プログラムスライシングを用いた機能的関心事の抽出手法の提案と実装

仁井谷 竜介[†]　石尾 隆[†‡]　井上 克郎[†]

ソフトウェアは数多くの機能的関心事から構成される．また，多くの場合 1 つの関心事は複数のプログラム要素が関係しあって実現される．開発者がある関心事を理解するには，その関心事を実現している要素を探し，それらの間の関係を調べる必要がある．プログラムスライシングは要素間の関係を自動的に抽出する手法ではあるが，出力される要素が多すぎるため関心事の理解には適切ではない．これは，複数の関心事から共通で利用されるライブラリなどの要素もスライス結果として出力されるためである．

本研究では，開発者が注目している要素と他の要素との関連の強さを計算するヒューリスティクスを導入し，関連の強い要素だけを抽出するようプログラムスライシング手法を拡張する．また，得られたスライスをもとにクラス・メソッド単位での可視化を行う．Java で実装されたソフトウェアに対して適用実験を行い，関心事の理解に適したプログラムスライスを抽出できることを示した．

Software includes many functional concerns. And a functional concern is often implemented by collaborative software modules. When developers understand the implementation of a concern, they need to find the module units contributing to the concern and understand how the units collaborate with one another. Although program slicing is an automatic method to extract relationship among program elements in modules, slicing often results in many program elements to understand. The reason is that the result includes the common elements in various concerns, such as library elements.

In this paper, we extend program slicing to extract closely related elements, by introducing heuristics to calculate the degree of interest to a developer among program elements. And we visualize the slicing results using classes and methods. An experimental study for Java software shows that our method extracts a program slice suitable to understand a concern.

1　まえがき

近年，ソフトウェアの大規模化にともない，ソフトウェアの開発工程において保守工程の占める割合は年々増加の一途を辿っている．その中でもプログラム理解が占める割合は半分以上であると言われている [4].

開発者がプログラムの修正や再利用をするためには，どのプログラム要素が，どのように関係しあって機能的関心事を実現しているかを理解しなければならない [11]．ソフトウェアは数多くの機能的関心事から構成されているが，1 つの関心事を実現するプログラム要素は 1 つとは限らず，複数の要素がソースコード上で分散していることが多い．

したがって，開発者はまず関心事に関連のあるプログラム要素を探し，次にそれらの関係を理解する，という手順を踏むことになる．関連のある要素を探す手段として， Feature Location [15] [22] や部品検索 [6] といった手法が提案されている．これらの手法により，開発者は関連のある要素の一覧を取得できる．プログラム要素を見つけた後に，開発者はそれらの要素がいかに関係しあって関心事を実現しているかを理解する．要素間の関係として，制御フローやデータフローといったものがあり，これらの関係をもとに，関心事を構成するプログラム要素（この場合文）とその間の関係を抽出する手法としてプログラムスライシング [20] がある．

プログラムスライシングは，注目しているプログラム文（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全てのプログラム文を抽出する手法であるため，巨大なシステムに対して用いると，非常に多くの文が出力される．また，異なる関心事を調べるために，異なる文をスライシング基準としてプログラムスライシングを適用したとき，それらの結果に数多くの共通のプログラム文が含まれてしまう．共通して含まれる文の中には，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，開発者が理解しようとしている関心事との関連が弱いものが多い．

本研究では，特定の関心事に強く関連したメソッド集合とそ

Ryusuke Niitani[†], Takashi Ishio[†‡], 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

[‡] 日本学術振興会特別研究員 (PD), Research Fellow of the Japan Society for the Promotion of Science

の間の制御フローやデータフローを開発者に提示するために，プログラムスライシングにヒューリスティクスを導入した手法を提案する．通常の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，スライシング基準に対応する頂点からのグラフ探索によって計算される．本手法では，開発者が着目している関心事と，各メソッドとの関連の強さを経験的な指標を用いて評価し，関連が弱いと判定された要素へのグラフ探索を打ち切る処理を加えることで，関心事に関連が強い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．

本手法は，開発者がキーワード検索や Feature Location 手法を用いて発見した関心事との関連性があると判断される文の集合と探したい関心事のキーワードを入力として受け取り，関心事に強く関連していると判定されたメソッド中の文だけを含んだプログラムスライスを出力する．得られたプログラムスライスをクラス・メソッド単位で関心事グラフとして可視化する，あるいはソースコード上にマッピングを行うことで，開発者は関心事の理解を進めることができる．

jEdit[†1]を題材に適用実験を行った結果，この手法により，通常のプログラムスライスに比べて非常に小さな部分集合を抽出でき，開発者が関心事を理解する場合に適した情報を提示できることを確認した．

以降，節 2 でプログラムスライシングと関心事グラフについて述べる．節 3 で提案手法について述べ，節 4 で評価について述べる．節 5 で関連研究について述べ，最後に節 6 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である**プログラムスライス**(*Program Slice*, 以降スライス) を抽出する技術である [20].

スライシングの過程で，ソースプログラム p は文を表す頂点と，**制御依存関係**(*Control Dependence*) と**データ依存関係**(*Data Dependence*) を表す辺からなる**プログラム依存グラフ**(*Program Dependence Graph, PDG*) に変換される．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 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，PDG 上のスライシング基準からの逆方向または順方向への頂点の訪問によって抽出される．注目した変数に影響を与える文の集合を**バックワードスライス**(*Backward Slice*)，注目した変数の影響を受ける文の集合を**フォワードスライス**(*Forward Slice*) と呼ぶ．

†1 jEdit http://www.jedit.org/

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

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，膨大な数の文が出力される．その巨大な出力には，長い依存関係の列を多く含む．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，デバッグなどの用途で有益な場合もあるが，スライシング基準付近の局所的な振る舞いや関係を理解したい場合には適切ではない．

また，異なる関心事に対して異なるスライシング基準を与えたとしても，得られる結果の多くは共通している．これは，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，どの関心事に対しても関連のあるプログラム要素スライスに含まれているためである．これらの共通する要素は，ほとんどの関心事に対して関連が弱く，特定の関心事のみを理解しようとするときには，理解する必要がないことが多い．

表 1 は，jEdit に対して 2 つの関心事 (Autosave, Undo) に対応するスライスを従来のスライシングで計算し，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 を用いてキーワード “autosave”, “undo” を持つ文を検索し，得られたプログラム文とその中の全ての変数を選択した．

また，スライスに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を用いている．

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，差分は小さく，そのほとんどが共通している．これは，たとえば「ファイルが変更済みかどうか」を表すフラグのような，両方の関心事に登場するプログラム要素が存在することに起因する．このように関心事の実装は独立ではなく，共通する要素を持つことから，特定の関心事のみを抽出することは困難である．

2.3 スライシングに基づく手法

Chopping[7] は，ある始点となる文から終点となる別の文へ，他のどのプログラム文を経由して影響を与えているかを調べるための手法である．これは，プログラム依存グラフ上で，始点から終点に到達する全ての経路を計算するもので，始点からのフォワードスライスと終点からのバックワードスライスの積集合を計算することで求められる．

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，ある程度の知識が必要となる．しかし，開発者が関心事に対して十分な知識がなく，検索などで手掛かりとなる要素を見つけただけの段階では，このような始点と終点の区分を行えるとは限らず，Chopping を直接用いることはできない．

Chopping の計算では，始点と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ループ経路などを除去している [9].

スライシングの結果のサイズを小さくするために，スライシング基準からグラフ探索する距離を制限したスライシング [10] も提案されている．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ものである．k 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ができるが，関心事を理解するために十分な情報が得られるように適切な k の値を定めることは困難である．

2.4 関心事グラフ

関心事の表現は，多くの場合ソースコードの行を用いる．しかし，ソースコードによる表現は複数のファイルの間を何度も行き来する必要があり，全体像が把握しづらく，理解に時間がかかる．また，実装の断片がそのまま出力されているため，全体の様子を知りたい場合，詳細すぎることがある．スライシングも，文を単位としたソースコードの抽出技術であるため，同様のことが言える．

表 2　関心事グラフの辺の生成条件

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

関心事グラフ(*Concern Graph*) は Robillard ら [14] 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．このグラフは，関心事に関連するプログラム要素および関係の詳細を抽象化したグラフで，関心事に関連するクラス，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 などを辺とする．

亀田ら [2] は，ソフトウェア中の横断的関心事の抽出を行うために，スライシングの結果に基づいて自動的に関心事グラフを構成するための変換ルールを提案している．我々の提案手法とは利用目的が異なっているが，この変換ルールは一般的なスライシングの結果に適用可能であり，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はこの変換ルールを用いている．

表 2 は，関心事グラフ構築の際の変換ルールを示している（ただし本研究用に条件を変えていたり一部の辺を除いている）．このグラフにより，開発者はソースコード表現より短い時間で概要を理解することが可能となる．

3 提案手法

特定の関心事と関連の強いプログラム要素とその間の制御フローやデータフローを得るため，スライシングに関連の強さを求めるヒューリスティクスを導入した手法を提案する．Krinke [9] は，バリアとはスライシングの途中経路に登場してはならない頂点である定義しているが，我々の提案手法では，バリアを辺として定義し，ヒューリスティクスに基づいてバリアを自動的に発見する．ヒューリスティクスとしては，グラフの構造に基づくものと，識別子の名前によるものを用いている．

本手法は，入力として，開発者が関心事と関連があるのではないかと判断したメソッドやフィールドの一覧と，調べたい関心事を表すキーワードを受け取る．これらのメソッドやフィールドとしては，キーワード検索や様々な Feature Location 手法を用いて特定されたものを想定している．

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，バリアの集合を取得する．複数のヒューリスティクスが有効な場合は，それらの辺の和集合が得られる．
3. 指定されたプログラム文の集合に対して，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 の提案する *JSDG (the software dependence graph for Java)* [21] の定義を用いた．

プログラム依存グラフの頂点 1 つは，Java のバイトコードをもとにした中間言語である *Jimple* [17] の 1 命令に対応するものとした．

3.1.1 Jimple

図 1 は，Java のソースコード（左）と対応する Jimple（右）である．Jimple は Java のバイトコードをもとにしてるため，ソースコード上には存在しないコンストラクタ (`<init>`) が存在したり，条件分岐が `if/goto` 命令になっている．

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

グラフの各頂点は，以下の制御依存関係とデータ依存関係を表す有向辺によって接続される．

制御依存辺 条件分岐辺と呼び出し辺を含む．

- **条件分岐辺** 分岐命令 v_{from} と，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．
- **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，呼び出し先（メソッド入口の特殊頂点） v_{to} は呼び出し元（呼び出し命令） v_{from} に依存する．

データ依存辺 メソッド内依存辺と，パラメータ渡し辺，フィールド依存辺を含む．

- **メソッド内依存辺** 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．
- **パラメータ渡し辺** メソッド呼び出しにまたがって起こる引数および戻り値のデータ依存に対応する辺である．呼び出し先への入力では，呼び出し先 v_{to} で参照される実引数は，呼び出し元 v_{from} に入力として与えられた仮引数に依存する．呼び出し先からの出力では，呼び出し元 v_{to} で得られる戻り値や変更された仮引数は，呼び出し先 v_{from} の戻り値や変更された実引数に依存する．
- **フィールド依存辺** メソッド呼び出しにまたがって起こるフィールドによるデータ依存に対応する辺である．

また，異なる変数が同じオブジェクトを指している可能性がある（エイリアシングが起こる）ため，同じオブジェクトを指しうる変数を異なるものとして扱うとデータ依存関係が不正確になる．これに対しては，Points-to 解析およびクラス階層関係の解析を適用することで，各変数について実際に指しうるオブジェクトの情報を得て，実行時に起こりうる全てのメソッド呼び出しについて呼び出し辺，パラメータ渡し辺，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，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()

```
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　左:Java / 右:Jimple

```
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

は，Child1#something と Child2#something の両方に接続される．

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，関心事に関連していると開発者が選んだプログラム文の集合に対応する PDG 中の頂点集合 $criteria$，ヒューリスティクスごとのパラメータ（閾値など） $options$ から，バリアとして特定された辺の集合 E を求める次のような関数となっている．

$$PDG \times criteria \times options \rightarrow E$$

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 $E_1, \ldots, E_n$ が得られるので，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，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．このことから，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と言える [12] [18]．したがって，依存辺の入次数が大きい頂点は，複数の関心事の合流点であったり，ユーティリティ的な要素であることが多いと考えられ，特定の関心事を抽出するには不適切であると言える．

そこで，閾値 th より大きな入次数 $indegree$ （対象となる依存辺は，PDG 上の依存辺，またはコールグラフ上の呼び出し辺）を持つ頂点 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，あるいは自動で頂点数 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 $P[X > x] \sim x^{-(a-1)}$ で表されることから（a はほぼ 2），$x = \sqrt{N}$ 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．ただし，この自動的に決定される閾値は，依存辺が集中し，明らかに関連が弱いと推測される箇所を除去するための値で，理解支援に適切なサイズのスライスを取得するためのものではない．

図表中での略記は I とする． また，入次数の対象となる辺を全ての辺としたものは Ia ，呼び出し辺のものは Ic とする．

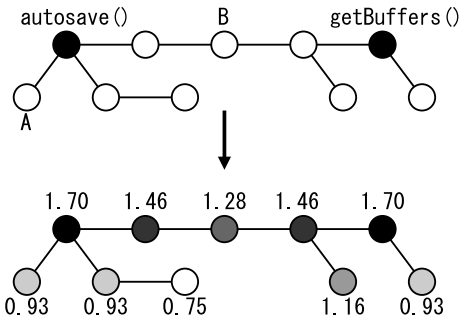

図 3　autosave + buffer に対するキーワードの評価値

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，識別子などに用いられている [15]．したがって，関心事を実現している箇所はキーワードを識別子に含むプログラム要素，あるいは，その近隣の要素からなっていると考えられる．

このヒューリスティクスは，開発者が探したい関心事についてキーワードを指定する必要がある．キーワードを名前の一部として含んだ各メソッドから，以下の方法でメソッドの評価値を決定する．

- 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_1 から m_2 までの最短パスの長さを頂点間の距離 $d(m_1, m_2)$ とする．
- ある評価対象のメソッド m に対する評価値 $V(m)$ は，キーワードを含んだ n 個のメソッド $m_1, \ldots, m_n$ 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．

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

上記の式の定義は,「キーワードを含んだメソッド自身はスライスに含まれやすくする」「キーワードを含んだ複数のメソッドと近い位置にあるメソッドもスライスに含まれやすくする」ということを考慮している．

図 3 は，あるプログラム依存グラフに対して autosave, buffer の 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．上段はもとのプログラム依存グラフで，黒く塗られた頂点がキーワードにマッチしたメソッドで，下段の数値が評価値である．

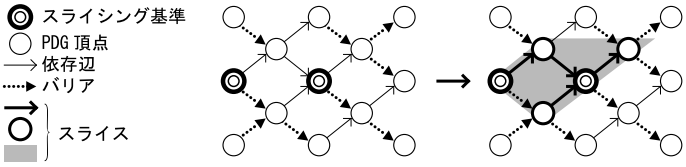

図 4　バリア付きスライシングの実行

こうして定めた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．

$$\begin{aligned} isBarrier(v_{from}, v_{to}) &= m_{from} \neq m_{to} \\ &\wedge \quad V(m_{from}) < th \\ &\wedge \quad V(m_{to}) < th \end{aligned}$$

この手法は，キーワードにマッチしたメソッドからの距離に基づいていることから，もし開発者が grep のようなキーワードに基づく手法によってスライシングへの入力を決定した場合，長さ制限付きスライシング [10] の手法と近い結果が得られるように見える．しかし，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が異なっている．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．この手法により，長さ制限付きスライシングでスライスサイズを絞り込んだときに途切れてしまうような関係を，同じサイズのスライスでも取得できることが期待される．

図表中での略記は K とする．

3.3　バリア付きスライシングの実行

このステップでは，PDG，スライシング基準，前のステップで得られたバリア集合を入力とし，与えられたバリア集合を通過しないような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．提案手法はスライシング基準を複数取りうるが，スライシング基準に対して入力や出力の区別をしないため，スライシングでは順方向と逆方向に依存辺を辿る．また，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るような形で出力される．

具体的なアルゴリズムとしては，Horwitz らの定義 [5] [13] に，バリア計算のアルゴリズム [9] を修正したものを組み合わせて

いる．[9] 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ことを目的としている．

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，最終的に得られたスライスを関心事グラフに変換する．

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コメントを記入などしてマッピングを行うことで，関心事グラフと，その詳細としてのソースコードとを併用して，関心事を理解していくことができる．

本手法における関心事グラフの頂点は [14] で提案されているものと同様に，クラス，メソッド，フィールドとし，辺は表 2 によって生成されるものとする．

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を作成した．このツールは，プログラム依存グラフ (PDG) 構築部と，スライシング計算部からなる．

まず，PDG 構築部は Java バイトコードを入力として受け取り，PDG の構築を行う．バイトコード解析および Points-to 解析には，Soot[†2] フレームワークを用いた．

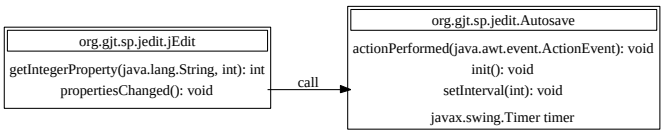

図 5　出力される関心事グラフの例

表 3　実験に用いた機能的関心事

	関心事		grep	
	メソッド	クラス	メソッド	クラス
Autosave	34	14	19	6
Undo	26	7	28	7
VFS Search	14	9	13	11

次に，スライシング計算部は，作成された PDG とスライシング基準，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の適用，スライシングを行う．最終的な出力は，計算されたスライスの情報と，それを関心事グラフに変換したものとなる．関心事グラフの可視化には，Graphviz[†3] を用いた．

関心事グラフの出力例を図 5 に示す．グラフの複雑化を避けるため，1 つのクラスに属するメソッドやフィールドは 1 つのクラスに対応する頂点の中に含めている．その際，private メソッドに対する呼び出し辺や，private フィールドの読み書きを表す辺は省略されている．

4.1 実験対象ソフトウェア

実験対象には， jEdit(4.2final) 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．開発の履歴は Subversion により管理されているため，機能追加や不具合修正などを行った際に，同時に更新されたファイルとその変更内容が容易に取得できる．

この実験で扱う機能的関心事として，表 3 の 3 つを用いた．各関心事の詳細は以下の通りである．

Autosave　マニュアルにある機能から選んだものの 1 つ．編集中のバッファの内容を自動的に保存する機能．この機能の動作は GUI による設定で変更が可能である．

Undo　マニュアルにある機能から選んだものの 1 つ．バッ

†2　http://www.sable.mcgill.ca/soot/
†3　Graphviz http://www.graphviz.org/

表 4　実行時間

PDG 構築	42m49s
関心事抽出	3m52s - 19m51s

ファに対する編集を取り消す機能．Redo と対になる．この機能の動作は GUI による設定で変更が可能である．

VFS Search　リビジョン 4199 で追加された機能．このリビジョンでは 12 個のクラスが変更されているが，実験対象のバージョンのソースコード上では存在しなかったり，大幅に変更を加えられたものを除くと 8 クラスになる．非ローカルファイルシステムに対するディレクトリ検索．

スライシング基準の選択には，プログラム全体に対してそれぞれ grep で “autosave”, “undo”, “vfs” + “seach” をキーワードとして検索をかけ，発見された文を選んでいる．ヒューリスティクスのキーワードとしても，同様に “autosave”, “undo”, “vfs” + “seach” を用いている．表 3 の “grep” の列は，それぞれ発見された文を含むメソッドおよびクラスの数を表している．

また，表 3 の “関心事” の列は，それぞれ関心事との関連が強いと判断されたメソッドおよびクラスの数を表している．

スライシングの実行には Intel Server (Xeon 2.8GHz, Memory 8GB) を用い，表 4 に示す時間がかかった．なお，スライシングにかかる時間は有効なヒューリスティクスとそのオプションに依存する．

4.2　スライスのサイズ

従来のスライシングは，得られるスライスのサイズが大きすぎることが，関心事理解支援における 1 つの問題であった．そこで，まず従来手法と本手法によって得られたスライスを比較することによって，本手法によって得られるスライスが，理解支援に利用することが現実的なサイズになりうることを確認する．

表 5 は，同一のスライシング基準 (Autosave) を与えたときの，従来手法と提案手法のスライスのサイズである．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（略称は小節 3.2 参照）．

この結果から，提案手法によって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シング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ことが確認された．

入次数のヒューリスティクスは閾値が整数であるが，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している．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．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．

表 5　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	19	6
従来手法	405,801	4,720	658
K 0.95	367,914	3,331	641
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 1.35	9,073	39	17
K 1.40	3,873	24	9
K 1.45	849	20	6
Ia auto	397,306	4,652	654
Ia 16	353,292	4,527	651
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
Ic 0	2,766	87	32

したがって，入次数のみで適切なサイズのスライスを得ることは困難であり，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　各ヒューリスティクスの組み合わせ

小節 4.2 では，スライスのサイズが現実的に使用可能な数に収まっていることを確認した．次に本小節では，各ヒューリスティクスを組み合わせたときのサイズ，適合率，再現率，F 値について評価を行う．

適合率(*Precision*) とは以下の式で与えられる値で，スライス

中にどれだけ関心事に適合するメソッドがあるかを表す.

$$Precision = \frac{|M_{slice} \cap M_{concern}|}{|M_{slice}|}$$

M_{slice} スライスに含まれるメソッド集合
$M_{concern}$ 関心事に適合するメソッド集合

同様に，**再現率**(*Recall*) とは以下の式で与えられる値で，関心事に適合するメソッドの内，どれだけのメソッドがスライスに含まれたかを表す.

$$Recall = \frac{|M_{slice} \cap M_{concern}|}{|M_{concern}|}$$

適合率と再現率は一般にトレードオフの関係にある．したがって，以下の式で与えられる F 値 を評価に用いる．

$$F = \frac{2Precision \times Recall}{Precision + Recall}$$

表 6 は，キーワードによるヒューリスティクスと入次数によるヒューリスティクスを組み合わせたときのスライスの値である．入次数によるヒューリスティクスは，閾値が小さい時を除き，従来のスライシングと大差のないサイズのスライスを出力していた.

しかし，2 つのヒューリスティクスを組み合わせた場合は，キーワードのヒューリスティクス単体の時 296 メソッドだったのに対し，それぞれ 239 メソッド (80.1%), 89 メソッド (30.1%) に減少させている．また，キーワード単体のときと比べ，再現率を全く損なわずに適合率を向上させている.

これは，2 つのヒューリスティクスが組み合わさることにより，一方だけではバリア以外の辺から迂回されて到達できた関連の弱い頂点群を，スライスから除去できたためだと考えられる.

4.4 スライスの共通箇所

従来のスライシングの関心事理解支援におけるもう 1 つの問題として，開発者が理解しようとしている関心事との関連が弱いプログラム要素が，異なる関心事のスライスに共通して出現することがあげられる．そこで次に，異なる関心事に対してスライシングを行い，どれだけのプログラム要素が共通しているかを調査する.

提案手法によって得られるサイズの大きなスライスは，関連の弱さをある程度許容したスライスであるため，従来のスライスと同様の性質を持つことが予想される．そこで，ここでは提案手法によって得られる小さなスライスについて評価する.

表 7 は，3 つの関心事に対して得られたスライスに共通している箇所のサイズを表している．いずれの組み合わせも，共通して出現するプログラム要素は十分に少ない．この結果から，本手法はそれぞれの関心事に強く関連した要素だけを容易に発見することができるといえる.

表 7　異なる関心事に対する提案手法のスライスの共通箇所

	頂点数	メソッド数	クラス数
Autosave	9,073	39	17
Undo	2,844	68	24
VFS Search	7,062	67	30
A ∩ U	70	3	2
A ∩ V	0	1	3
U ∩ V	0	0	1

Autosave, Undo に共通して得られたプログラム要素は以下の通りであった.

メソッド

- org.gjt.sp.jedit.Buffer#finishSaving
- org.gjt.sp.jedit.Buffer#setDirty
- org.gjt.sp.jedit.PluginJAR#activatePlugin

クラス

- org.gjt.sp.jedit.Buffer
- org.gjt.sp.jedit.PluginJAR

2 つの関心事は，ともに「バッファが変更済みかどうか」を表すフラグを扱っているため，finishSaving, setDirty が共通して登場している．これは，関心事の抽出として正しい動作である.

また，activatePlugin はプラグインを有効化するメソッドであるが，このメソッドはどちらの関心事ともに関連が弱い.

4.5 長さ制限付きスライシングとの比較

本手法で提案しているキーワードを用いたヒューリスティクスは，頂点間の距離に基づいているため，長さ制限付きスライシングと類似している．しかし，意味的な情報や複数頂点を考慮している点など，長さ制限付きスライシングより有益なスライスが得られることが期待される．ここでは，キーワードのヒューリスティクスと長さ制限付きスライシングの 2 つの手法により得られたスライスを比較する.

4.5.1 スライスの比較

表 8 は，2 つの手法により得られた同程度の頂点数のスライスおよび同程度のメソッド数のスライス間の比較である．K 1.35 はキーワードのヒューリスティクスとその閾値を，L 1 は長さ制限付きスライシングとその長さを表している.

表 6　2 つのヒューリスティクスを組み合わせたときのスライスのサイズ

	頂点数	メソッド数	クラス数	適合率	再現率	F 値
K 1.20	56,007	296	135	0.0642	0.5938	0.1159
+ Ia auto	43,138	239	108	0.0794	0.5938	0.1402
+ Ia 4	7,288	89	29	0.2134	0.5938	0.3140
Ia auto	397,306	4,652	654	0.0069	1.0	0.0136
Ia 4	109,727	2,179	485	0.0091	0.625	0.0181

表 8　同程度の頂点数 / メソッド数の長さ制限付きスライスとの比較

	頂点数	メソッド数	クラス数	適合率	再現率	F 値
Autosave						
K 1.35	**9,073**	**39**	17	0.3590	0.4375	0.3943
L 1	567	**45**	16	0.2667	0.3750	0.3117
L 4	**9,249**	479	175	0.0480	0.7186	0.0900
Undo						
K 1.15	**3,490**	**75**	31	0.2667	0.7692	0.3960
L 2	1,289	**98**	30	0.1837	0.6923	0.2903
L 4	**3,961**	274	83	0.0802	0.8461	0.1467
VFS Search						
K 1.15	**7,062**	**67**	30	0.0896	0.4286	0.1481
L 1	490	**54**	25	0.0926	0.3571	0.1471
L 5	**6,531**	541	195	0.0203	0.7857	0.0396

3 つの関心事で，ともに提案手法が長さ制限付きスライシングより高い F 値 を示した．したがって，長さ制限付きスライシングより有益なスライスが取得できたと言える．

また，関心事に関係なくメソッド数，クラス数ともに長さ制限付きスライスの方が多くなっている．言い換えれば，提案手法のスライスは，メソッド当たり（あるいはクラス当たり）の頂点数の密度が高いと言える．

4.5.2　関心事グラフの比較

次に，関心事 Autosave について，含まれるメソッド数が同程度のスライスから得られた関心事グラフの比較を行う．同程度の頂点数のスライスは明らかに理解支援に適さないサイズなので，ここでは扱わない．

図 6 はキーワードのヒューリスティクスの関心事グラフ，図 7 は含まれるメソッド数が同程度の長さ制限付きスライスから得られた関心事グラフを表してる．

2 つの関心事グラフを比べると，長さ制限付きスライスの図の上部にある接続されていないクラスが目立っている．その中には，com.microstar.xml.XmlParser，org.gjt-.sp.jedit.MiscUtilities，bsh.Primitive のように関連の弱いクラスや，org.gjt.sp.jedit.PerspectiveManager，org.gjt.sp.util-.WorkThreadPool のように，関心事との関連の強いクラスが含まれていた．関連の弱いクラスはそもそも含まれることが望ましくないが，関連の強いクラスも，ただそこに含まれるだけでは関心事理解支援に有益とは言い難い．そこに強い結びつきがあるのであれば，他の関連の強いクラスとの関係を提示できる方が望ましい．

このことからも，提案手法は長さ制限付きのスライシング手法と比べ，関心事の理解に適したスライスを抽出できていると言える．

4.6　閾値の変化による関心事グラフの変化

提案手法は，ヒューリスティクスに与える閾値によって得られるスライスが大きく変化する．ここでは，キーワードマッチと距離に基づく境界のヒューリスティクスの閾値を変化させた

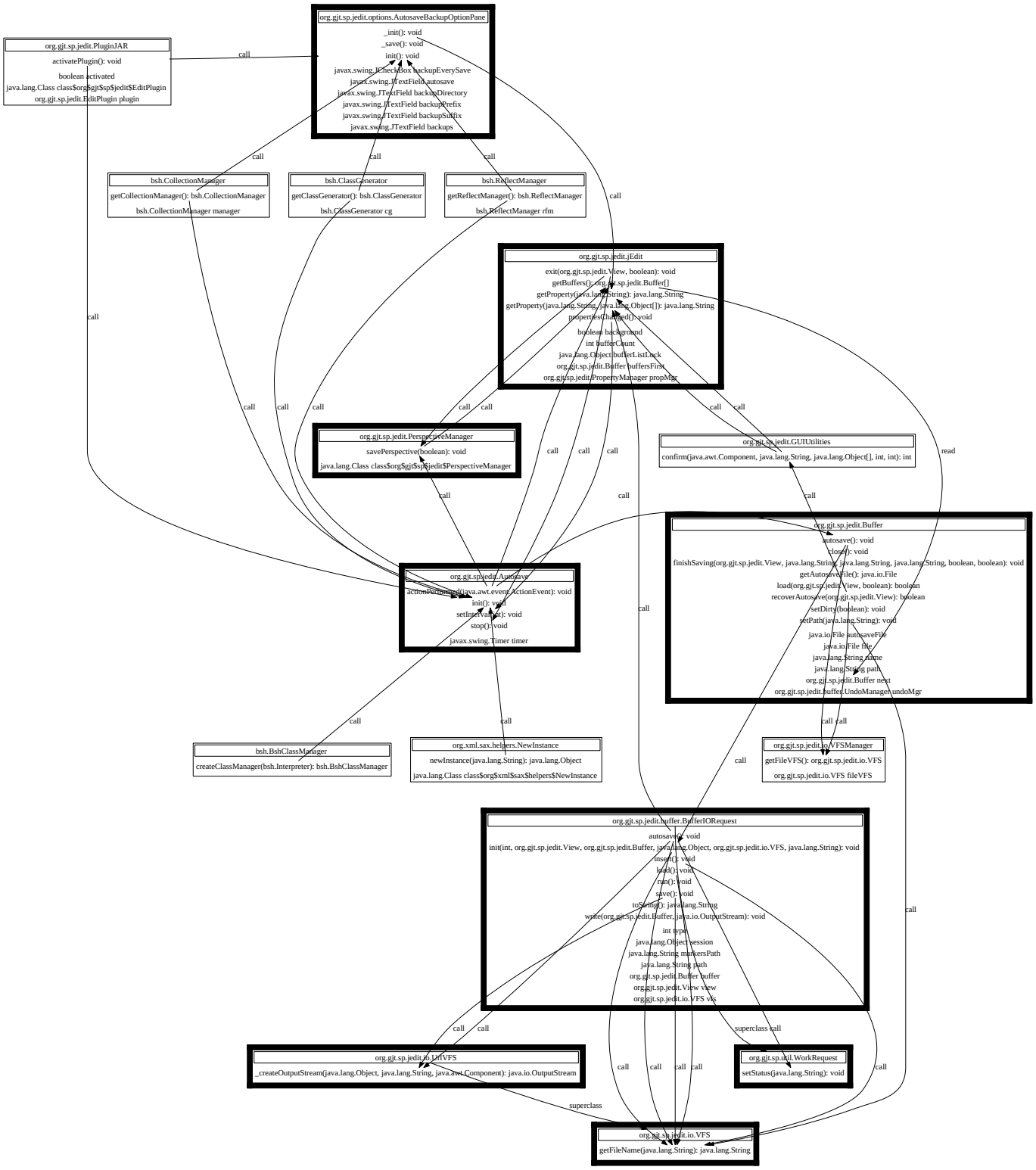

図 6　キーワードのヒューリスティクスのみ (Autosave / 閾値 1.35)

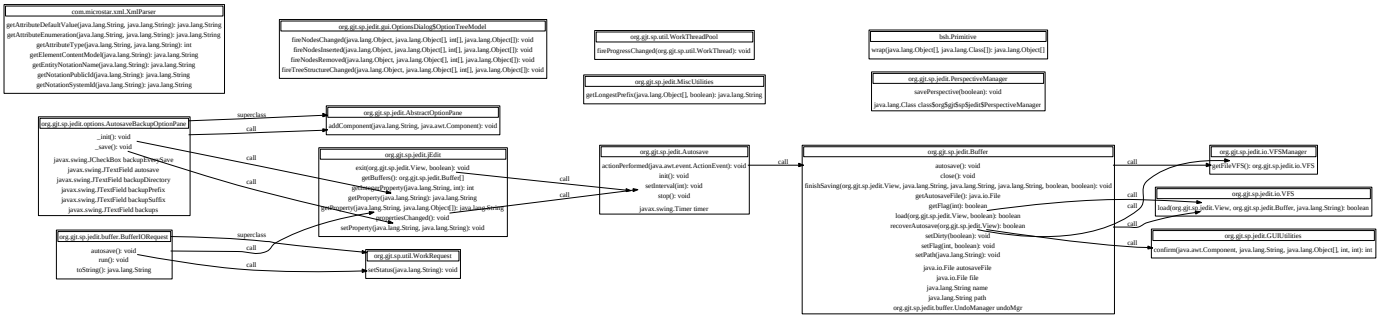

図 7　長さ制限付きスライシング (Autosave / 最大長 1)

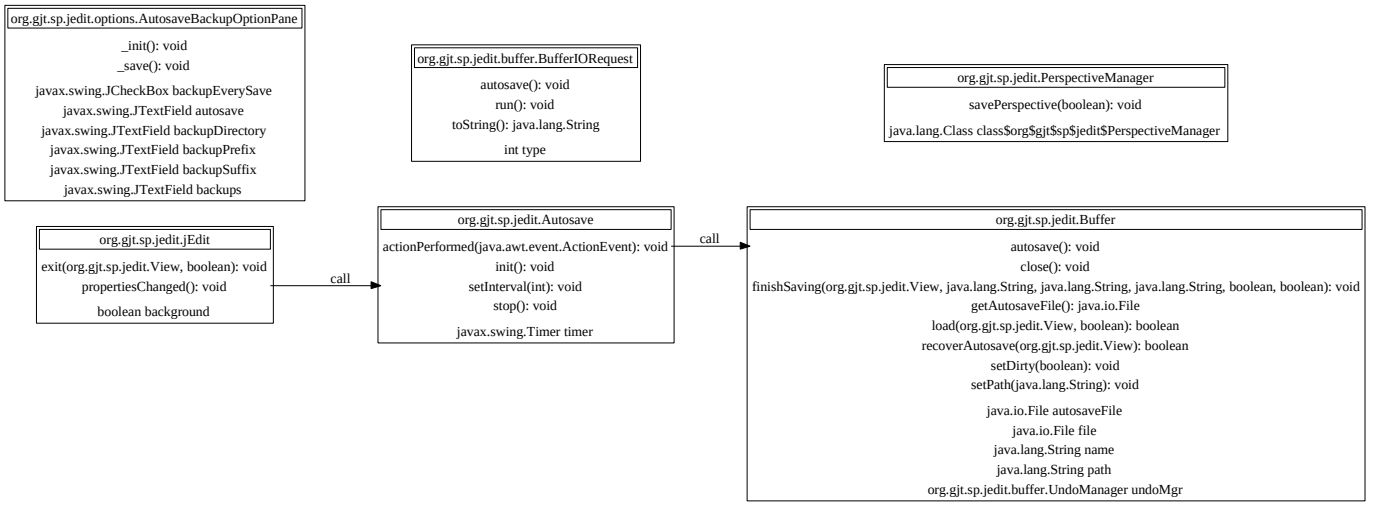

図 8　最もクラス数の少なかった Autosave 関心事グラフ

例を図 6, 8 に示す.

図 8 は，スライシング基準として選ばれたクラス以外はグラフに含んでいない．そのため，非常に簡潔な図となっている．

一方で，関心事 Autosave を理解する上で正確であるものは図 6 であった．サイズや適合率などの情報は，表 8 の Autosave K 1.35 の行に示している．関心事グラフに含まれていたクラスは 17 個あり，そのうち Autosave の実装において重要なクラスは 9 個あった．重要なクラスは，太枠で囲んである．

図 8 は関心事の全てを網羅しているわけではないが，Autosave を構成する基本的なクラス間の関係を提示している．そこで，開発者には，まずこの簡潔な図を提示しておき，段階的に閾値の低い，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が考えられる．これは，Storey ら [16] 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．

適切な閾値の決定については，メソッドの評価値の分布に基づいて閾値を決定する方法などが考えられるが，それは今後の課題とする．

5　関連研究

SNIAFL [22] は機能に関連するメソッドを抽出するために，情報検索手法と仮想の実行トレースを用いている Feature Location 手法である．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構造を調査するのに有益である．

Kersten らは Mylar プロジェクト [8] で，開発者が頻繁に見ている箇所を関心事とみなしている．Mylar の DOI モデルでは，開発者の興味に従って IDE の View 上での表示（強調，非表示）

を変更することで，開発者が効率的にタスクをこなすことができるよう支援している．開発者が閲覧したり編集したりした箇所を興味の対象としてランクを上げ，見なくなった箇所は興味がなくなったものとしてランクを下げており，特別な入力を必要としないことが利点である．しかし，ある程度開発者がタスクをこなしていないと DOI モデルが使えないため，開発初期は支援ができない．また，恣意的な検索などはできない．

Shepherd ら [15] は，ソースコードに対して自然言語処理を行うことにより，関心事の抽出を可能としている．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，抽出されたメソッド間の依存関係については，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．この手法によって出力さ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．

関心事の表示方法としては，行に色を付けるものや，Mylar のように IDE の View 上に色を付けるもの以外にも，Seesoft [3] のようにプログラム全体を見渡せる表示を持つものがある．Seesoft での表示は，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，関心事に関連する行に相当する部分を，その関心事に対応した色でマークづけるものである．

Walkinshaw ら [19] は，特定のユースケースやシナリオでの振る舞いの理解支援のため，スライシングに基づく手法を提案し，プログラムのコールグラフから部分グラフを抽出している．この手法は，開発者に調べたい振る舞いが含むべきメソッド群 (*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．ユースケースやシナリオは通常，非機能的関心事を含むいくつかの関心事により構成される．しかし，我々の手法は他のユースケースでも出現するような関心事を除いた，開発者によって指定された関心事のみに含まれるプログラム要素の抽出を目的とする．

Chen ら [1] は Feature Location をするシナリオとして，起点からの依存グラフの作成した上で，ユーザによる選択と検索グラフの更新を繰り返すことを提案している．彼らの手法では，ユーザの選択する候補は依存関係のあるもの全てだが，我々の手法でフィルタすることによって，選択のコストが抑えられるのではないか，と考えられる．

Krinke [10] は，スライシング基準からの距離を制限したスライシングを提案している．しかし，我々の手法は単純なグラフ上の距離ではなく，関心事との関連性の強さを計ることにより訪問を停止しているので，より適切な構造を抽出できることが期待される．

6 まとめ

プログラム要素がどのように関係して機能的関心事を実現しているかを理解することは，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，開発者が指定したプログラム要素から，機能的関心事の理解支援に適切な，プログラム依存グラフの部分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頂点からの距離を用いたヒューリスティクスによって，開発者の入力と関連の強いと判定された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 を対象とした適用実験によって，関心事の理解に適したプログラムスライスを抽出できることを示した．

今後の課題としては，複数の関心事の間の相互作用を調査するための本手法の拡張や，ヒューリスティクスに与える閾値の自動計算手法の確立などがあげられる．

謝辞 有益なご助言をいただきました University of British Columbia, Gail Murphy 教授に心より深く感謝いたします．

参 考 文 献

[1] Chen, K. and Rajlich, V.: Case Study of Feature Location Using Dependence Graph, *IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension*, 2000, pp. 241.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] Jackson, D. and Rollins, E. J.: A new model of program dependences for reverse engineering, *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, 1994, pp. 2–10.

[8] Kersten, M. and Murphy, G. C.: Mylar: a degree-of-interest model for IDEs, *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, 2005, pp. 159–168.

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

[10] Krinke, J.: Visualization of Program Dependence and Slices, *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, 2004, pp. 168–177.

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

[12] 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.

[13] Reps, T., Horwitz, S., Sagiv, M., and Rosay, G.: Speeding up slicing, *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, 1994, pp. 11–20.

[14] Robillard, M. P. and Murphy, G. C.: Concern graphs: finding and describing concerns using structural program dependencies, *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, 2002, pp. 406–416.

[15] 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, *accepted for publication at the International Conference on Aspect Oriented Software Development, 2007*, 2007.

[16] 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.

[17] Vallee-Rai, R. and Hendren, L. J.: Jimple: Simplifying Java Bytecode for Analyses and Transformations.

[18] 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.

[19] Walkinshaw, N., Roper, M., and Wood, M.: Understanding Object-Oriented Source Code from the Behavioural Perspective, *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, 2005, pp. 215–224.

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

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

[22] Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F.: SNIAFL: Towards a Static Non-Interactive Approach to Feature Location, *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 293–303.