Institute of Electronics, Information, and Communication Engineers

社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE
SS2002-39(2003-1)

アスペクト指向プログラムに対するプログラムスライシング

石尾 隆† 楠本 真二‡ 井上 克郎‡

大阪大学大学院 †基礎工学研究科 ‡情報科学研究科
〒560-8531 大阪府豊中市待兼山町 1-3

E-mail: {t-isio, kusumoto, inoue}@ist.osaka-u.ac.jp

あらまし アスペクト指向プログラミングは，ロギングや同期処理のような複数のオブジェクトが関わる処理を単一のモジュールに記述するための新しいモジュール単位「アスペクト」を導入する．アスペクトの特徴は，ある時点・条件で実行する処理を，呼び出し処理なしに単独で記述することにある．しかし，その特性上，アスペクトを結合したプログラムの動作全体の把握が困難となり，アスペクトに関連した欠陥の除去作業等にツールによるサポートの必要性が指摘されている．本研究ではその一つの支援方法として，プログラムスライシングの適用を検討する．プログラムスライシングは，プログラムの依存関係を解析することでプログラマに必要な情報を提示する手法で，プログラム理解やデバッグなどに有効である．本論文では，アスペクト指向プログラムに対するプログラムスライシングの適用方法と，その実装への課題について説明する．

キーワード プログラムスライシング，アスペクト指向プログラミング，デバッグ，AspectJ，Java

An Application of Program Sclicing to Aspect-Orineted Program

Takashi ISHIO† Shinji KUSUMOTO‡ and Katsuro INOUE‡

†Graduate School of Engineering Science, ‡Graduate School of Information Science and Technology
Osaka University, 1-3 Machikaneyama-cho, Toyonaka-shi, Osaka, 560-8531 Japan
E-mail: {t-isio, kusumoto, inoue}@ist.osaka-u.ac.jp

Abstract Aspect-Oriented Programming (AOP) is a new technology for separation of concerns in program development. Using AOP, it is possible to modularize crosscutting aspects of a system. A main feature of AOP is that procedural codes which calls aspects are not required. Developers can write the point of time when the aspect is executed in the aspect. This feature improves separation of concerns, but it may be hard for developers to understand behaviors of the entire software including aspects. It is important for the developers to support analyzing the behaviors of aspect oriented programs. In this paper, we propose an application of program slicing, one of the promising approaches, which analyzes dependencies between program statements to assist debugging and program understanding. We also discuss the way how to calculate program slice from AOP, and describe several issues about implementation of the slicing algorithm.

Keyword Program Slicing, Aspect-Oriented Programming, Debugging, AspectJ, Java

1. はじめに

近年，プログラムの新しいモジュール化手法としてアスペクト指向プログラミングが提案され，利用されるようになってきている[1]．従来のオブジェクト指向プログラミングでは，ロギングや同期処理のような複数のオブジェクトが関わる処理を単一のモジュールに記述することはできず，その処理に参加するすべてのオブジェクトにコードが分散し，保守性を悪化させる要因となっている．このような処理は「横断要素」(crosscutting)と呼ばれている．これに対し，アスペクト指向プログラミングは，ひとつの横断要素を単一のモジュールとして記述するための新しいモジュール単位「アスペクト」を導入している．

アスペクトはオブジェクト間でやり取りされるメッセージに連動する手続きとして記述される．そのため，オブジェクトの枠にとらわれることがなく，横断要素をオブジェクトから分離でき，再利用性および保守性の向上が実現できる．アスペクトの応用事例も数多く報告されており，その有用性も示されつつある[2][3][4]．

しかし，アスペクトの導入が，プログラムに新しい複雑さをもたらすことも指摘されている．これはアスペクトの干渉問題[5]と呼ばれている．これは，複数のアスペクトが相互に干渉を与え，単体ではそれぞれ正しいはずのアスペクトが，同時

NII-Electronic Library Service

Institute of Electronics, Information, and Communication Engineers

に使うときには正しく動作しなくなるという問題である[5]. また，アスペクトはオブジェクトの振る舞いに連動するという性質のため，プログラマの予期しない時点でアスペクトが動作する等の，発見が困難な欠陥を作り込む可能性もある.

そこで，本研究では，上述した欠陥を検出するための支援方法として，プログラムスライシング技術[8]の適用を検討する. プログラムスライシングは，プログラム内部の依存関係を解析することで，プログラマが注目すべきコードを提示する技術である. アスペクト干渉など，アスペクトがもたらす複雑さは，相互の依存関係によって発生するため，依存関係を利用するプログラムスライシングが有効であると期待できる.

以降，2. ではアスペクト指向プログラミングの特徴とモデルについて説明する. 3. ではプログラムスライシングの概要とアスペクト指向プログラミングへの拡張について説明し，4.でアスペクト指向プログラムに対するプログラムスライシングツールの実装について述べ，最後に 5.でまとめと今後の課題を述べる.

2. アスペクト指向プログラミング

2.1. アスペクト指向の特徴

アスペクト指向プログラミングは，オブジェクト指向プログラミングを基に，その弱点を補うプログラミング手法である. オブジェクト指向プログラミングでは，オブジェクトの相互通信としてシステムをモデル化する. オブジェクトはそれぞれ特定の機能を担当し，その機能の実現に必要なデータを内部に保持する. オブジェクトは他のオブジェクトと相互にメッセージ通信を行い，全体としてひとつのシステムを実現する.

しかし，オブジェクトという単位でシステムの機能を分担する都合上，担当するオブジェクトを一つに決められないような特性はうまく扱えない. たとえばシステムの動作を記録していくロギング，エラーが起こったときの例外処理，データベースなどにおけるトランザクションの手続きは，システム内の複数のオブジェクトが連携して実現することが多い. このような処理は横断要素と呼ばれており，横断要素に関わる複数のオブジェクトにコードが分散するという問題がある. コードの分散は，次の 1.〜3.のような事態を引き起こす.

1. 横断要素の仕様が変わると，すべての関連したコードを変更しなければならない. そのためには，横断要素に関連したコードを正しく識別できる必要がある.
2. 横断要素を含んだオブジェクトを再利用しようとすると，その横断要素に関連したコードを取り除く，あるいは，関連したオブジェクト群をまとめて再利用するかのどちらかとなる. 横断要素の除去には，再利用しようとするオブジェクトに関する十分な知識が必要である. オブジェクト群をまとめて再利用する場合は，不要なオブジェクトや横断要素を引き継ぐことになり，システムを肥大化させる.
3. 横断要素だけを再利用することはできない. もし別の場所で同じ横断要素が必要であれば，再度実装しなければならない.

これに対して，アスペクト指向プログラミングは，横断要素を分離・記述するためのモジュール単位「アスペクト」を導入する.

アスペクト指向プログラミングでは，横断要素を捉えるために，オブジェクト指向プログラム内に含まれる実行時点を結合基準(join points)と呼び，その時点に対して手続きを結合することを可能としている. この手続きを特にアドバイス(advice)と呼び，アスペクトは，アドバイスとそれを結合したい結合基準の組を記述することによって構成される.

結合基準は言語処理系によって異なっているが，以下に挙げるような，オブジェクト間のメッセージ送受信のタイミングが一般的である.

1. オブジェクトに対するメソッド呼び出しの直前あるいは直後.
2. オブジェクトの持つフィールドへのアクセスの直前あるいは直後.

図 1 に，このプログラムのモデルを示す. オブジェクトがメッセージ通信を行うところではオブジェクト指向プログラミングと同様だが，特定のメッセージ群の送受信に対して連動する手続きがアスペクトとして定義されている.

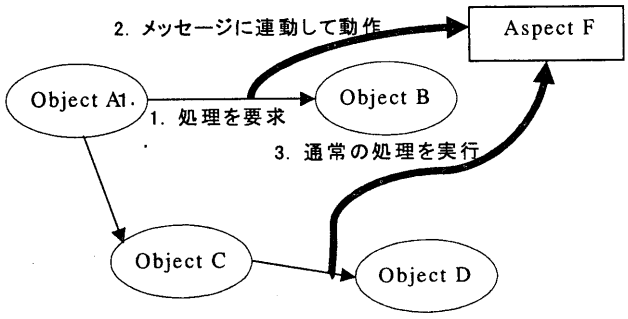

図 1：アスペクト指向プログラムのモデル

アスペクトが複数追加されようとも，オブジェクト側の記述自体は変更されない. オブジェクトは単にメ

NII-Electronic Library Service

Institute of Electronics, Information, and Communication Engineers

ッセージを送るだけであり，どのアスペクトが連動するかを決めるのはアスペクトとその言語処理系の仕事である．通常，オブジェクトがメッセージを送ると，そのメッセージに連動することを指定したアスペクトが動作し，その動作が終わった後，本来受け取るべきオブジェクトがメッセージを受け取る．オブジェクトの処理が終わると，その終了に連動するアスペクトが動作し，アスペクトの動作が終わってからメッセージを送信したオブジェクトに制御が戻る．オブジェクトが意識せずにアスペクトを呼び出すことができるため，本来は複数のオブジェクトに分散していたコードを，単一のアスペクトにまとめることが可能となる．アスペクトの利点は，コードの分散によって引き起こされる事態の逆であり，次の 1.〜3. にまとめられる．

1. 横断要素の仕様が変更されても，それに関連した全てのコードは単一のアスペクトに記述されているため，変更漏れがなく，変更の波及も認識しやすい．
2. オブジェクトには横断要素のコードが含まれないため，オブジェクトの再利用が容易となる．
3. 横断要素だけが記述されたアスペクトは，その結合基準だけを変更することで，他の場所でも再利用することができる．言語処理系によっては，これをオブジェクト指向言語における継承と同等のメカニズムとして実現している．

他の特徴として，アスペクトが結合基準に結合されるという特性上，言語処理系によっても異なるが，いくつかの特殊な情報にアクセスできる権限を与えられている．たとえば，現在実行されようとしている結合基準がどこにあるのか，メッセージに連動する場合はそのメッセージが何であるか，送り主あるいは送り先のオブジェクトがどれであるか，といったコンテキストに関する情報である．これらを用いることで，横断要素のうち，実行コンテキストに依存した処理も容易に記述することができるようになる．AspectJ によって記述された，クラス Foo へのメソッド呼び出しを記録するロギングのサンプルを次に示す．[13]

```
aspect SimpleLogging {
  // Foo へのメソッド呼び出しを表す結合基準
  pointcut Call_To_Foo_Instance(): call(* Foo.*(..));
  // 結合基準の直前に，その呼び出しを記録
  before() : Call_To_Foo_Instance {
    System.out.println(thisJoinPoint.getSignature();
  }
}
```

2.2. アスペクトの用途

アスペクトの用途については，これまでに多くの研究が行われている．たとえば，従来，オブジェクト指向プログラミングで提案されていたデザインパターンと呼ばれるオブジェクト間の連携の仕組みがあるが，そのうちのいくつかはアスペクトを用いてより簡潔な形で書き換えることが判明している[3][12]．その他にも，事前・事後条件の強制，分散オブジェクトの記述，プログラム実行の解析など，着実に応用分野が広がりつつある．

アスペクトの用途は，主として次の三つに分類される[6]．

(a) 開発途中に，開発作業を支援する目的で導入するアスペクト：実行速度などの性能計測，プログラム解析のためのメソッド呼び出し関係の記録などに利用され，最終的な製品出荷時には取り除かれる．

(b) プログラムの機能を実現するためのアスペクト：プログラムが正常に動作するために必要なアスペクトである．オブジェクトの事前・事後条件の強制，システム内での統一的な例外処理の実装，トランザクションの実行などである．

(c) 性能改善のためのアスペクト：実行速度の向上やメモリ節約などを目的とする．データの先読みやキャッシュなどがある．

2.3. アスペクトのもたらす複雑さ

アスペクトの利便性については広く認められるようになってきたが，アスペクトのもたらす複雑さについては，未だに多くの課題が残されている．それらの多くは，動作条件がアスペクト自身に記述されているという特徴から生じているものである．

1. 同一の条件下で動作する複数のアスペクトが存在しうる．アスペクトの動作順序によって，結果が異なる場合がある．
2. あるアスペクトの動作中に，他のアスペクトの動作条件が成立する場合がある．動作条件によっては，二つのアスペクトの動作が相互に条件を満たしてしまい，無限ループを生じることもある．
3. 動作条件の記述を誤ると，予期せぬアスペクトの動作を招くことがある．ソフトウェアが拡張された際に，それまで使用していた条件の完全性，正確性が損なわれることがある．

先に挙げた三つのうち，1. と 2. はアスペクトの干渉と呼ばれる問題である．アスペクトが単体

NII-Electronic Library Service

Institute of Electronics, Information, and Communication Engineers

では機能するが，複数を合わせた場合に動作しなくなることもあるため，干渉を検出する，あるいは予防するための研究が数多く成されている[7].

また，3. はオブジェクトやアスペクトを変更する際に起きる問題である．他のアスペクトが動作する条件を把握した状態でのプログラミングが必要となっており，統合開発環境などでサポートを試みる動きがあるが，複数のアスペクトが干渉している場合への対処は不十分なのが現状である．

アスペクトの干渉の例を図 2 に示す．メソッド呼び出しを記録するアスペクトと，ファイル I/O をサーバへのネットワークアクセスに変換するアスペクトである．これらは独立では正しく動作する．しかし，これらを同時に動かそうとすると，次のようになる．

1. メソッド呼び出しが発生し，アスペクトが起動される．アスペクトは，その呼び出し情報をファイルに記録しようとする．
2. データを出力の呼び出しを，ネットワークアクセスに変換するためにアスペクトが起動される．
3. ネットワーク I/O オブジェクトへデータを出力しようとするが，これもオブジェクトの呼び出し処理なので，呼び出し記録のアスペクトが起動される．
4. 再び，ファイルへの出力をネットワーク I/O に変換しようとするアスペクトが動作する．以降，アスペクトが相互に起動され続ける．

このような問題の多くは，アスペクトが他のアスペクトに対して直接，あるいは間接的に連動し，影響を与えるために生じる．これに対して，依存関係を解析し利用するプログラム解析手法の一つである，プログラムスライシングの適用を提案する．

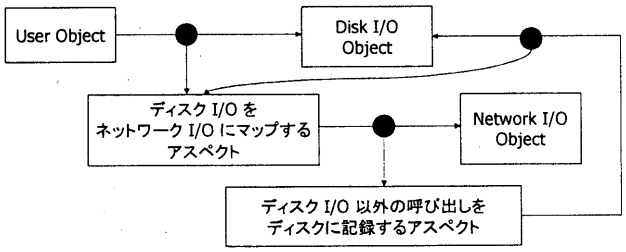

図 2: アスペクト干渉による無限ループ

3. プログラムスライシング

プログラムスライシングは，プログラムに含まれる手続き呼び出し関係や変数の参照・代入関係などの依存関係を解析し，プログラム内の注目すべき部分だけを抽出，提示する技術である[8]．具体的には，ある文のある変数を入力として，その変数の値に影響を与える文の集合を取り出す．入力として与えるコード内の変数の参照位置をスライス基点と呼び，抽出されたプログラム文の集合をプログラムスライス，あるいは単にスライスと呼ぶ．

3.1. スライス計算のアルゴリズム

プログラムスライシングは，次の三つのフェイズからなる．(1)プログラムからの依存関係の抽出，(2)プログラム依存グラフの構築，(3)グラフ探索によるスライスの抽出である．

(1)は，データ依存関係，制御依存関係の二つの依存関係を解析するフェイズである．

データ依存関係プログラム中の二つの文 s，t について以下の条件が成り立つとき，s から t に対して変数 v に関するデータ依存関係があると言う．

- 文 s で変数 v に値を代入している．
- 文 t で変数 v の値を参照している．
- 文 s から t に到達可能な制御フローがある．
- 文 s から t に到達する制御フローのうち，途中で変数 v への代入文がないような経路が少なくともひとつ存在する．

また，制御依存関係は，プログラム中の文 s，t について，以下の条件が成り立つとき，s から t への制御依存関係が存在すると言う．

- s が条件節である．
- t が実行されるかどうかが，s の結果によって決まる．

フェイズ(2)で，これらの依存関係と，手続き・メソッドの呼び出し関係を用いて，プログラム文を頂点，依存関係を有向辺としたグラフを作成する．このグラフをプログラム依存グラフ（Program Dependence Graph，以下 PDG)と呼ぶ．フェイズ(3)では，PDG 上でスライス基点となる頂点を選び，有向辺を逆向きに探索していくことで，ある注目したい文，変数に影響を与える文を抽出していく．得られた頂点集合をエディタなどに表示されたソースコード上へ反映し，プログラマへ情報を提供する．

プログラムスライシングの性能は，その依存関係の抽出方法によって決まる．情報をソースコードから取得するか，実行時にプログラムの動作を監視して取得するかによって，静的スライスと動的スライスに分けられている[8][9]．静的スライスはすべての可能性を抽出するため，プログラム理解や検証などに用いられる．一方，動的スライスはある特定の入力に対して実際に発生した依存関係だけを解析することで，静的スライスよりも対象とするコードを絞り込むため，デバ

NII-Electronic Library Service

Institute of Electronics, Information, and Communication Engineers

ッグ等の支援に用いられる．

動的スライスは，プログラムの実行系列を監視する必要があるため，実行時のコストが非常に高くつくという問題があったが，これに対して，制御構造は静的に解析し，データ依存関係のみ実行時に取り出すDependence-Cache(DC)スライスが提案されている．DCスライスは，実用的なコストでスライスサイズを減少させることが知られている[10][11]．

DCスライスの例を次に示す．14行目の変数xをスライス基準とすると，プログラムの6行目，11行目から13行目の下線部分が取り除かれる．6行目は決して実行されない文なのでDCスライスには含まれない．

```
1 : class Foo {
2 :     public static void main(String[] args) {
3 :         int x = 0;
4 :         for (int y=0; y<100; y++) {
5 :             if (x > 1000) {
6 :                 x = 0;
7 :             } else {
8 :                 x += y % 2;
9 :             }
10:         }
11:         if (y == 100) {
12:             y = 0;
13:         }
14:         System.out.println(x);
15:     }
16: }
```

制御依存 x

データ依存 x

3.2. アスペクト指向プログラムへのスライスの拡張

アスペクト指向プログラムに対してプログラムスライシングを適用し，デバッグを支援することを考える．開発者がバグを発見した際に，その原因となるコードを探し出す作業が必要となるが，そこで扱わなければならないコード量をいかに減らすかが重要となる．また，同一の結合基準に対して複数のアスペクトが連動することがあるが，その動作順序は，開発者が特に指定していない限りは処理系依存となり，ソースコードを解析しただけでは十分な情報を開発者に提供できない．そこで，本研究では，実行時情報を使用するDCスライスを用いる．発見されたバグを再現させるテストケースを入力としてプログラムを実行し，その実行を観測して得られた情報を基にプログラム依存グラフの構築とスライス計算を行い，開発者にフィードバックを行う．

現在，オブジェクト指向言語Javaに対するプログラムスライシングが既に実現されているが，アスペクト指向プログラムに対する適用は基本的な手段の提案のみであり，詳しい議論はなされていない[14]．

本研究では，対象とするアスペクト指向言語としてAspectJを選択し，Javaに対するプログラムスライシングを拡張することで実現する．具体的には，アスペクトが結合された場所からその動作するアスペクトへと手続き呼び出しと同様の関係があると考え，その依存関係をPDGに追加する．

DCスライスの計算には対象プログラムの実行が必要だが，アスペクトの干渉による無限ループの発生がそれを妨げることになる．これに対しては，静的な依存関係のチェックが必要である．メソッドの呼び出し関係とアスペクトの連動関係をコールグラフ(Call Graph)として表現し，開発者に提示するものとする．

コールグラフの例を図3に示す．Mn，Anはそれぞれメソッドとアスペクトを表現する頂点で，Mnへ向かう辺はそのメソッドを呼び出すことを，Anへの辺はアスペクトが起動されることを意味する．たとえば，M1→A1の辺はM1の処理中にA1が起動される可能性があることを表し，A3→M2→A2の経路の存在はA3が呼び出すM2がA2を起動するので，A3の動作がA2に干渉されていることを表す．また，A1→A2→M3→A1のループは，無限ループの可能性を表している．このようなグラフを用いて，アスペクト間の干渉が意図したものであるかどうか，またループを自動検出することで無限ループの可能性を検知し，ツールの利用者に提示するものとする．開発者が意図して再帰ループなどを作成する可能性もあるため，そのまま実行することも選択できるものとする．

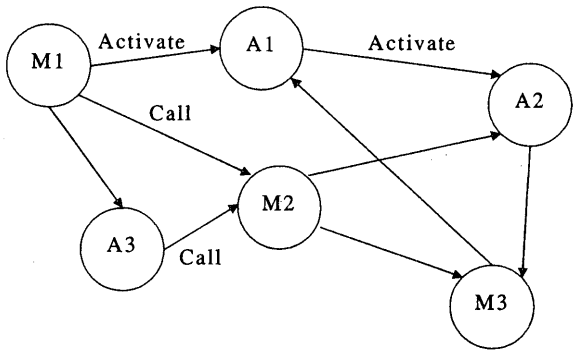

図3: アスペクトを含むコールグラフ

4. 実装方針

デバッグ作業は，コードを書き換え，テストケースを再実行するという反復作業として考えられる．このような反復テストをサポートする統合開発環境も多く，プログラムスライシングのよう

NII-Electronic Library Service

Institute of Electronics, Information, and Communication Engineers

な支援ツールも，それらと統合して利用可能にすることが望ましい．本研究では，統合開発環境としてEclipse[15]を使用し，プラグインとしてスライス計算環境を追加する形で実装するものとする．

DC スライス計算には実行時のプログラムを監視する処理が必要である．この処理自体も横断要素の一つとして考えると，アスペクトを用いて実現することが自然である．AspectJ ではローカル変数の監視はできないが，実行時のオブジェクトの個々のインスタンスの識別と，フィールドのデータ依存関係を取り出すだけで十分有効であることは Java を対象としたプログラムスライシングの実現で既に示されている[2]．

システムの動作概要を図に示す．利用者はまず静的な呼び出し関係をツールによって調べ，ループが存在する場合はそれを実行しても問題ないか確認する．次に実行を監視するアスペクトを追加し，そのアスペクトの動作が他のアスペクトの振る舞いの影響を受けないこと，あるいは影響を受けても問題ないことを確認する．そしてテストケースを与えてプログラムを実行し，DC スライスに必要な実行時情報を抽出する．得られた情報を静的な解析情報と合わせて PDG を構築し，ユーザがエディタから指定したスライス基準から，グラフ探索を行ってスライスを計算，結果をエディタに反映する．

5. まとめ

アスペクト指向プログラミングは新たなモジュール単位「アスペクト」を導入し，オブジェクトに分散する横断要素の分離を実現する．

アスペクトは，オブジェクトの動作に連動する手続きの集合として記述され，オブジェクトを書き換えることなくオブジェクトの動作を変えることができる．

アスペクトの利点は多いが，オブジェクトから明示的な呼び出し処理なしに実行されるため，予想外の場所でアスペクトが連動してしまう，あるいは複数のアスペクトが連動するために動作が予想できない，といった問題が発生することがある．これに対して，依存関係を用いるプログラム解析手法のひとつ，プログラムスライシングが有効であると考えられる．

アスペクト指向プログラムに対してプログラムスライシングを適用することは，アスペクトの連動をメソッドの呼び出しと同等に考えることで実現される．ただし，同一の結合基準に複数のアスペクトが連動する場合，その動作順序はコンパイラや実行環境の実装に依存するため，単純な静的スライスではデバッグ支援には不足だと考えられる．これに対し，データ依存関係を実行時に抽出する DC スライスを用いるものとする．

DC スライスを計算するには，プログラムの実行が必要となるが，アスペクトが干渉して無限ループを生じた場合は，プログラムを実行できない．これに対しては，メソッド呼び出しとアスペクトの連動関係だけを抽出したコールグラフを生成し，ループの可能性を開発者に提示するものとする．

スライスツールは統合開発環境 Eclipse をベースに，プラグインの形式で実装中であり，その評価を今後の課題とする．

文　献

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier and J. Irwin: "Aspect Oriented Programming", Proceedings of ECOOP, vol.1241 of LNCS, pp.220-242(1997).

[2] 石尾隆，楠本真二，井上克郎: "アスペクト指向プログラミングの動的スライス計算への応用", 2002 年電子情報通信学会総合大会講演論文集，D-3-4，p.30 (2002).

[3] J. Hannemann, G. Kiczales: "Design Pattern Implementation in Java and AspectJ", Proceedings of OOPSLA 2002, pp.161-173, Nov. (2002).

[4] S. Soares, E. Laureano, P.Borba: "Implementing Distribution and Persistence Aspects with AspectJ", Proceedings of OOPSLA 2002, pp.174-190, Nov. (2002).

[5] R. Pawlak, L. Seinturier, L. Duchien, G. Florin: "JAC: A Flexible Solution for Aspect-Oriented Programming in Java", Proceedings of REFLECTION 2001, pp.1-24 (2001).

[6] I. Kiselev: "Aspect-Oriented Programming with AspectJ", Sams Publishing, Indiana (2002).

[7] 一杉裕志，田中哲，渡部卓雄: "安全に結合可能なアスペクトを提供するためのルール", ソフトウェア科学会第 19 回大会, Sep.(2002).

[8] M. Weiser: "Program slicing", IEEE Transactions on Software Engineering, SE-10(4):352-357(1984).

[9] H. Agrawal and J. Horgan: ``Dynamic Program Slicing'', SIGPLAN Notices, Vol.25, No.6, pp.246-256 (1990).

[10] T. Takada, F. Ohata, K. Inoue: "Dependence-Cache Slicing:A Program Slicing Method Using Lightweight Dynamic Information", Proceedings of the 10th International Workshop on Program Comprehension (IWPC2002), pp.169-177, Paris, France, June (2002).

[11] F. Ohata, K. Hirose, M. Fujii, and K. Inoue: "A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic Information", Proceedings of APSEC2001, pp.273-280(2001).

[12] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley (1995).

[13] AspectJ Team, "The AspectJ Programming Guide", http://aspectj.org/doc/dist/progguide/

[14] J.Zhao, "Slicing Aspect-Oriented Software", Proceedings of IWPC2002, pp.251-260 (2002).

[15] Eclipse Project, http://www.eclipse.org/

NII-Electronic Library Service