remote(h) {
    ...
    dAspect.deploy();
    ...
  }}
```

この例では、ポイントカットと織り込むアスペクトをパラメタ化することで、タイミング制御の部分だけを再利用可能にすることができている。この抽象アスペクトを継承し、ポイントカットを具体化し、コンストラクタに織り込みたい dynamic アスペクトのインスタンスを渡すことで、タイミング制御用の具象アスペクトを定義することができる。

4 活性化のタイミング制御の実装例

前章では活性化のタイミング制御をアスペクトとして分離記述するための言語を提案した。ここでは、本言語を用いた具体的な活性化のタイミング制御の実装例を示す。

表 2: dynamic アスペクトで利用できるオペレータ

オペレータ	説明
void weave()	ジョインポイントに非活性状態でアドバイスを呼び出す準備をする
void unweave()	アスペクトを取り除く
void deploy()	アスペクトを活性化する
void undeploy()	アスペクトを非活性化する
boolean isWoven()	アスペクトがジョインポイントに準備ができているかどうか調べる
boolean isDeployed()	アスペクトが活性化されているかどうかを調べる

4.1 N 体問題の視覚化アスペクト

まず 2.1 章の視覚化アスペクトを活性化するタイミングを制御するアスペクトを示す。2.1 章で述べた通り、N 体問題計算プログラムに視覚化アスペクトを動的に織り込む場合は、バリア同期をしているタイミングに合わせて活性化する必要がある。そのため、各遠隔ノードがバリア同期をしているタイミングを把握して活性化のタイミングを制御するアスペクトを実装しなければならない。また、すべてのノードでアスペクトが準備できていることが確認できるまで活性化することはできない。これを踏まえて、実装したプログラムを図 4、5、6 に示す。

図 4 に示した BarrierSync 抽象アスペクトで、バリア同期に合わせて目的のアスペクトを活性化するタイミング制御アスペクトを実装している。まず、7 行目のコンストラクタで織り込みたい dynamic アスペクトとバリア同期をしている遠隔ノードのホスト名を取得する。その際、13 行目で目的のアスペクトを非活性状態で織り込みを行っている。BarrierSync にはアドバイスが 2 つ定義されている。1 つはバリア同期を行っているメソッドの直前に実行され、2 つ目はその直後で実行される。この 2 つのアドバイスにより、バリア同期のタイミングに合わせて目的のアスペクトを活性化している。活性化後は、活性化の制御アスペクトは必要ないため、30 行目で、unweave() で取り除いている。BarrierSync 抽象アスペクトを継承して N 体問題の視覚化アスペクト用に具象化したアスペクトを図 5 に示す。

最後に、タイミング制御アスペクトの織り込みを行うためのプログラムを図 6 に示す。これを動かすことで、タイミング制御用アスペクトが織り込まれ、最終的に視覚化アスペクトが織り込まれる。目的のアスペクトである視覚化アスペクト DrawAspect は 4 行目でインスタンス化され DrawPolicy に渡されている。

```
1  public abstract aspect BarrierSync {
2    private String[] nodes;
3    /* 織り込みたいアスペクト */
4    private DAspect dAspect;
5    private Map<String, Boolean> map
6        = new Hashtable<String, Boolean>();
7    public BarrierSync (DAspect dAspect,
8                        String[] nodes) {
9      this.dAspect = dAspect;
10     this.nodes = nodes;
11     for(int i = 0; i < nodes.length; i++)
12       map.put(nodes[i], false);
13     dAspect.weave();
14   }
15   /* バリア同期のメソッドの実行を選択 */
16   abstract pointcut sync();
17
18   before(String host): sync()
19                       && hostName(host) {
20     map.put(host, true); }
21
22   after(String host): sync()
23                       && hostName(host) {
24     synchronized(this) {
25       if(dAspect.isWoven()
26           && !map.containsValue(false)) {
27         if(!dAspect.isDeployed())
28           dAspect.deploy();
29         if(this.isWoven())
30           this.unweave(); }}
31     map.put(host, false); }}
```

図 4: バリア同期を利用したタイミング制御アスペクト

これらの活性化のタイミング制御用アスペクトを用いると、DrawAspect は図 7 のようになる。このアスペクトには活性化タイミングを制御するための実装は含まれていない。

4.2 サーバ・クライアント間のメッセージ暗号化アスペクト

次に 2.2 章のサーバ・クライアント間のメッセージ暗号化アスペクトを記述する例を示す。この例では、

```
1 public dynamic aspect DrawPolicy
2                       extends BarrierSync {
3   private DAspect dAspect;
4   public DrawPolicy(DAspect dAspect) {
5     super(dAspect, dAspect.getHosts());
6     this.dAspect = dAspect; }
7   pointcut sync():
8           execution(void Nbody.sync())
9           && hosts(dAspect.getHosts()); }
```

図 5: 視覚化アスペクト用のタイミング制御アスペクト

```
1 public class Weaver {
2   public static void main(String[] args) {
3     DrawPolicy dp =
4       new DrawPolicy(new DrawAspect());
5     dp.weave();
6     while(!dp.isWoven){Thread.sleep(10); }
7     dp.deploy(); }}
```

図 6: 視覚化アスペクトの織り込み処理

メッセージのやり取りのセッションごとに、サーバとクライアントの両方で暗号化アスペクトを同時に活性化する必要がある。そのため、活性化のタイミング制御用のアスペクトで、メッセージの送信を一時ストップさせ、送信中のメッセージを受信させた後、アスペクトを活性化してアプリケーションを再開させる。この制御を実装すると図 8 のようになる。

図 8 の ServerClient 抽象アスペクトにはアドバイスが 2 つ定義されている。1 つはクライアント側で通信を行うメソッドの直前で実行され、もう 1 つはサーバ側で通信を行うメソッドの直前で実行される。クライアント側で実行されるアドバイスは、クライアント側の目的のアスペクトを活性化し、通信を 1 回だけ許可する。その後、28 行目でしばらく処理を停止させ、先にクライアントで通信メソッドを呼んだ処理がサーバ側で終了するのを待っている。その後の通信は wait() によって止め、サーバ側のアスペクトの活性化を待たせる。クライアント側のアドバイスで許可された通信がサーバ側に届くと、サーバ側のアドバイスで活性化処理が行われる。サーバ側のアドバイスは、サーバ側の目的のアスペクトを活性化し、39 行目でクライアント側で停止させていた通信を再開させる。その後、活性化のタイミング制御用アスペクトは必要ないため、41 行目の unweave() で制御アスペクトを取り除いている。このアスペクトは抽象アスペクトなので、4.1 章の例と同様、このアスペクトを継承した具象アスペクトを定義するこ

```
1 public dynamic aspect DrawAspect {
2   private String[] nodes
3           = {"csg000", "csg001", "csg002"};
4   DrawSatellite ds = new DrawSatellite();
5   before(Satellite s):
6         execution(void Nbody.calc(Satellite))
7         && hosts(nodes) && args(s) {
8     ds.draw(s); }
9   public String[] getHosts(){ return nodes; }}
```

図 7: 視覚化アスペクト

とで様々なアスペクトの活性化制御に再利用できる。

5 関連研究

CaesarJ [2] は、アスペクトをファーストクラス・オブジェクトとして扱うことができるアスペクト指向言語である。この言語では、静的に織り込んだアスペクトに対して、活性状態で織り込むか、非活性状態で織り込むかをアスペクトごとに指定することができる。非活性状態のアスペクトは、アスペクトやクラスで deploy 命令を呼ぶことで活性化される。また、ブロック内のみでアスペクトを活性化をする deploy ブロックが使用できる。分散環境では遠隔ホストを表すオブジェクトを利用できる。活性化用メソッドの中でこのオブジェクトに活性化したいアスペクトのインスタンスを渡すことで、遠隔ホスト上でアスペクトを活性化できる。この操作を実行するためには、活性化する予定のアスペクトとともに活性化を実行するプログラムを事前に用意しておく必要がある。プログラム本体の実行後に作成したアスペクトを後から動的に織り込むことはできない。また、この言語には遠隔ノードの処理の実行状況を把握するための機構がないため、遠隔ノードに対する活性化の集中制御ができない。

DJAsCo は JAsCo [5] を分散環境用に拡張した分散動的アスペクト指向言語である [3]。DJAsCo は、remote pointcut を使うことができる。この言語では、アスペクトに具体的なジョインポイントは書かず、行う処理をあらわすフックを定義する。さらに、アスペクトとは別にコンテナを実装し、そこでフックを挿入する具体的なジョインポイントを指定する。動的な織り込みを行うときは、プログラムのクラスパスのディレクトリにコンテナとアスペクトを置く。すると、言語のランタイムシステムがアスペクトを動的に織り込む。活性化のタイミング制御の機構として、活性化するフックを制限することができる。コ

```
1  public abstract aspect ServerClient {
2    /* 織り込みたいアスペクト */
3    private DAspect server, client;
4    private boolean waiting = true;
5    private boolean sending = false;
6
7    public ServerClient (DAspect server,
8                         DAspect client) {
9      this.server = server;
10     this.client = client;
11     server.weave();
12     client.weave();
13     while(!server.isWoven() || !client.isWoven())
14       Thread.sleep(100);
15   }
16   /* サーバのジョインポイント */
17   abstract pointcut server();
18   /* クライアントのジョインポイント */
19   abstract pointcut client();
20
21   before(): client() {
22     synchronized(this) {
23       if(!client.isDeployed())
24         client.deploy();
25       if(sending) {
26         while(waiting) wait();
27       } else {
28         Thread.sleep(500);
29         sending = true;
30   }}}
31
32   before(): server() {
33     if(sending) {
34       synchronized(this) {
35         if(!server.isDeployed())
36           server.deploy();
37         if(waiting) {
38           waiting = false;
39           notifyAll(); }
40         if(this.isWoven())
41           this.unweave();
42 }}}}
```

図 8: 暗号化アスペクト用の活性化のタイミング制御アスペクト

ネクタでフックを具体的に定義する際、フックの制御をするクラスを指定することができる。そのクラスには、フックのリストが渡されるようになる。フックのリストには活性化されたフックが入っているため、リストからフックを取り除くことで、活性化を制御できる。しかし、このフックリストに対する処理をアドバイスとして記述することができないため、遠隔ノードの処理の進行状況を把握しつつ活性化制御することは難しい。

DyReS [6] は、アスペクトと活性化のタイミング制御を切り分けることができるフレームワークである。DyReS では遠隔ノードに対するアスペクトの活性化を集中制御することができる。タイミング制御には XML で記述したスクリプトを用いる。スクリプトには、DyReS が提供する命令を用いて活性化までの流れを記述する。活性化のタイミング制御用のフレームワークであるため、細かく制御をすることはできるが、遠隔ノード上の処理の進行状況を把握することはできない。また、スクリプトを抽象化して再利用可能にすることができず、XML で記述するため複雑なスクリプトになりやすい。

6 まとめと今後の課題

分散動的アスペクト指向における、アスペクトの活性化のタイミング制御を別のアスペクトとして分離して実装することができる言語を提案した。本言語を利用することで、分散環境における各遠隔ノードの処理の進行状況を把握しつつ、アスペクトの活性化制御ができる。このことを、N 体問題の視覚化アスペクトなどを例にとって示した。現在、本言語は設計が完成した段階で、実装は未完成である。実装を完成させることが今後の課題である。

参考文献

[1] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming, LNCS2027*, pages 327–353, London, UK, 2001. Springer-Verlag.

[2] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM.

[3] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed aop using awed. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2006. ACM.

[4] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote pointcut: a language construct for distributed aop. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 7–15, New York, NY, USA, 2004. ACM.

[5] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM.

[6] Eddy Truyen, Nico Janssens, Frans Sanen, and Wouter Joosen. Support for distributed adaptations in aspect-oriented middleware. In *AOSD '08:*

Proceedings of the 7th international conference on Aspect-oriented software development, pages 120–131, New York, NY, USA, 2008. ACM.