推薦論文●第 9 回プログラミングおよびプログラミング言語ワークショップ (PPL2007)

プログラムスライシングを用いた機能的関心事の抽出

石尾 隆† 仁井谷 竜介† 井上 克郎†

ソフトウェアは数多くの機能的関心事，すなわち，特定の機能要求を実現するためのソースコードの集合から構成される．1 つの関心事に属するソースコードは，通常，複数のモジュールに分散しているため，開発者がある関心事を理解するには，その関心事に含まれるソースコードを探し，それらの間の制御フローやデータフロー情報を調べる必要がある．

本研究では，開発者が注目しているプログラム要素群に対し，それらのプログラム要素に依存関係を持つ他のプログラム要素を探索し，要素間の関係を関心事グラフとして可視化する手法を提案する．具体的には，プログラム依存グラフの探索に基づくプログラムスライシング手法に対して，プログラム構造やキーワードマッチに基づく経験的な指標を導入し，関心事に含まれない可能性が高い要素に対するグラフ探索を打ち切るよう拡張する．Java を対象とした解析ツールとして本手法を実装し，従来のプログラムスライシング手法に基づく手法との比較実験を行った結果，人間が作成したものにより近い関心事グラフを抽出していることを確認した．

A functional concern, code that helps fulfill a functional requirement, is typically implemented by collaborative software modules. When developers understand the implementation of a concern, they need to find code fragments contributing to the concern and understand how the modules collaborate with one another.

In this paper, we propose an automatic approach to extract program elements closely related to developer's interest and visualize the relationship among the elements as a concern graph. We extend program slicing by introducing heuristics to calculate the degree of interest to a developer among program elements for excluding unrelated elements from a program slice. We have implemented our approach as a slicing tool for Java software and conducted an experiment. The result shows that our approach extracts a concern graph more suitable to understand a concern than a traditional program slicing.

1 まえがき

ソフトウェアは数多くの機能的関心事から構成される．ここでの機能的関心事とは，開発者がひとまとまりの単位でをあると考える，特定の機能要求を実現するためのソースコードの集合である．1 つの関心事を構成するソースコードは，一般に複数のモジュールに分散しているため，ソフトウェア保守作業において何らかの改修ないし拡張を行う際には，どのメソッドのどのコードが関心事に含まれており，どのように相互作用を行っているかを理解する必要がある [21]．ソフトウェアを保守していく過程では，1 つの関心事に対して複数回の変更作業が発生するため，その都度，関心事に該当するメソッドの再発見と理解が必要になる [28]．このようなプログラム理解に関するコストは，ソフトウェアのライフサイクル全体で半分以上を占めるといわれている [7]．

開発者が関心事を発見し理解する作業は，関心事の一部であるようなプログラム文やメソッド，フィールドなどのプログラム要素のいくつかを特定する作業と，それらの要素に何らかの関係を持った周辺のコードを探索していく作業からなる．関心事に含まれるプログラム要素を探す代表的な方法はキーワード検索であり，Feature Location [28] [37] や部品検索 [11] といった手法も利用可能である．そして，発見さ

Locating a Functional Concern Based on a Program Slicing Technique.

Takashi Ishio†, Ryusuke Niitani†, Katsuro Inoue†

† 大阪大学 大学院情報科学研究科, Graduate School of Information Science and Technology, Osaka University

れたプログラム要素と関連するほかのプログラム要素を，メソッド呼び出し関係やクラス階層などの情報を参照しながら発見していき，また，それらがどのような制御フロー，データフローによって相互作用を行うかを調査することで，関心事の理解を行う．

本研究では，関心事の理解を支援するために，プログラムスライシング [35] を拡張した手法を提案する．提案手法は，開発者が関心事の手がかりとして発見したソースコード行の集合と，関心事を表すキーワード文字列を入力として受け取り，関心事に該当するクラス，メソッド，フィールド間の関係を関心事グラフ [27] として出力する．

プログラムスライシングは，制御依存関係やデータ依存関係に基づいてプログラム依存グラフを構築し，入力として与えられた文に影響を与える，あるいは影響を受ける可能性のあるすべてのプログラム文をグラフ探索によって抽出する手法である．通常のプログラムスライシングは，アプリケーションのエントリポイントや一般的なデータ構造を表すユーティリティクラスなども含めて網羅的な探索を行っていくため，開発者が理解しようとしている関心事に特有のプログラム要素のみを抽出することはできず，関心事の理解にそのまま適用することはできない．そこで，本研究では，プログラム中の各メソッドについて，開発者の着目する関心事に該当するかどうかを評価するための経験的な指標を導入する．関心事の一部ではないと推測されるプログラム要素に対するグラフ探索を打ち切ることで，関心事に含まれることが期待されるメソッド中の文だけをプログラムスライスとして出力する．経験的な指標としては，プログラム依存グラフの各頂点についての辺の入次数，探索したい関心事を示したキーワードを含んだメソッドとの距離を用いた．この指標における距離の計測には，プログラム依存グラフの頂点をメソッド単位に集約したグラフを用いている．

本手法の出力である関心事グラフは，クラス，メソッド，フィールドを頂点とし，クラスの継承関係やメソッド呼び出し関係，フィールドの参照，更新など関心事の静的な構造の概要を記述するためのグラフである [27]．得られたプログラムスライスを関心事グラフに変換し可視化することで，開発者は入力として与えたソースコード集合の関係を把握することができ，関心事の理解を進めることができる．

提案手法をツールとして実装し，そのツール自身とjEdit[†1] を題材に適用実験を行った結果，経験的な指標に基づくグラフ探索の打ち切りによって，小さなプログラムスライスを抽出できることを確認した．また，抽出された関心事グラフについて，人間が手作業で作成した関心事グラフとの比較を行い，従来のスライシング手法によって得られる結果よりも人間が作成したものに近い関心事グラフを抽出していることを確認した．

以降，2 章では，研究で用いているプログラムスライシングと関心事グラフについて解説する．3 章では提案手法について，また 4 章では評価実験について，それぞれ述べる．5 章で関連研究に対する位置づけを行い，最後に 6 章でまとめと今後の課題について述べる．

2 背景

2.1 プログラムスライシング

プログラムスライシング（*Program Slicing*，以降スライシング）とは，プログラム中の文間の依存関係に注目し，**スライシング基準**（*Slicing Criterion*，$< s, v >$ で表される文 s と変数 v の対）に依存関係のある文の集合を抽出する技術である [35]．抽出された文の集合は，プログラムスライス，あるいは単にスライスと呼ばれる．

本研究では，スライシング技術として，プログラム依存グラフ（Program Dependence Graph，PDG）に基づく手法を用いる [10]．プログラム依存グラフは，頂点としてソースプログラムの各文に対応する頂点およびメソッドの引数などを扱うための特殊頂点を持ち，それらの頂点間を制御依存関係（Control Dependence）やデータ依存関係（Data Dependence），メソッド呼び出し，パラメータ渡しなどを表す辺によって接続した有向グラフである．制御依存関係とは，文 s_1 が制御文であり，s_1 の結果によって文 s_2

†1 jEdit http://www.jedit.org/

が実行されるかどうかが決定されるとき，文 s_1 から s_2 に対し成り立つ関係である．また，データ依存関係とは，変数 v を定義している文 s_1 から，v を参照している文 s_2 へ，v を再定義しない実行経路が少なくとも 1 つ存在するとき，文 s_1 から s_2 に対し変数 v に関して成り立つ関係である．

スライスは，スライシング基準に対応する PDG 上の頂点を基点として，グラフを逆方向または順方向へ，メソッドの call-return 関係を維持しながら探索することによって得られる [24]．逆方向への探索は，基点となった変数に影響を与える文の集合を意味するバックワードスライス（Backward Slice）の抽出となり，順方向への探索は，基点となった変数の影響を受ける文の集合を意味するフォワードスライス（Forward Slice）の抽出となる．

本研究では，ツールへの入力を簡単化するため，スライシング基準はプログラム文の集合によって与えられるものとし，各プログラム文で定義または参照されているすべての変数をスライシング基準とみなしたスライシングの計算を行っている．

2.2 スライシングを関心事理解支援に用いるときの問題点

スライシングは，スライシング基準に影響を与える，あるいは影響を受ける可能性のある文を全て抽出する手法であるため，多数の文が出力される．Binkley らによる実験では，C 言語のプログラムにおいて，任意の文を基準としてスライスを実行した結果，平均で 28 %のプログラム文がスライスに含まれたことが報告されている [1]．この問題の原因の 1 つは，プログラムの実行に影響を与えるアプリケーションのエントリポイント（Java における main メソッド）や各関心事で内部的に使われる一般的なデータ構造など，どの関心事に対しても依存関係を持つプログラム要素がスライスに含まれることである．しかし，実際には，アプリケーションに固有でないユーティリティは，特定の関心事のみを理解しようとするとき，理解する必要がないことが多い [8]．

また，多くのプログラムスライスは，他の関心事を経由した，長い依存関係の列を多く含んでいる．例えば，アプリケーションのエントリポイントから，スライシング基準までの推移的な制御依存関係の列などは，デバッグなどの用途で有益な場合もあるが，途中で実行される他の様々な機能も，スライスへと含めてしまう．その結果，異なる関心事を発見するために，異なるスライシング基準を与えたとしても，まったく同一のスライスとなってしまうことがある．

表 1 異なる関心事に対する従来のスライスの差分

	頂点数	メソッド数	クラス数
Autosave	364,672	4,031	562
Undo	364,752	4,038	562
差分	86	7	0

表 1 は，jEdit に対して，ファイルの自動保存機能（Autosave）と，ファイルの変更を元に戻す機能（Undo）に対応するスライスを計算し，その差分を求めた結果である．スライシング基準としては，jEdit のソースファイルのうち Autosave.java と UndoManager.java に対して grep を適用し，キーワード “autosave”，“undo” を持つ文の集合を与えた．また，比較したスライスは，フォワードスライスとバックワードスライスの和集合である．

Autosave, Undo ともに非常に多くの頂点が得られたにもかかわらず，その差分はきわめて小さい．この原因の 1 つは，Undo 機能によってファイルの変更内容を破棄する処理が，Autosave の実行条件である「ファイルが変更済みかどうか」を表現するフラグを書き換えるためである．このように，複数の関心事は相互に依存関係を持っていることから，特定の関心事のみを通常のスライシングで抽出することは困難である．

実際に自動保存機能の仕組みを開発者が理解する場合には，定期的にファイルの自動保存処理を起動するタイマーを備えた Autosave クラスや，その実際の処理を行う Buffer クラスの autosave メソッド，BufferedIORequest の autosave メソッド，また BufferedIORequest クラスや FileVFS クラスに含まれるファイルへの書き込み処理，AutosaveBackupOptionPane と jEdit クラスに含まれるオプショ

ン設定値の編集処理などの情報が有用である．その一方で，Undo をはじめとする他のエディタの機能に関わるクラスや，ログ記録用の Log クラス，編集中の文字列のデータ構造の実装クラスなどは，理解する必要のない情報である．

2.3 スライシングに基づく理解支援手法

開発者が着目した特定のプログラム文についての局所的な依存関係を抽出する手法として，Chopping と Length-Limited Slicing が提案されている．

Chopping は，ある文から *src* 別の文 *dest* への影響を調べるための手法である [12]．これは，プログラム依存グラフ上で，*src* に対応する頂点から，*dest* に対応する頂点へのすべての到達経路を探索することによって求められる．Chopping の計算では，始点および終点として選ばれた頂点をバリア（Barrier）として，スライシングの途中経路に登場することを禁止することで，始点から出発して再び始点に戻ってくるような循環経路を除去している [16]．

Chopping を使うには，どの文が始点であり，どの文が終点であるかを開発者が指定する必要がある．そのため，調べている対象のソフトウェアの構造についての知識が必要となる．しかし，開発者が関心事に対して特に知識がなく，検索などで手掛かりとなる要素を発見した状態で，ただちにこのような区別を行うことは困難である．

Length-Limited Slicing は，開発者が興味を持つプログラム文は依存関係上で局所的に存在するという仮定に基づいて，スライシング基準からグラフ探索を行う距離を制限したスライシングである [17]．この手法は，頂点への訪問をスライシング基準から距離 k 離れた頂点で停止する．k の値を小さくすることで結果のプログラム文の集合を小さくすることはできるが，小さな k の値を選んでも，関心事を理解するために適切な情報が得られるとは限らない．

プログラムスライシング手法 [10][35] で抽出されるプログラムスライスは，それ自体実行可能なプログラムであり，たとえば手続きの抽出 [15] に利用されている．一方，Chopping や Length-Limited Slicing はプログラム理解に役立つ小さいプログラム文の集合を求めることに主眼を置いており，得られるスライスは実行不可能である．我々の提案手法も同様に，抽出結果は実行不可能なスライスとなる．

2.4 関心事グラフ

関心事グラフ（*Concern Graph*）は Robillard らによって提案された，関心事を表現するためのグラフである [27]．このグラフは，関心事に含まれるプログラム要素とその関係を抽象化したグラフで，関心事に含まれるクラス，メソッド，フィールドを頂点とし，メソッド呼び出し calls，フィールドの読み出し reads など，各要素間の関係を辺とする．

関心事は，ソースコードとしては複数のファイルに分散して記述されているため，ツールの出力としてソースコードの断片や位置情報を直接出力しても全体像が把握しにくいという問題がある．そこで，本研究では関心事グラフを出力形式として採用し，関心事に含まれたプログラム要素の関係を開発者に提示するものとした．

関心事グラフの構築に必要な情報とは，メソッド呼び出しやフィールドの読み書きなどであり，プログラムスライスから取得可能である．スライスから自動的に関心事グラフの辺を生成するためのルールは既に亀田らによって提案されており，表 2 はその一部を抜粋したものである [13]．亀田らのルールは，本来ソフトウェア中の横断的関心事を抽出するために提案されたものであるが，それ自体は一般的なスライシングの結果に適用可能である．そのため，我々の手法で得られたスライスを関心事グラフとして可視化する際にもこの変換ルールを用いた．

3 提案手法

開発者が注目している関心事についての情報を開発者に簡潔に提示するため，プログラムスライシングを用いて情報を抽出し，関心事グラフに変換する手法を提案する．

本手法は，入力として，関心事の一部であると開発者が判断したプログラム文の集合と，調べたい関心事を表すキーワードを受け取る．入力となるプログラム文の集合は，キーワード検索や様々な Feature

表 2 スライスに基づく関心事グラフの辺の生成ルール

種類	生成条件
(calls m_1 m_2)	スライスがメソッド m_1 からメソッド m_2 への呼び出し辺を含む
(reads m f)	スライスがフィールド f からのデータフローを持つメソッド m の頂点を含む
(writes m f)	スライスがフィールド f へのデータフローを持つメソッド m の頂点を含む
(creates m c)	スライスがクラス c のインスタンスを生成するメソッド m の頂点を含む
(declares c f|m)	スライスがクラス c のフィールド f またはメソッド m を含む
(superclass c_1 c_2)	関心事グラフがクラス c_1 と親クラス c_2 を含む

Location 手法を用いて特定されたものを想定している．本手法の出力は，関心事グラフである．

本手法は，関心事に含まれるプログラム要素がプログラムスライスに優先的に含まれるよう，グラフの構造に基づくヒューリスティクスと，開発者が与えたキーワードとメソッド名とのマッチングに基づくヒューリスティクスとを用いて，プログラムスライシングにおけるグラフ探索を打ち切る仕組みをプログラムスライシングに導入する．

本手法は，次の 5 つのステップで構成される．

1. 対象ソフトウェアのプログラム依存グラフを構築する．
2. プログラム依存グラフの各要素に対してヒューリスティクスの評価を行い，グラフ探索を打ち切るバリアとなる辺の集合を計算する．複数のヒューリスティクスが個別にバリアとなる辺を特定し，それらの辺の和集合を最終的に使用するバリアとする．
3. 入力として指定されたプログラム文の集合をスライシング基準として，バリア付きのスライシングを適用する．
4. 得られたスライスの中から，ライブラリに属する頂点のフィルタリングを行う．
5. フィルタリングの結果を関心事グラフとして可視化する．

以降，各ステップの詳細について述べる．

3.1 プログラム依存グラフの構築

本研究では，対象として Java を選択した．プログラム依存グラフには，Zhao の *JSDG*（the software dependence graph for Java）[36] の定義を用いた．

プログラム依存グラフの各頂点は，Java のソースコードではなく，バイトコードとの中間表現であり三番地コードの一種である *Jimple* [32] の各命令に対応させた．JSDG では，プログラム文に該当する頂点以外に，いくつかの特殊頂点が使用される．また，制御依存辺，データ依存辺のほかに，メソッド呼び出し辺，パラメータ渡し辺，フィールド依存辺が使用される．

3.1.1 Jimple

Jimple は Java ソースコードとバイトコードの中間表現であり，三番地コードの一種である．図 1 に，Java のソースコード（左）とそれに対応する Jimple コード（右）を示す．Jimple コードは，1 つの命令では高々1 つのメソッドしか呼び出さず，また高々 1 つの変数しか値を更新しないことから，データフロー解析が容易であるという特徴がある．制御構造は `if/goto` 命令として表現されるほか，ソースコード上では記述が省略されているデフォルトのコンストラクタ（`<init>`）などのコードが補完される．また，通常のメソッド呼び出しには `virtualinvoke` が対応する．例えば，Java の 15 行目の `notZero(i)` には，Jimple の 46 行目が対応し，戻り値は `$z0` に格納される．

3.1.2 Jimple をもとに構築されるプログラム依存グラフ

本研究で用いたプログラム依存グラフは，次の頂点によって構成される．

Jimple 命令頂点 ラベルや宣言を除いた，Jimple

```
01 public class Foo
02 {
03   public void nothing()
04   {
05   }
06
07   public boolean notZero(int i)
08   {
09     return i != 0;
10   }
11
12   public void ifStatement()
13   {
14     int i = 10;
15     if (notZero(i)) {
16       nothing();
17     } else {
18       nothing();
19     }
20   }
21 }
```

```
01 public class Foo extends java.lang.Object
02 {
03   public void <init>()
04   {
05     Foo r0;
06
07     r0 := @this: Foo;
08     specialinvoke r0.<java.lang.Object:
         void <init>()>();
09     return;
10   }
11
12   public void nothing()
13   {
14     Foo r0;
15
16     r0 := @this: Foo;
17     return;
18   }
19   public boolean notZero(int)
20   {
21     Foo r0;
22     int i0;
23     boolean $z0;
24
25     r0 := @this: Foo;
26     i0 := @parameter0: int;
27     if i0 == 0 goto label0;
28
29     $z0 = 1;
30     goto label1;
31
32   label0:
33     $z0 = 0;
34
35   label1:
36     return $z0;
37   }
38   public void ifStatement()
39   {
40     Foo r0;
41     int i0;
42     boolean $z0;
43
44     r0 := @this: Foo;
45     i0 = 10;
46     $z0 = virtualinvoke r0.<Foo: boolean notZero(int)>(i0);
47     if $z0 == 0 goto label0;
48
49     virtualinvoke r0.<Foo: void nothing()>();
50     goto label1;
51
52   label0:
53     virtualinvoke r0.<Foo: void nothing()>();
54
55   label1:
56     return;
57   }
58 }
```

図 1　Java ソースコード（左）に対応する Jimple コード（右）

の実行可能な命令に対応する．1 頂点は，1 つの数値演算，メソッド呼び出し，フィールド参照，フィールド更新，分岐命令などに対応する．

Entry 頂点　メソッドの実行開始位置を表現した頂点で，各メソッドに必ず 1 つだけ存在する．メソッドの最初の命令文への制御フローを持つ．

Exit 頂点　メソッドの実行終了を表現した頂点で，各メソッドに必ず 1 つだけ存在し，すべての return 文からの制御フローを持つ．制御依存関係を計算するために便宜的に用いられる．

仮引数頂点　メソッド宣言の引数および戻り値に対応して生成される．メソッドの引数に対応する頂点は，データ依存解析において，各メソッドの先頭で，仮引数の値を定義した頂点として扱う．また，戻り値に対応する頂点は，データ依存解析において，return 文が定義した戻り値を参照する頂点として扱う．

実引数頂点　メソッド呼び出し命令ごとに，かつ，動的束縛で実際に実行される可能性のあるメソッドの候補ごとに，実引数および戻り値の受け渡しを表現するために生成される．

各頂点は，以下の有向辺によって接続される．

制御依存辺　ある分岐命令 v_{from} から，実行されるかどうかが v_{from} の結果によって直接決定される命令 v_{to} があった場合，v_{to} は v_{from} に依存する．例えば，Jimple の 47 行目の if 命令は，49, 50, 53 行目への制御依存辺を持つ．また，あるメソッド呼び出しに対応する実引数頂点には，そのメソッド呼び出し命令の頂点からの制御依存辺を接続する．

データ依存辺　変数の定義と参照に対応する辺で，ある変数の値を定義する命令 v_{from} と，その変数の値を参照する命令 v_{to} があった場合，v_{to} は v_{from} に依存する．例えば，Jimple の 45 行目は `i0` に関して，46 行目への依存辺を持つ．メソッド呼び出しの実引数に関してはデータ依存のみ扱いが異なり，メソッド呼び出し命令の頂点ではなく，対応する実引数頂点へとデータ依存辺を接続する．戻り値を参照している場合も，対応する実引数頂点からのデータ依存辺を接続する．

呼び出し辺　呼び出し辺はメソッド呼び出しに対応する辺で，メソッド呼び出し文に対応する頂点から，呼び出されるメソッドの Entry 頂点へと接続される．動的束縛によって複数のメソッドが呼ばれる可能性がある場合は，可能性のあるすべてのメソッドへと辺を接続する．

パラメータ渡し辺　メソッド呼び出しによって渡される引数および戻り値のデータ依存に対応する辺である．メソッド呼び出し側の実引数頂点から，呼び出し先の対応する仮引数頂点へと接続される．また，戻り値については，呼び出し先の仮引数頂点から，呼び出し側の実引数頂点へと接続される．

フィールド依存辺　フィールドの値の更新および参

```
Child1 c1 = new Child1();
Child2 c2 = new Child2();
Parent p = (condition) ? c1 : c2;
p.something();
```

図 2 エイリアシングの例

照によって起こりうるデータ依存に対応する辺である．フィールドの値を更新する命令の頂点から，そのフィールドの値を参照する命令の頂点へと接続される．

Java ではオブジェクトが参照として扱われるため，異なる変数が同じオブジェクトを指すエイリアシングが生じる可能性がある．これに対しては，Points-to Set 解析 [18] を適用することで，各変数について実際に参照しうるオブジェクトの集合を計算しておく．その結果とクラスの継承関係から動的束縛の可能性を調べ，実行時に起こりうるすべてのメソッド呼び出しについて実引数頂点の生成，呼び出し辺とパラメータ渡し辺の接続を行う．フィールド依存辺については，フィールドの参照命令とフィールドの更新命令のペアに対して，同じオブジェクトを参照する可能性があればフィールド依存辺を接続する．

図 2 は，Parent が Child1, Child2 の親クラスで，変数 p が c1, c2 のどちらかを指している例である．このとき p.something() の呼び出し辺は，Child1#something と Child2#something の両方に接続される．

本研究では，例外処理（Java における try-catch 文）によって発生する特別な制御フローには対応しておらず，例外による脱出頂点（Exceptional Exit）は作成していない．catch 節については，catch 節の中の命令から，それ以降に続く命令への制御フロー，データフローは解析しているが，try 節のどの命令から catch 節に移動しうるかは解析していない．

3.2 ヒューリスティクスを用いたバリアの特定

本手法で用いるヒューリスティクスは，適用対象となるプログラム依存グラフ PDG とヒューリスティクス用のパラメータ（キーワードおよび閾値） $options$ に基づいて，PDG に含まれる辺の集合 E からバリアとなる辺を特定する関数として表現できる．

$$PDG \times options \rightarrow 2^E$$

複数のヒューリスティクスによって複数のバリア辺の集合 $E_1, \ldots, E_n$ を得られるので，それらの和集合を，次のステップで行われるスライシングへの入力パラメータとする．

ヒューリスティクスが利用できる情報としては，プログラム依存グラフの持つ構造に関する情報と，各頂点に対応した命令が持つ識別子や型などの意味的な情報がある．本手法では，構造の情報に基づくヒューリスティクスとして辺の入次数を用いたヒューリスティクス，意味的な情報に基づくヒューリスティクスとして識別子に対するキーワードマッチを用いたヒューリスティクスを提案する．

3.2.1 入次数の大きい頂点に入る辺

ソフトウェアの依存関係はべき乗則に従い，プログラム中の特定のプログラム要素に対して依存関係が集中している [22] [33]．べき乗則では，ある頂点の依存辺の入次数が x である確率が $P[X = x] \sim x^{-a}$ で表される．ここで分布を近似するための定数 a は，$a > 1$ の実数であり，対象ソフトウェアによって多少異なるが，2 に近い値をとる†2．

多数のメソッド呼び出しが集中する（入次数が大きい）要素としては，Java における String クラスや，java.util パッケージに含まれるコレクションクラス，横断的関心事を実現するためのクラスなどがある [20]．このような場合は，呼び出し側のメソッドの機能と，呼び出し先のメソッドの機能との独立性が高いと考えられる．

そこで，PDG の依存辺の入次数 $indegree$ が閾値 th より大きな頂点 v_{to} に入る辺をバリアとする．

$$isBarrier(v_{from}, v_{to}) = indegree(v_{to}) > th$$

閾値は手動で指定するか，あるいは自動で頂点数

†2 被参照数が 1 であるような頂点の割合が大きいほど，a の値は大きくなる．約 38 %のとき a=1.5，60 %のとき a=2，83 %のとき a=3，92 %のとき a=4 となる．

N に対して $\sqrt{N}$ を与える．依存辺の入次数が x より大きい頂点を持つ確率が $P[X > x] \sim x^{-(\alpha-1)}$ で表されることから，$x = \sqrt{N}$ のとき，オーダー $O(\sqrt{N})$ 個の頂点がバリアの対象の頂点となることが見積もられる．

このヒューリスティクスは，以降の図表中では I と略記する．また，入次数の対象となる辺を全ての辺としたものは Ia，PDG 上の呼び出し辺のみ（あるメソッドを実行する可能性のある呼び出し文の数）を用いたものは Ic と表記する．

3.2.2 キーワードマッチと距離に基づく境界

多くの場合，関心事には対応するキーワードがあり，その関心事を実現するコードのメソッド名など，重要な識別子に用いられている [28]．これらの関心事を実現するコード同士が直接何らかの依存関係を持っているとは限らない [29] が，開発者が注目している要素と直接の依存関係を持つ要素は開発者の作業に関係する要素であることが多い [25]．

そのため，キーワードを含むようなメソッド同士を接続する依存関係の経路を可視化することは，たとえばどのメソッドが他のメソッドを呼び出す役割を担うのか，どのクラスが処理に必要なデータを提供するのか，といった情報を開発者が理解する際に有用である．

このヒューリスティクスは，開発者が指定したキーワードを用いて，以下の手順で各メソッド m についての評価値 $V(m)$ を決定する．$V(m)$ の値の計算式は，キーワードを名前の一部に持つメソッド自身や，またキーワードを名前の一部に持つ複数のメソッドとプログラム依存グラフ上で近い位置にある（依存関係を接続するための経路上にある可能性が高い）メソッドが優先的にスライスに含まれるよう決定した．

- 通常のプログラム依存グラフから，メソッドを頂点とし，呼び出し辺，パラメータ渡し辺，フィールド辺を用いてメソッド間の接続関係グラフを作成する．このグラフ上で，頂点 m_{from} から m_{to} までの最短パスの長さを頂点間の距離 $d(m_{from}, m_{to})$ とする．
- メソッド名の中に部分文字列としてキーワードを含んだメソッド群 $\{m_1, \ldots, m_n\}$ を抽出する．
- ある評価対象のメソッド m に対する評価値 $V(m)$ を，キーワードを含んだメソッド群 $\{m_1, \ldots, m_n\}$ からの距離を用いて，次のように定義する．定数 $\alpha(= 0.8)$ は，評価値の距離に応じた減衰率である．

$$V(m) = \max_{k=1\ldots n} \alpha^{d(m,m_k)} + \operatorname*{avg}_{k=1\ldots n} \alpha^{d(m,m_k)}$$

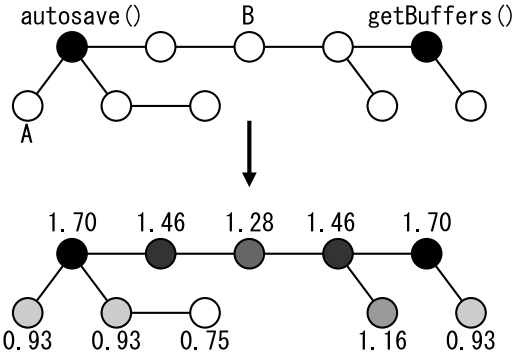

図 3 autosave + buffer に対するキーワードの評価値

図 3 は，あるプログラム依存グラフから得られたメソッド間の接続関係グラフの例である．グラフの各頂点はメソッドであり，上段のグラフでは，autosave と getBuffers という 2 つのメソッドが黒色で示されている．下段のグラフは，上段のグラフに対して，autosave，buffer という 2 つのキーワードを与えたときの評価値 $V(m)$ を表している．

このようにして得られるメソッドごとの評価値と，閾値 th を用いて，$isBarrier(v_{from}, v_{to})$ を次のように定義する．ただし v_{from}, v_{to} はそれぞれメソッド m_{from}, m_{to} に所属する頂点であるとする．

$$isBarrier(v_{from}, v_{to}) = m_{from} \neq m_{to} \wedge (V(m_{from}) < th \vee V(m_{to}) < th)$$

この手法は，キーワードを名前の一部に持つメソッドからの距離に基づいていることから，開発者が grep のようなキーワード検索によって本手法への入力を

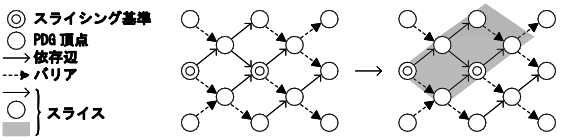

図 4 バリア付きスライシングの実行

決定した場合，Length-Limited Slicing [17] の手法と近い結果が得られるように見える．しかし，本手法は，複数の頂点に対する位置関係によって距離を決定している点，また，メソッド単位を頂点としたグラフ上で距離を計測している点で，Length-Limited Slicing とは異なっている．図 3 の例では，キーワードにマッチしたメソッドからの距離は，頂点 A は 1，頂点 B は 2 と，頂点 A の方が近い．しかし，評価値で見ると，2 つのマッチしたメソッドに挟まれている頂点 B の方が高い値を示している．これによって，Length-Limited Slicing でスライスを小さく絞り込んだときに途切れてしまうような依存関係の経路を，本手法では取得可能である．

このヒューリスティクスは，以降の図表中では K と略記する．

3.3 バリア付きスライシングの実行

このステップでは，入力として与えられたプログラム文の集合（に含まれるすべての変数）をスライシング基準とし，前のステップで得られたバリア集合を通過しない経路で到達可能な頂点の集合をスライスとして抽出する．

図 4 は，バリア付きスライシングの模式図である．スライシング基準は一般に複数の頂点に変換され，それぞれの頂点から，順方向および逆方向のグラフ探索が行われる．スライシングはバリアで停止するため，得られるスライスは図 4 のようにバリアに囲まれた形になる．

具体的なアルゴリズムとしては，Horwitz らの定義 [10] [24] に，バリア計算のアルゴリズム [16] を修正したものを組み合わせている．[16] では，スライシングの途中経路に登場してはならない（ただし，始点もしくは終端としては登場してよい）頂点としてバリアを定義しており，バリアの制約を破らないスライシングアルゴリズムが定義されている．本研究では，このアルゴリズムを，バリアが有向辺の場合に修正して適用している．

3.4 ライブラリ頂点のフィルタリング

このステップでは，得られたスライスからライブラリに属する頂点を取り除く．

ライブラリ（例えば java, javax パッケージ）に含まれるような要素は，それ自体が使われていることは理解する必要があったとしても，その詳細は理解する必要がないことが多い．ライブラリに関する情報を除外ないし詳細を省略することで，スライシングの結果を簡潔な形で開発者に提示する．

現時点では，Java の標準ライブラリに含まれる頂点を単純にスライスから取り除いている．そのため，ライブラリを経由した依存関係は開発者に提示されない．ライブラリを経由した依存関係を明示するためには，頂点を取り除くことで消失する経路を，ライブラリを代替する辺で接続しておく等の工夫が必要だと考えられるが，それは今後の課題である．

3.5 スライスの関心事グラフへの変換

このステップでは，フィルタリング適用後のスライスを，亀田らのルール [13] に基づいて関心事グラフに変換する．

スライスを関心事グラフとして可視化することで，開発者はプログラム要素間の関係を把握することができる．また，対応するソースコードにもスライスの情報をマッピングすることで，関心事グラフと，その詳細としてのソースコードを併用して，関心事を理解していくことができる．

4 実験

本手法の有効性を示すため，Java を対象としたスライシングツール（Slicing based COncern LOCation tool，SCOLOC）を作成した．このツールは，プログラム依存グラフ構築部と，スライシング実行部からなる．

まず，PDG 構築部は Java バイトコードを入力とし

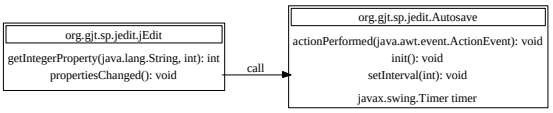

図 5　出力される関心事グラフの例

て受け取り，PDG の構築を行う．この構築部は，Java バイトコードを解析・変換するためのフレームワーク Soot †3 に基づいており，フレームワークが提供する制御フロー解析，データフロー解析，Points-to Set 解析および動的束縛の解析を用いて PDG を構築する．Soot は Points-to Set 解析の実装として Spark, Paddle など複数の実装を提供しているが，本実装では Spark を使用した．また，PDG 構築部は，PDG の各頂点に対して，ソースコードの行番号との対応関係も保存する．

次に，スライシング実行部は，構築された PDG とスライシング基準となるプログラム文（ファイル名と行番号）の集合，ヒューリスティクス用の引数（キーワードと閾値）を入力として受け取り，ヒューリスティクスによるバリアの決定，スライスの計算，そして関心事グラフへの変換を行う．

最終的な出力としては，エディタ上にマーキングを行うための行番号情報と，関心事グラフのテキスト記述，そして関心事グラフを Graphviz †4 によって可視化したものの 3 種類をサポートした．

Autosave 機能の簡単な関心事グラフを Graphviz で可視化した出力例を図 5 に示す．Graphviz による可視化では，図の複雑化を避けるためクラスを 1 ノードとして表現した．メソッドやフィールドはクラスノードの内部に含め，クラス間を接続する辺はそのまま表示し，クラス内部の辺を省略している．

スライシング基準は，本来，各ソースコード行の変数を指定するものであるが，本研究では，キーワード検索やその他の Feature Location ツールとの連携を容易にするため，プログラム文の行番号が指定されればその行中のすべての変数を，また，メソッド名が指定された場合はそのメソッドに含まれるすべての変数の出現を，スライシング基準として扱うようにしている．

†3 Soot, a Java Optimization Framework http://www.sable.mcgill.ca/soot/

†4 Graphviz http://www.graphviz.org/

表 3　手作業で作成した関心事グラフとスライシング基準のサイズ

関心事		関心事グラフ		スライシング基準	
		クラス	メソッド	クラス	メソッド
Autosave	1	7	18	4	8
	2	12	48		
Marker	1	8	30	3	13
	2	12	33		
Undo	1	3	6	5	6
	2	12	21		
Batch	1	20	32	2	4
	2	7	14		
Input	1	9	11	3	8
	2	4	7		
MethodID	1	5	13	4	4
	2	7	9		

4.1　実験対象ソフトウェア

実験対象には， jEdit 4.2 final および我々の開発したツール SCOLOC を選択した．

jEdit は Java で書かれたテキストエディタで，プラグインを除くクラス数は 777 クラス，行数は 140,316 行である．また，スライシングツールのサイズはクラス数 183，行数 16,144 である．いずれの解析でも，Sun JDK 1.4 に含まれるライブラリを含めた解析を行っている．

本手法は，保守作業におけるプログラムの変更を想定し，jEdit および SCOLOC についてそれぞれ 3 個，変更作業の説明という形で機能的関心事を記述した．

あるメソッドやフィールドが関心事に含まれるかどうかは開発者によっても判断が分かれるため，被験者として，大学院生 2 名に関心事グラフの構築を依頼した．この 2 名は，いずれも Java プログラムの開発経験があり，また jEdit のコードを過去に読んだこと

がある．スライシングに関する用語などの基礎知識は持っているが，本手法および SCOLOC ツールの開発には関与していない．被験者 2 名は，与えられた変更作業の説明を元に，変更作業の事前計画として関心事グラフを構築した．作成した関心事グラフのサイズを表 3 に示す．著者らは，各関心事に対応するキーワードを用意し，主として grep によるキーワード検索の結果を用いてスライシング基準を作成した．以下，作成した計 6 個の関心事を示す．

Autosave (jEdit) ユーザが編集中のバッファの内容を，特別な名前のファイルに定期的に自動保存する機能である．自動保存先のファイル名を変更することを作業目的とした．
ツールへの入力には，キーワード “autosave” と，同キーワードを引数として grep を実行した結果得られたコード集合を用いた．

Marker (jEdit) jEdit は，エディタ領域の左端（ガター）部分を右クリックすることで，マーカーを設置できる．このマーカーを設置する際に，マーカーの意味などの任意の文字列を追加で記入，表示する機能を追加することを作業目的とした．
ツールへの入力であるキーワードには “marker” を用いた．また，スライシング基準としては，jedit.Buffer クラス等にコメントとして “marker manipulation methods” と記載されていたメソッド群（のすべての行）を指定した．

Undo (jEdit) Undo 機能では，バッファに対する編集を取り消すことができるが，連続した変更を元に戻すための作業ヒストリ長に 0 を設定してしまうと，テキストの編集を行うごとに，配列の不正アクセス（ArrayIndexOutOfBoundsException）が発生してしまう．この問題を修正することを作業目的とした．
ツールへの入力であるキーワードには，“undo” を使用した．また，スライシング基準には，送出された例外のスタックトレース情報（クラス名と行番号のペアの列）を使用した．

Batch (SCOLOC) 複数のスライシングを連続して実行するバッチ処理機能では，スライシングに必要なアルゴリズムに対応するオブジェクトを毎回作り直している．これらを適切に初期化し，再利用する処理を追加実装することを作業目的とした．
オブジェクトの生成に Factory パターンを用いていたことから，キーワード “create” と，該当する Factory パターンのメソッド群をツールへの入力とした．

Input (SCOLOC) SCOLOC は，スライシング基準を，コマンドライン引数や CSV ファイルによって，ファイル名と行番号，あるいはクラス名とメソッド名のリストとして受け取っている．この形式として，クラス名やメソッド名にワイルドカード文字列を許すよう機能を拡張することを作業目的とした．
ツールへの入力には，スライシング基準のコード上での表現であったキーワード “criterion” と，同キーワードを引数として grep を実行した結果得られたコード集合を使用した．

Method ID (SCOLOC) SCOLOC は，解析対象プログラムの各メソッドに ID を割り当て，そのリストをファイルに保存する．しかし，オブジェクト共有の取り扱いに誤りがあり，1 度の実行でファイルが 2 回保存されていた．この誤りを修正することを作業目的とした．
ツールへの入力には，キーワード “methodid” と，同キーワードを引数として grep を実行した結果得られたコード集合を使用した．

厳密に時間計測を行ったわけではないが，手作業での関心事グラフの構築には，1 つのグラフにつき 1 時間から 1 時間 30 分の時間が必要であった．

ツールの実行環境には Windows Vista 64 ビット版，Intel Core 2 Duo 6300 (1.86GHz)，64 ビット版 Sun JVM 1.6 を用いており，4GB のメモリを JVM に割り当てた．PDG 構築には，jEdit，SCOLOC のいずれの場合でも約 20 分の時間を要した．この構築時間のうち約 10 分は Points-to Set 解析とコールグラフの構築に費やされており，残り 10 分がメソッド単位での解析とファイルへの出力である．スライシ

表 4　Autosave に対するスライスのサイズ

	頂点数	メソッド数	クラス数
基準	290	8	4
通常スライス	492,300	5,442	724
Length 10	459,132	5,295	721
Length 9	418,546	5,178	720
Length 8	360,242	4,871	717
Length 7	278,430	4,351	688
Length 6	154,320	3,552	647
Length 5	76,798	2,399	588
Length 4	30,278	1,324	429
Length 3	8,572	698	295
Length 2	3,489	171	71
Length 1	672	63	19
K 0.80	483,568	4,946	724
K 0.90	479,578	4,829	724
K 1.00	270,838	1,853	580
K 1.10	233,582	1,580	548
K 1.20	56,007	296	135
K 1.30	11,002	54	20
K 1.40	3,873	24	9
K 1.50	389	9	4
K 1.60	363	8	4
Ia auto	397,306	4,652	654
Ia 8	284,346	4,157	644
Ia 4	109,727	2,179	485
Ia 2	1,751	75	31
Ia 1	845	20	6
Ic auto	399,570	4,688	657
Ic 8	326,504	4,418	653
Ic 4	247,767	3,817	626
Ic 2	97,259	2,160	503
Ic 1	40,472	1,153	378

ングには，実行の前準備に約 5 分程度，1 回のスライシングとそれに伴う関心事グラフの生成に 10 秒から 1 分程度の時間を要した．2 つの対象ソフトウェアで時間に差がなかったのは，同時に解析した Sun JDK 1.4 のクラスが占める割合が大きいためであると思われる．また，スライシングに要する時間にはかなりの幅があるが，グラフ探索が早く打ち切られ，結果が小さなスライスとなったときほど，計算時間は短くなった．

4.2　スライスのサイズ

本実験では 6 つのタスクに対して関心事グラフを作成したが，これらは，高々 48 のメソッドしか含んでいない．このサイズを大きく超えたスライスの抽出は，出力の品質（次節で行う適合率による評価結果）を著しく悪化させ，また，たとえば Graphviz による可視化手法では対応できないといった問題を生じる．

表 4 に，関心事 Autosave を発見するために用意したスライシング基準を用いて，従来のプログラムスライシング手法（バリアなし），Length-Limited Slicing，そして提案手法で得られたスライスのサイズを示す．本手法では，スライシングの結果を関心事グラフへ変換して開発者に提示するため，スライスに含まれるメソッド数が，最終的に出力される関心事グラフのサイズへと影響する．

表 4 における「通常スライス」は，Horwitz らのアルゴリズムの適用結果である．Length k とあるのは，スライシング基準である頂点から依存辺を高々 k 回たどってたどり着ける頂点だけを抽出したスライスである．$k = 0$ の場合，出力はスライシング基準と等しい．Length-Limited Slicing では，$k = 1$ の場合にのみ小さいスライスを抽出することに成功しているが，$k = 2$ 以上になると，多数のメソッドをスライスとして抽出している．

K 1.00 は，キーワードを用いたヒューリスティクスによるスライシングで，閾値として 1.00 を与えたことを表している．提案手法では，抽出されるスライスのサイズが，閾値によって従来のスライスとほぼ同等のものから，スライシング基準周辺のみの最小限のものにまで変化する．

入次数のヒューリスティクスでは，手動で設定したいくつかの閾値と，自動設定（auto）の閾値 $\lfloor\sqrt{N}\rfloor$ を用いた．自動設定の値における N は，プログラム依存グラフの頂点数である．

閾値が小さな値の範囲では，閾値が 1 変化するだけでスライスのサイズが大きく変化する．これは，入次数がべき乗則に従っており，入次数の小さい頂点の数が非常に多いためである．また，過剰に依存関係が集中したプログラム要素を除外する目的から考えても，あまり小さな閾値は有効ではないと考えられる．したがって，入次数によるヒューリスティクスは単独で使うのではなく，他のヒューリスティクスと併せて使うべきだと考えられる．

4.3 ヒューリスティクスの効果

提案手法と Length-Limited Slicing [17] で得られる関心事グラフを，被験者が手作業によって作成した関心事グラフと比較した．

この評価基準には，適合率 (Precision)，再現率 (Recall)，そしてそれらのトレードオフを示す F **値**を用いた．これらの数値は，プログラムスライスに含まれるメソッドの集合 M_{slice}，人間が作成した関心事グラフに含まれるメソッドの集合 $M_{concern}$ に対して，以下の式で計算される．

$$Precision = \frac{|M_{slice} \cap M_{concern}|}{|M_{slice}|}$$
$$Recall = \frac{|M_{slice} \cap M_{concern}|}{|M_{concern}|}$$
$$F = \frac{2Precision \times Recall}{Precision + Recall}$$

この定義では，メソッド集合 M の要素数を $|M|$ と表記している．$|M_{concern}|$ の具体的な値は，表 3 の「関心事グラフ」列に示しているメソッド数となる．結果の F 値が高いほど，抽出されたメソッド集合 M_{slice} は人間が選択したメソッド集合 $M_{concern}$ に近い．

このように定義した F 値を用いて，我々が提案した入次数およびキーワードマッチのヒューリスティクス，Length-Limited Slicing を以下のように組み合わせたスライシング計算を行い，得られた関心事グラフの比較を行った．

1. Length-Limited Slicing のみ
2. キーワードマッチのみ
3. Length-Limited Slicing + 入次数
4. キーワードマッチ + 入次数

このうち，1 が従来手法であり，4 が提案手法を完全に適用したものである．キーワードマッチと Length-Limited Slicing は，いずれも距離を用いていることから，同時には適用していない．

まず，入次数のヒューリスティクスについて，他の 2 つの手法との組み合わせに適した閾値の設定を検討した．具体的には，入次数のヒューリスティクスの閾値を変化させて F 値の計算を行い，使用しない場合と比較した．図 6 は，関心事 Autosave に対して，異なる入次数の閾値を用いて，キーワードマッチのヒューリスティクスと組み合わせて適用した場合の F 値である．`w/o I` の値は，キーワードマッチ単独の場合の値である．この結果から，ヒューリスティクスの組み合わせによって F 値が向上しており，Ia 2 の値が最も良かった．この図は 1 人の被験者が作った Autosave に関するメソッド集合に対しての F 値を表しているが，他のデータに対しても確認した結果，この Ia 2 の設定を以降の実験に用いることにした．

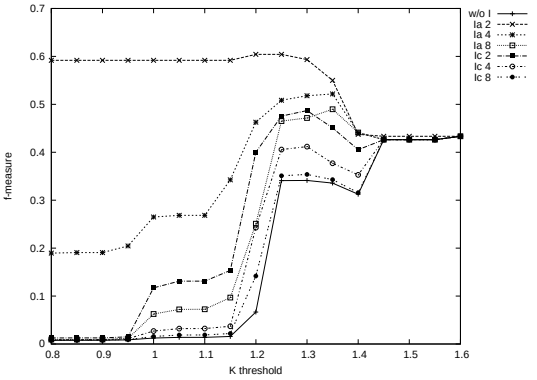

図 6 Autosave (被験者 1) でのヒューリスティクスの組み合わせによる F 値の変化

Ia は，Ic と異なり，メソッド呼び出し辺だけでなく，多数のメソッドから代入される，グローバル変数に近いフィールドによるデータ依存関係もバリアとして検出する．Ic との差の原因については今後も調査が必要であるが，ある 1 つの機能の内部ではデータの多くを引数によって伝達しており，異なる機能間ではフィールドに一度データを保存してデータを受け渡す形式が多かったという可能性がある．

続いて，従来手法である Length-Limited Slicing を用いて，たどる依存辺の数 k を 1 から 10 まで変化させて得られた関心事グラフの F 値を図 7 に示す．スライシング基準となる頂点はキーワード検索で選択しているため，$k = 1$ または 2 のとき，良い結果を出すことができる．一方で，4.2 節で示したように，k が増加するとスライスサイズに含まれるメソッド数が大きく増加し，関心事グラフのサイズを大幅に上回るため，適合率と F 値を悪化させる．

キーワードマッチによるヒューリスティクスのみを適用した結果を図 8 に示す．サイズが最も小さなスラ

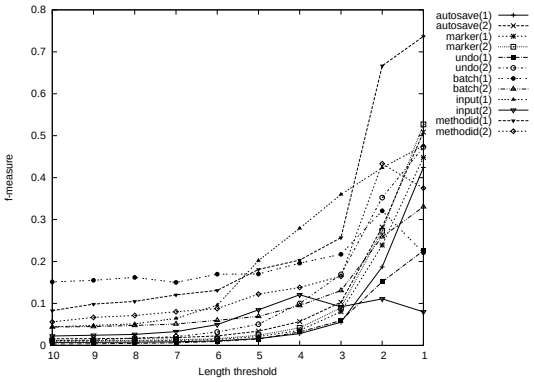

図 7　Length-Limited Slicing の結果

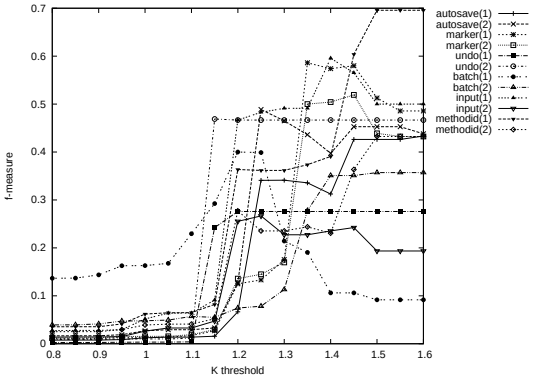

図 8　キーワードマッチのヒューリスティクスのみの適用結果

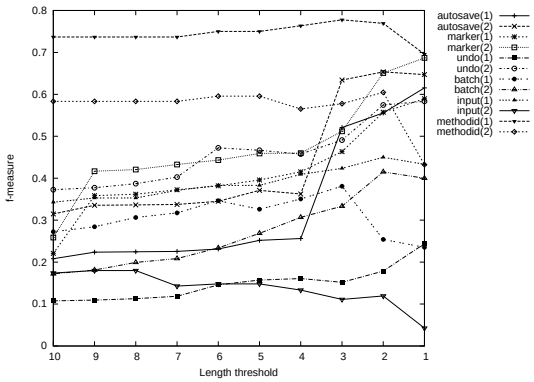

図 9　Length-Limited Slicing に入次数 (Ia 2) を組み合わせた結果

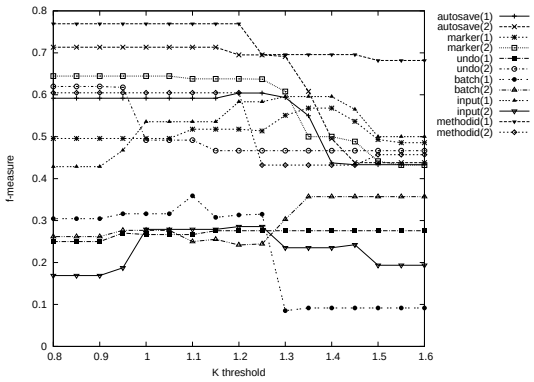

図 10　キーワードマッチと入次数 (Ia 2) を組み合わせた結果

イスは，スライシング基準として選ばれたコードを含んだクラスのみの関係を示す非常に簡潔な関心事グラフとなり，再現率が低いため，F 値としては高くならない．閾値の低下に伴ってスライスサイズが増加すると，再現率および F 値が上昇するが，スライスのサイズがある程度以上大きくなると，適合率の減少によって F 値も減少に転じる．

$k = 1$ で適用した Length-Limited Slicing は，キーワードマッチによるヒューリスティクスの結果を上回る良好な結果を出すことがある．しかし，$k = 1$ とは，スライシング基準として選んだプログラム文からただ 1 度依存辺を辿っただけであり，メソッド呼び出し関係やフィールドによる直接的な依存関係を持つメソッドだけを抽出したものとなっている．キーワードマッチのヒューリスティクスで閾値を高く設定した場合（たとえば 1.4 などを選んだ場合）は，F 値では Length-Limited Slicing とほぼ同じ値を取るが，Length-Limited Slicing では $k \geq 2$ でなければ登場しないようなメソッドをスライスに含めることができる．これに該当するメソッドの例としては，関心事 Autosave において，変更された設定を Autosave クラスに伝える経路である jEdit.propertiesChanged メソッドが挙げられる．このような差は，我々の用いたヒューリスティクスが，キーワードにマッチした（スライシング基準としても用いている）メソッド間を接続するような頂点を優先的に抽出しているためであると考えられる．

図 9 は Length-Limited Slicing に対して，また図 10 はキーワードマッチのヒューリスティクスに対し

て，それぞれ入次数のヒューリスティクス（Ia 2）を組み合わせたときの結果である．入次数のヒューリスティクスの導入によって，いずれの結果も，導入前に比べ，閾値の変化に対して F 値の悪化が抑制されているが，Length-Limited Slicing が k の値から受ける影響などには変化がなかった．

図 7 で示した従来の Length-Limited Slicing 手法を用いた場合に比べ，提案手法である入次数とキーワードマッチのヒューリスティクスは，人間が作成した関心事グラフに近い結果を抽出することに成功している．我々の提案手法の一部である入次数のヒューリスティクスだけを Length-Limited Slicing に組み合わせた状態でも，提案手法とほぼ同程度の F 値となることがある．ただし，提案手法は，複数のキーワードにマッチしたメソッド間を接続するような経路を優先的に発見しており，たとえば Length-Limited Slicing よりサイズが小さいとき（例：表 4 における K=1.3 以上）でも，距離 $k = 2,3$ となるようなメソッドをスライスに含めている．距離が遠いメソッドほど数が多いため，そのようなメソッドを自動で特定できる点で，提案手法は有効である．

最終的な関心事グラフの要素数が大きくなってきたときは，提案手法のほうが F 値が高いグラフを提示できるが，やはり本研究で用いているような関心事グラフの直接の可視化は困難となる．しかし，関心事を文書化する手法 [19] やコードを読解する順序の構築 [23] のように，他のプログラム理解手法を適用するための情報として利用可能となる可能性があり，Length-Limited Slicing を用いた場合よりも適用可能な場面が多いと考えられる．

4.4 議論

4.4.1 関心事グラフの出力例

提案手法を完全に適用した（キーワードマッチと入次数の両方を用いた）スライシングによって得られた F 値の高い（手作業で作成した関心事グラフと近い）関心事グラフの例として，図 11 を示す．図 11 は Autosave についての関心事グラフで，含まれていたクラスは 17 個あった．著者らが実際に jEdit における Autosave 機能を改変し，自動保存で作成されるファイル名を変更しようとしたところ，図中に太枠で示す 9 個のクラス群が実際に重要であると判明した．この関心事グラフでは，従来のスライシングでは含まれてしまっていた UndoManager クラスや，オプション設定値を管理するデータ構造の実装である PropertyManager クラスなどが除去されている．また，jEdit クラスは 90 のメソッドを持つ約 3800 行の巨大クラスであり，Buffer クラスも約 4300 行，120 以上の GUI 処理のメソッドを持っているが，これらのメソッド群のほとんどをスライスから除外することに成功している．

関心事 Autosave では，ユーザが自動保存の間隔などを設定するための設定ダイアログである AutosaveBackupOptionPane に関する処理と，実際に保存処理を定期的に実行する Autosave.actionPerformed メソッドとの間には，直接のメソッド呼び出し関係が存在しない．フィールドに保存された値を経由したデータ依存関係を用いてクラス間の関係を探索できる点が，プログラムスライシングを用いた手法の利点である．

従来手法である Length-Limited Slicing によって得られる最も F 値の高かったグラフは，$k = 1$ の場合に得られる 19 個のクラスであった．サイズは提案手法で得られたものとほぼ同じであったが，頂点同士を接続するような経路が発見されておらず，関心事グラフが 8 つの部分グラフに分割された（ある部分グラフからは他の部分グラフに到達することができない）状態となっていた．提案手法で得られた関心事グラフは，頂点が連結されており，従来手法で得られたものと比べて，メソッド間の関連を知ることができるという点で有用なものとなっていた．

4.4.2 手作業で作成した関心事グラフとの違い

被験者 2 名に対して，提案手法で抽出された関心事グラフおよび被験者らが作成した関心事グラフとの差分を提示し，提案手法の有用性についてインタビューを行った．

その結果，まず，提案手法によって抽出された関心事グラフは，被験者らが発見したにも関わらず関心事グラフに入れ忘れたメソッドを発見していたことが判明した．また，動的束縛の解析によって実際には呼ば

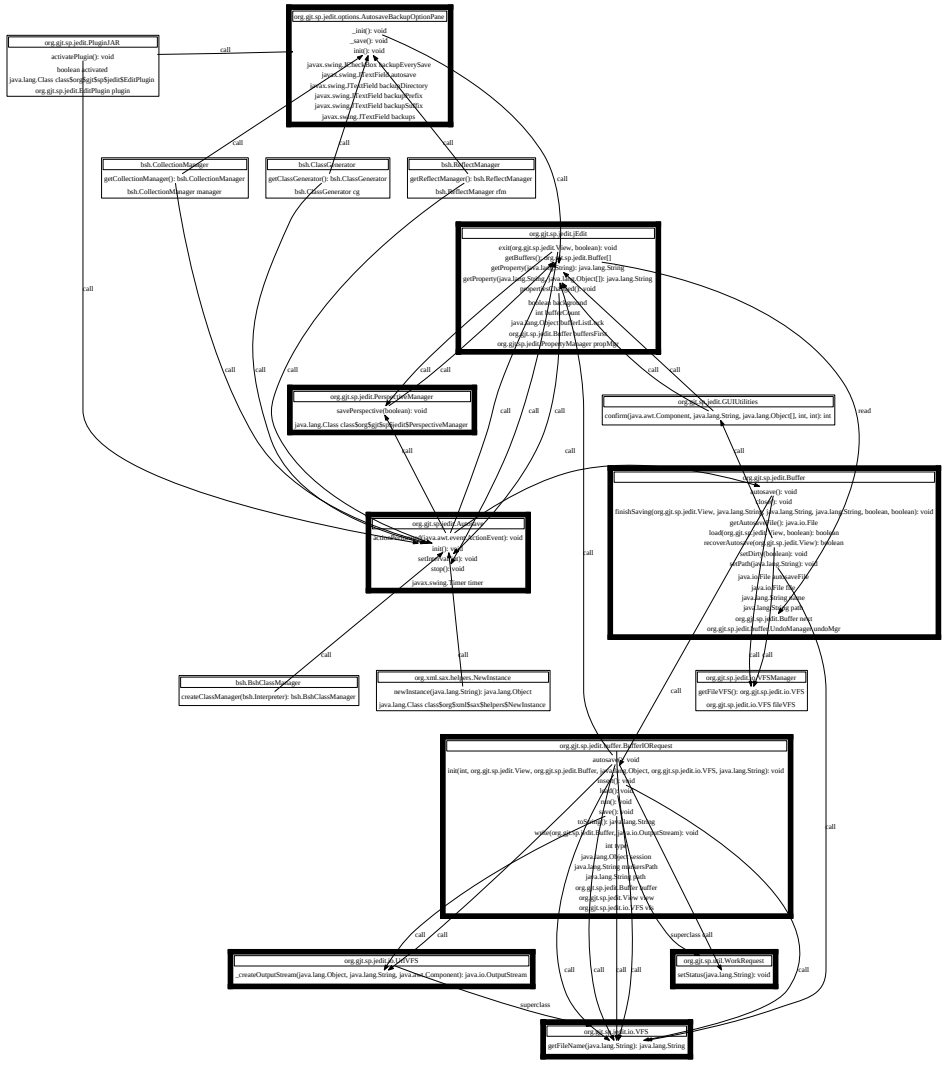

図 11　関心事理解に適した関心事グラフ（Autosave）

れないようなメソッドが除去されている点，フィールドを経由した推移的なデータフロー関係の追跡が容易である点が有用であるとの回答を得た．

提案手法は grep によるキーワード検索の結果などをそのまま関心事グラフに変換できるため，得られたグラフを手作業での洗練していくか，あるいは適切にスライシング基準を追加していくといった対話的なインタフェースがあれば，より有用であろうとの意見が得られた．

一方で，オブジェクトの振る舞いに関する情報（たとえば Autosave クラスが Timer に対する Event Listener として登録されること）などは関心事グラフ上に提示されないとの指摘もあった．これに対しては，たとえば実行履歴の可視化手法 [31] と組み合わせ，構造と振る舞いの要約を同時に開発者が閲覧できるようにするといったツールの拡張が考えられる．

4.4.3　入次数によるユーティリティクラスの判定

入次数のヒューリスティクスは，ソースコードの多数の場所に出現するメソッド呼び出しを，探索の範囲から除去する効果がある．一方で，わずかな回数しか参照されないクラスは，たとえライブラリであっても結果に含まれる．本手法では，出力を簡略化するた

め，Java 標準のクラスについては，パッケージ名による出力のフィルタリングを行っている．

本実験では，ユーティリティと判断されたが，入次数が少なく，かつ関心事の理解に役立つクラスとして，たとえば Autosave を実装するために使われていた Timer や，Input を実装するために使われていた CSVReader というクラスが発見された．

これらのクラスは，それぞれの機能の実装でのみ利用されているので，入次数が少ない．また，実装の方法を端的に示しているため，関心事の理解として有用であると判断された．Timer は Java 標準クラスであったのでフィルタリングの対象であったが，CSVReader は SCOLOC のプログラムの一部としてソースコードごと再利用されていたため，呼び出し関係などが関心事グラフ中に示された．

入次数の少ないもののうち，実際に不要であると判明したものは手動でフィルタリングを追加設定することができる．また，逆に適用済みのフィルタを解除することも可能である．フィルタリングは，スライシング結果に対して適用するため，プログラムスライスさえ保存しておけば，関心事グラフをただちに更新することができる．

その一方で，ライブラリやフレームワークの中から重要クラスだけを的確に取りだす手法は，今後の課題である．動的解析では，あるユーザの操作に対応する実行履歴の中から，その操作に固有の重要なメソッドだけを抽出する方法が提案されており [3]，このような「機能に対して固有である」性質を入次数よりも適切に反映するメトリクスがあれば，それをヒューリスティクスとして本手法の枠組みに導入することが可能である．

4.4.4 ライブラリの解析

本論文で用いた実装では，ライブラリを解析対象に含めている．Java のライブラリは，イベントハンドラやオブジェクトを比較するためのメソッド（Object.equals）を内部で使用することが多く，これらの呼び出し関係がヒューリスティクスに影響を及ぼす可能性がある．

本実験では，すべてのメソッドを解析した状態での実験を行っている．実装に用いた Soot フレームワークは，Java 標準のクラスに付属する native メソッドに関する「事前に準備された」依存関係を提供しており，ユーザが追加した native メソッド以外のすべてのメソッドが解析対象となっている．

本実験で，このようなライブラリ内部の依存関係の影響を受けていたのは，関心事 Autosave に用いられた Timer オブジェクトから Autosave.actionPerformed メソッドへのコールバックなど，イベントハンドラに関する呼び出しであった．

ライブラリを除外した安全な解析を実現できれば提案手法の実行性能の改善が見込めるが，そのためには，ライブラリ内部のデータ依存関係を近似する方法の考案が必要である．また，コールグラフの頂点数などが変化することから，得られるスライスや関心事グラフへの影響も評価していく必要があるが，これらは今後の課題である．

4.4.5 閾値の設定方法

実際の運用においては，開発者らは事前に定義された正解集合を持たないため，F 値を用いて閾値の値を決定することはできない．本実験において被験者が作成したグラフのサイズが表 3 に示した最大のもので 48 個のメソッドであったこと，また，Robillard が行った実験においても jEdit に対する関心事グラフが 35 個のコード片から構築されていたこと [26] などから，開発者が作成する関心事グラフの情報を集めていけば，出力として適切だと思われる関心事グラフのサイズを自動推定できる可能性がある．

しかし，一般的な関心事グラフのサイズというのは知られていないため，現段階での利用方法としては，関心事グラフの段階的な提示を考えている．これは，まず閾値を自動で最も厳しい値に設定することでスライシング基準となったメソッド群を含んだ簡潔な関心事グラフを提示しておき，閾値を段階的に変化させ，より詳細な情報を含んだ関心事グラフへと理解を進めていくという方法である．この方針は，Storey らの提案する「ソフトウェア理解支援ツールが満たすべき項目」の 1 つ「E5: システム（本研究の場合，関心事の実装）の構造を表すオーバービューが抽象度を変化させながら使えること」とも合致する [30]．また，バリアを用いたスライシングのアルゴリズム [16] は，

バリアを通過しない経路を網羅的に探索していく反復アルゴリズムとして定義されており，バリアが除去された経路だけを追加で探索することが可能である．そのため，最も厳しい閾値から順に複数の関心事グラフを抽出する方法は，ツールの性能面での問題も少ないと考えている．

5 関連研究

ソフトウェア中の特定の機能や関心事に属するソースコードを発見，調査するための手法は，広く研究されている．

SNIAFL は，メソッド群から，そのメソッドに関係する機能への対応関係を抽出する情報検索手法である [37]．この手法は，各機能の実行の様子を表現する擬似実行トレースを出力としている．我々の手法は，このような手法によって得られた機能の静的な構造を調査するのに有益である．

我々の手法に近いものとして，Chen らの依存グラフを用いたアプローチがある [4]．Chen らの手法は，スライシング基準からの依存グラフを作成した上で，ユーザが調査したい頂点の選択と，検索用の依存グラフの更新を繰り返す対話的な環境を提案している．我々の手法は，スライシング基準の選択以降の作業をヒューリスティクスによって自動化している点で異なっている．

Krinke は，スライシング基準からの距離を制限した Length-Limited Slicing を提案している [17]．これは，我々の手法におけるヒューリスティクスの代わりに，プログラム依存グラフ上の距離を用いたものだと考えることができる．我々の手法は，グラフ上の距離ではなく，キーワードを含む複数のメソッドからの距離や，入次数を用いた基準によって訪問を停止するヒューリスティクスを用いることで，関心事に該当するコードを抽出している．

Shepherd らは，ソースコードに対する自然言語解析に基づいた関心事の抽出を提案している [28]．この手法は，あるメソッドが所属する関心事を，識別子やコメント中に登場する単語から取り出せる動詞とその目的語になる名詞のペアであると定義している．この手法により開発者は，実装がソースコード上で分散しているような関心事の抽出が行える．しかし，出力の可視化については，抽出されたメソッド間の直接の呼び出し関係と継承関係以外はサポートしていない．この手法によって得られたメソッド群を我々の提案手法の入力とすれば，より有効な情報を開発者に提示できると推測される．

我々の提案手法によって得られるグラフの可視化において，コードが多数のメソッドに分散すると，グラフが巨大化し，可読性を損なう可能性がある．このようにコードが著しく分散する例としては，エラー処理やロギングなどの横断的関心事があり，100 以上のメソッドに分散する事例などが報告されている [2]．このような横断的関心事に対しては，複数の類似したコード断片を簡潔な 1 つのコードとして提示，保守を行う Fluid AOP [9] のアプローチが有効であると考えられる．また，Seesoft [5] のように，1 つのファイルを 1 つのファイルの長さに比例した長方形で表示し，関心事に対応する行に相当する部分を着色して表示することでプログラム全体におけるコード分布を表示する手法なども利用可能である．

ソースコードだけでなく実行履歴情報を用いたアプローチとして，Walkinshaw らの手法がある [34]．この手法は，特定のユースケースやシナリオでプログラムを動作させ，その実行履歴情報から構築したコールグラフのうち，重要だと思われる部分グラフを抽出する．入力として，開発者が調べたいメソッド群（landmark methods）を指定させ，その間のパスを含むコールグラフを抽出する．また，実行履歴情報を用いるもう 1 つの手法として，Eisenbarth らによる Formal Concept Analysis を用いた手法がある [6]．この手法は，実行履歴ごとにどの機能を実行したかというキーワードを設定することで，特定のキーワードに対応する手続きの発見を行う．これらの動的解析手法は，機能に対応するコードを特定するために多数の入力と実行履歴の観測が必要であることから，我々は静的解析のアプローチを採用した．

Kersten らが開発している開発支援ツール Mylar は，統合開発環境において開発者が頻繁に参照しているプログラム要素群を関心事とみなす DOI モデルを用いている [14]．Mylar の DOI モデルは，開発者

の作業内容に従って開発環境上での表示状態（強調，非表示）を変更することで，開発者の効率的な作業を支援する．我々の提案手法の出力で得られた関心事グラフに対応するファイル集合と，Mylar によって得られたファイル集合とを比較することで，開発者が存在に気づいていないファイルの再発見や，提案手法の出力には含まれていたが実際には不要な要素の除去などを行うことができる．

6 まとめ

機能的関心事を実現するプログラム要素を発見し，それらの間の相互作用を理解することは，ソフトウェアの保守作業において非常に大きな割合を占めている．

我々は，開発者が着目した関心事について，手がかりとなるプログラム文の集合とキーワードをもとに関心事グラフを抽出する手法を提案した．本手法は，プログラムスライシングに基づいているが，プログラム依存グラフ上での辺の入次数，キーワードを含んだメソッドからの距離を用いたヒューリスティクスによって，関心事に含まれると推測されるプログラム文だけをスライスとして出力している．

提案手法に基づくスライシングツールを実装し，jEdit および提案手法を実装したツールを対象とした適用実験を行った．その結果，従来のプログラムスライシング手法に基づく手法と比べ，人間が作成したものにより近い関心事グラフを抽出していることを確認した．

今後の課題としては，対話的に興味あるメソッドやフィールドを追加していくことで関心事グラフを構築していく環境の実現が挙げられる．また，メソッド単位で頂点を作成した依存グラフを用いたプログラムスライスの近似解計算 [1] や，コールグラフだけを用いた計算など，スライシング計算を簡略化した場合での実行時性能および出力結果のトレードオフを評価する予定である．

謝辞 有益なご助言をいただきました University of British Columbia, Gail Murphy 教授および匿名の PPL2007 査読者に心より深く感謝いたします．

本研究は，日本学術振興会科学研究費補助金若手研究（スタートアップ）（課題番号:19800021）の助成を得た．

参 考 文 献

[1] Binkley, D., Gold, N., and Harman, M.: An Empirical Study of Static Program Slice Size, *ACM Transactions on Software Engineering Methodology*, Vol. 16,No. 2(2007), pp. 8.

[2] Bruntink, M., van Deursen, A., van Engelen, R., and Tourwé, T.: On the Use of Clone Detection for Identifying Crosscutting Concern Code, *IEEE Transactions on Software Engineering*, Vol. 31,No. 10(2005), pp. 804–818.

[3] Chan, K., Liang, Z. C. L., and Michail, A.: Design Recovery of Interactive Graphical Applications, *Proceedings of the 25th International Conference on Software Engineering*, 2003, pp. 114–124.

[4] Chen, K. and Rajlich, V.: Case Study of Feature Location Using Dependence Graph, *Proceedings of the 8th International Workshop on Program Comprehension*, 2000, pp. 241–247.

[5] Eick, S. G., Steffen, J. L., and Jr., E. E. S.: Seesoft-A Tool for Visualizing Line Oriented Software Statistics., *IEEE Transactions on Software Engineering*, Vol. 18,No. 11(1992), pp. 957–968.

[6] Eisenbarth, T., Koschke, R., and Simon, D.: Locating Features in Source Code, *IEEE Transactions on Software Engineering*, Vol. 29,No. 3(2003), pp. 210–224.

[7] Fjeldstad, R. K. and Hamlen, W. T.: Application Program Maintenance Study: Report to Our Respondents, *Proceedings of GUIDE 48*, April 1983.

[8] Hamou-Lhadj, A. and Lethbridge, T. C.: Reasoning about the Concept of Utilities, *Proceedings of the Workshop on Practical Problems of Programming in the Large*, 2004.

[9] Hon, T. and Kiczales, G.: Fluid AOP Join Point Models, *Proceedings of the 2nd Asian Workshop on Aspect-Oriented Software Development*, 2006, pp. 14–17.

[10] Horwitz, S., Reps, T., and Binkley, D.: Interprocedural Slicing Using Dependence Graphs, *ACM Transactions on Programming Languages and Systems*, Vol. 12,No. 1(1990), pp. 26–60.

[11] Inoue, K., Yokomori, R., Yamamoto, T., Matsushita, M., and Kusumoto, S.: Ranking Significance of Software Components Based on Use Relations, *IEEE Transactions on Software Engineering*, Vol. 31,No. 3(2005), pp. 213–225.

[12] Jackson, D. and Rollins, E. J.: A New Model of Program Dependences for Reverse Engineering, *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1994, pp. 2–10.

[13] 亀田大輔, 滝本宗宏: プログラムスライシングに基づく関心事グラフ構築, 情報処理学会論文誌: プログラミング, Vol. 46,No. SIG 11 (PRO 26)(2005), pp. 45–56.

[14] Kersten, M. and Murphy, G. C.: Mylar: a Degree-of-Interest Model for IDEs, *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, 2005, pp. 159–168.

[15] Komondoor, R. and Horwitz, S.: Effective, Automatic Procedure Extraction, *Proceedings of the 11th International Workshop on Program Comprehension*, 2003, pp. 33–42.

[16] Krinke, J.: Slicing, Chopping, and Path Conditions with Barriers, *Software Quality Journal*, Vol. 12,No. 4(2004), pp. 339–360.

[17] Krinke, J.: Visualization of Program Dependence and Slices, *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 2004, pp. 168–177.

[18] Lhotak, O. and Hendren, L.: Context-Sensitive Points-to Analysis: Is It Worth It?, *Proceedings of the 15th International Conference on Compiler Construction*, 2006, pp. 47–64.

[19] Marin, M., Moonen, L., and van Deursen, A.: SoQueT: Query-Based Documentation of Crosscutting Concerns, *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 758–761.

[20] Marin, M., van Deursen, A., and Moonen, L.: Identifying Aspects using Fan-in Analysis, *Proceedings of the 11th Working Conference on Reverse Engineering*, 2004, pp. 132–141.

[21] Murphy, G. C., Kersten, M., Robillard, M. P., and Cubranic, D.: The Emergent Structure of Development Tasks, *Proceedings of the 19th European Conference on Object-Oriented Programming*, LNCS, Vol. 3586, 2005, pp. 33–48.

[22] Myers, C. R.: Software Systems as Complex Networks: Structure, Function, and Evolvability of Software Collaboration Graphs, *Physical Review E*, Vol. 68,No. 4(2003), pp. 046116.

[23] Oezbek, C. and Prechelt, L.: JTourBus: Simplifying Program Understanding by Documentation that Provides Tours Through the Source Code, *Proceedings of the 23rd International Conference on Software Maintenance*, 2007, pp. 64–73.

[24] Reps, T., Horwitz, S., Sagiv, M., and Rosay, G.: Speeding up Slicing, *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1994, pp. 11–20.

[25] Robillard, M. P.: Automatic Generation of Suggestions for Program Investigation, *Proceedings of the 13th Symposium on Foundations of Software Engineering*, 2005, pp. 11–20.

[26] Robillard, M. P.: Tracking Concerns in Evolving Source Code: An Empirical Study, *Proceedings of the 22nd International Conference on Software Maintenance*, 2006, pp. 479–482.

[27] Robillard, M. P. and Murphy, G. C.: Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies, *Proceedings of the 24th International Conference on Software Engineering*, 2002, pp. 406–416.

[28] Shepherd, D., Fry, Z. P., Gibson, E., Pollock, L., and Vijay-Shanker, K.: Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns, *Proceedings of the 6th International Conference on Aspect Oriented Software Development*, 2007, pp. 212–224.

[29] Shepherd, D., Pollock, L., and Tourwé, T.: Using Language Clues to Discover Crosscutting Concerns, *Proceedings of the Workshop on Modeling and Analysis of Concerns in Software*, 2005, pp. 1–6.

[30] Storey, M.-A. D., Fracchia, F. D., and Müller, H. A.: Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration, *The Journal of Systems and Software*, Vol. 44,No. 3(1999), pp. 171–185.

[31] 谷口考治, 石尾隆, 神谷年洋, 楠本真二, 井上克郎: プログラム実行履歴からの簡潔なシーケンス図の生成手法, コンピュータソフトウェア, Vol. 24,No. 3(2007), pp. 153–169.

[32] Vallee-Rai, R. and Hendren, L. J.: Jimple: Simplifying Java Bytecode for Analyses and Transformations, Technical report, Sable Technical Report 1998-4, Sable Research Group, McGill University, 1998.

[33] Valverde, S., Ferrer-Cancho, R., and Solé, R. V.: Scale-free Networks from Optimal Design, *Europhysics Letters*, Vol. 60,No. 4(2002), pp. 512–517.

[34] Walkinshaw, N., Roper, M., and Wood, M.: Understanding Object-Oriented Source Code from the Behavioural Perspective, *Proceedings of the 13th International Workshop on Program Comprehension*, 2005, pp. 215–224.

[35] Weiser, M. D.: Program slicing, *Proceedings of the 5th International Conference on Software Engineering*, IEEE Computer Society Press, 1981, pp. 439–449.

[36] Zhao, J.: Applying Program Dependence Analysis to Java Software, *Proceedings of the Workshop on Software Engineering and Database Systems*, December 1998, pp. 162–169.

[37] Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F.: SNIAFL: Towards a Static Non-Interactive Approach to Feature Location, *Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 293–303.