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.

プログラム依存グラフの一貫性検査に基づく欠陥候補の検出手法

山田　吾郎† 　吉田　則裕†† 　井上　克郎†

† 大阪大学 大学院情報科学研究科
〒 560-0871 大阪府吹田市山田丘 1 番 5 号
†† 奈良先端科学技術大学院大学 情報科学研究科
〒 630-0192 奈良県生駒市高山町 8916-5
E-mail: †{g-yamada,inoue}@ist.osaka-u.ac.jp, ††yoshida@is.naist.jp

あらまし　本稿では，プロジェクト内の各ソースファイルから構築したプログラム依存グラフに対し一貫性検査を行うことで，ソースコードから欠陥候補を検出する手法を提案する．プログラム依存グラフとは，文を頂点とし，制御依存辺とデータ依存辺の 2 種類の辺をもつ有向グラフである．まず，構築したプログラム依存グラフの集合から，頻出部分グラフを抽出する．次に，頻出部分グラフの一部が欠落した部分グラフを検出し，対応するソースコードを欠陥候補として開発者に提示する．

キーワード　ソースコード解析，欠陥検出，プログラム依存グラフ

A Defect Detection Technique Based on Consistency Checking of Program Dependence Graphs

Goro YAMADA†, Norihiro YOSHIDA††, and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka, 565-0871 Japan
†† Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama-cho Ikoma, Nara, 630-0192 Japan
E-mail: †{g-yamada,inoue}@ist.osaka-u.ac.jp, ††yoshida@is.naist.jp

Abstract　In this paper, we propose a novel approach to detecting defects from source code, which checks for consistency of program dependence graphs. A program dependence graph is a labeled directed graph, of which nodes represent statements in a method, and edges represent control and data dependences. At first, our method extracts frequent subgraphs from program dependence graphs. Then, it detects defect candidates which lack some nodes compare to its corresponding frequent subgraphs. Finally, it provides code fragments corresponding to those candidates.

Key words　Source code analysis, Defect detection, Program dependence graph

1. ま え が き

イディオム [9] とは，特定の処理を実現するためパターンである．イディオムはソースコード中の複数箇所に，類似したコード片となって出現する．イディオムにはファイルを開き，それを閉じる，といった一般的なものがある一方，プロジェクト固有なイディオムも存在する．イディオムを用いたコード片は，開発者により一から実装される他，コピーアンドペーストを用いて複製されることもある [3].

イディオムを実装する際には欠陥が混入する危険性がある．例えばプロジェクトに不慣れな開発者がプロジェクト固有の API を用いる場合，ドキュメントの不備により誤った実装をする可能性がある．また，コピーアンドペーストをしたのち，識別子の変更を忘れてしまい欠陥を作り込むことも考えられる [4].

これまでに，このような欠陥を検出する手法が考案されてきた．Li らは，プログラムの関数中で同時に使われることの多い関数呼び出し文の集合をイディオムとして抽出し，イディオムに違反しているコード片を欠陥候補とする手法を考案した [5]. Li らはこの手法を用い，Linux カーネルのソースコードから 16

個，PostgreSQL のソースコードから 6 個の欠陥を発見した．また，Wasylkowski らはオブジェクトの使用方法をイディオムとして抽出し，それ従わないコード片が少数であった場合に欠陥候補とする手法を考案した [8]．Wasylkowski らはこの手法を AspectJ のソースコードに適用し，2 個の欠陥を発見した．

いずれの手法においても，欠陥の検出という点で一定の成果を出した．しかし，これらの手法ではイディオムの抽出の時点でデータ依存関係, 制御依存関係が消失する．これらの消失により，例えば，Wasylkowski らの手法では複数オブジェクトの相互作用を表すイディオムが抽出できなくなる．つまり，イディオムの抽出精度において問題が生じる．

本研究では，**プログラム依存グラフ**の集合から抽出した頻出部分グラフがイディオムを表すと仮定し，頻出部分グラフから違反を検出することでこの問題の解決を試みた．プログラム依存グラフとは，頂点が文を表し，辺が制御依存・データ依存を表すグラフである．これらを考慮することにより，従来手法では抽出できなかったイディオムが取得できると考えられる．提案手法は以下の 3 ステップにより実現される．

ステップ 1　入力としてソースコードを受け取り，各メソッドからプログラム依存グラフを構築する．

ステップ 2　構築したプログラム依存グラフから頻出する部分グラフを抽出する．この頻出部分グラフがソースファイル中でのイディオムに該当する．

ステップ 3　全ての頻出部分グラフに対し，相関ルールと呼ばれるルールを構築する．さらにそれぞれの相関ルールについて，確信度と呼ばれる値を計算する．計算した確信度に基づき頻出部分グラフに違反しているコード片を検出し，欠陥候補とする．

ステップ 1，**ステップ 2** は肥後らの作成したツール **Scorpio**(注1) [11] を元に，本手法に適するよう調整を行った手法を用いる．

提案手法の有効性を確認するため，オープンソースプロジェクトである MASU を対象に適用実験を行った．その結果，検出された欠陥候補の中に 1 つの欠陥が含まれており，本手法の欠陥検出能力を確認できた．

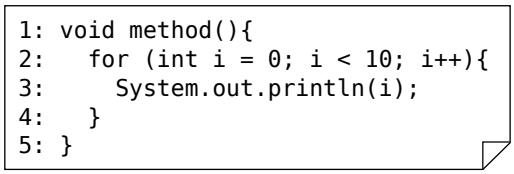
以降，2. 節ではプログラム依存グラフの構築方法と頻出部分グラフの抽出方法について，3. 節ではプログラム依存グラフからの欠陥候補の検出方法を述べる．4. 節で試作したツールの使用例とその考察を行い，5. 節では本研究の関連研究を解説する．最後の 6. 節でまとめと今後の課題について述べる．

2. Scorpio を用いたイディオムの抽出

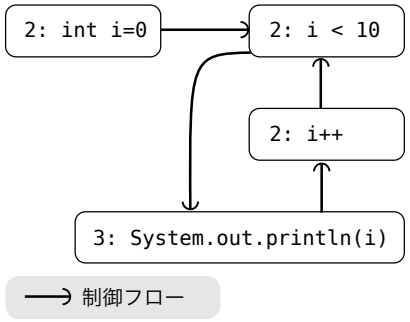
本節では，イディオムの抽出に用いた手法について説明する．

本手法で抽出するイディオムとは，プログラム依存グラフから抽出した頻出部分グラフである．頻出部分グラフの抽出には肥後らの開発したツール Scorpio を欠陥検出に適するよう調整して用いた．Scorpio では，まず，ソースコード中の各メソッドからプログラム依存グラフを構築し，次に構築したプログラム依存グラフから頻出部分グラフを抽出する． 以降，2.1 節でプログラム依存グラフの構築方法を，2.2 節で頻出部分グラフの抽出方法を説明する．

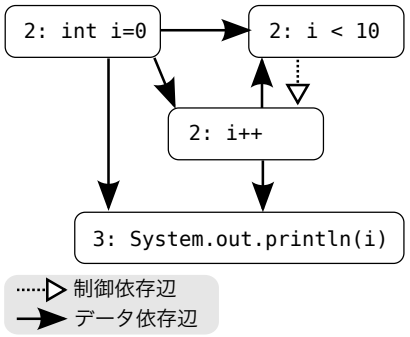
(注1)：Scorpio はコードクローン検出ツールとして開発されたが，本研究ではイディオム抽出に用いる



(a) ソースコード



(b) (a) から構築される制御フローグラフ



(c) (a) から構築されるプログラム依存グラフ

図 1　制御フローグラフとプログラム依存グラフの例

2.1 プログラム依存グラフ

Scorpio でのプログラム依存グラフは制御フローグラフから生成される．以降で制御フローグラフの構築手法，および制御フローグラフからのプログラム依存グラフ構築手法を説明する．なお，図 1(b) は図 1(a) から構築される制御フローグラフを，図 1(c) は図 1(a) から構築されるプログラム依存グラフをそれぞれ表している．

2.1.1 制御フローグラフの構築

制御フローグラフはプログラムの表現形態の一種であり，プログラムの開始から終了までの全ての実行経路を表す．

Scorpio での頂点・辺の定義は以下のとおりである．

頂点　文と条件節の式. 図 1(a) では，文`System.out.println(i)`, 式`i < 10`, `int i = 0`, `i++`が頂点となる．

辺　プログラム要素 A の実行直後に要素 B が実行される可能

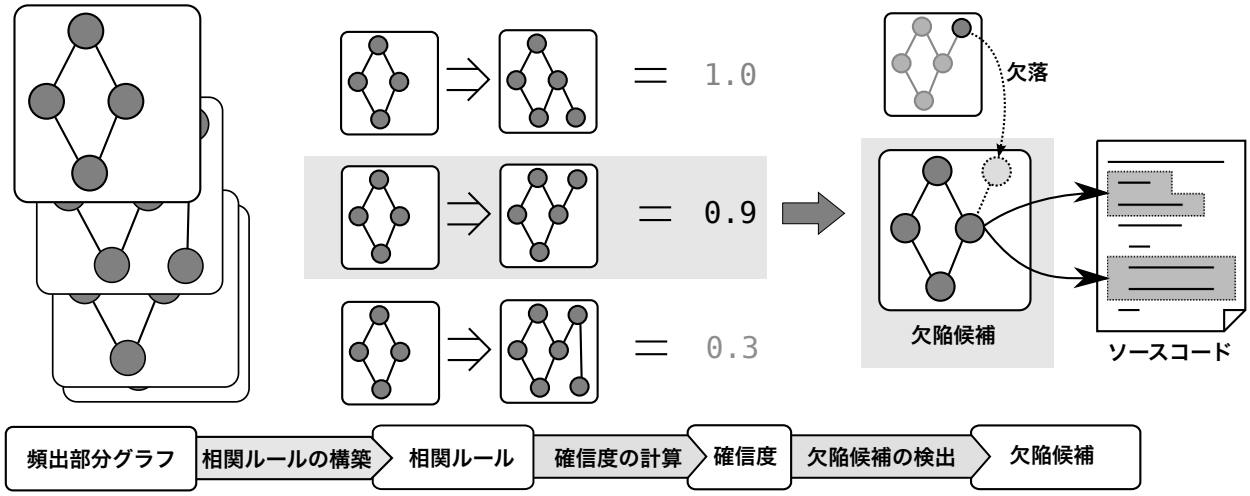


図 2　提案手法の概要

性がある場合，頂点 A から B に向けて辺を引く．
制御フローグラフはソースコード中の各メソッドに対し構築される．

2.1.2　プログラム依存グラフの構築

プログラム依存グラフは制御フローグラフを入力として，以下の方法で構築される．

頂点の生成　1 つの制御フローグラフの頂点から 1 つのプログラム依存グラフの頂点が生成される．したがって，両者の頂点は完全に一対一対応する．

データ依存辺の生成　制御フローグラフにおいて，変数に値を代入している各頂点に対し，その変数を代入している頂点を制御フローをたどりながら検出する．そして，プログラム依存グラフ上の対応する頂点間に，代入している頂点から参照している頂点に向けて辺を引く．

制御依存辺の生成　条件式を表す頂点から，ソースコード中で条件文の内部にある文に向けて辺を引く．

2.2　プログラム依存グラフからの頻出部分グラフ抽出

本節では，2.1 節で述べたプログラム依存グラフから頻出部分グラフを抽出する手法について述べる．ここで抽出される頻出部分グラフが，本研究でのイディオムである．抽出は以下の 4 ステップにより行われる．

ステップ 1　プログラム依存グラフの全頂点のハッシュ値を求め，ハッシュ値が同じ頂点ごとにグループを作成する．ハッシュは頂点が表すプログラム要素の構造に基づいて計算される．ハッシュ値が計算される前に変数やリテラルはその型に変換されるため，利用している変数やリテラルが異なっていても，それらの型が同じであり，そのプログラム要素の構造が等しければ，それらは同じハッシュ値を持つ．

ステップ 2　同じグループに属する頂点のペア $(r1, r2)$ を含む同型部分グラフのペアを検出する．このペアは頻出部分グラフの候補となる．検出の基点は頂点 $r1$ と $r2$ であり，そこから引かれている辺を順方向，及び逆方向に探索する．2 つの探索は同期して行われる．探索により新たにたどった頂点のハッシュ値が等しい場合はそれらを同型部分グラフの頂点として加える．下記条件のいずれかを満たすとき，たどった頂点は同型部分グラフに加えられず，探索を終了する．

条件 1　新たにたどった頂点のペア $(p1, p2)$ が異なるハッシュ値を持つ場合

条件 2　$(p1, p2)$ のハッシュ値は等しいが，$r1$ のグラフ (または $r2$ のグラフ) がすでに $p1$(または $p2$) を含んでいる場合 (この条件は，無限ループを回避するために設定).

条件 3　$(p1, p2)$ のハッシュ値は等しいが，$r1$ のグラフ (または $r2$ のグラフ) が $p2$(または $p1$) を含んでいる場合 (この条件は，2 つの同型部分グラフが頂点を共有しないために設定).

この処理を同じグループに属する全ての頂点のペアに対して行う．

ステップ 3　ステップ 2 で検出した頻出部分グラフの候補 $(s1, s2)$ が他の候補 $(s1', s2')$ に含まれていた場合 ($s1 \subseteqq s1' \cap s2 \subseteqq s2'$)，その候補を検出された頻出部分グラフの候補の集合から削除する．

ステップ 4　同部分グラフを持つ候補から頻出部分グラフを形成する．例えば，2 つの候補 $(s1, s2), (s2, s3)$ があった場合，頻出部分グラフとして $\{s1, s2, s3\}$ が形成される．

3.　頻出部分グラフからの欠陥候補の検出

本節では，2. 節で述べた手法により抽出した頻出部分グラフから，どのように欠陥候補を検出するか説明する．

頻出部分グラフからの欠陥候補の検出は，2 つの頻出部分グラフ間における**相関ルール** [1] の**確信度**を用いて行われる．本研究での相関ルールとは，プログラム依存グラフ P において，頻出部分グラフ G_A が存在したとき，頻出部分グラフ G_B も存在すると言う規則であり，$G_A \Rightarrow G_B$ と表記する．確信度とは，相関ルールが成立する強さ表す値であり，大半のイディオムの実装は正しく行われるという仮定の下検出を行う．検出手法の概要を図 2 に挙げた．詳細を以下で説明する．

手順 1　相関ルールを構築する．頻出部分グラフ G_A に対し，G_A の頂点を全て含む頻出部分グラフ G_B を探し，$G_A \Rightarrow G_B$ という相関ルールを構築する．G_A は G_B の部分グラフである．

手順 2　相関ルール $G_A \Rightarrow G_B$ の確信度を計算する．確信度はプログラム依存グラフ P 中に G_A が存在したとき，同じプログラム依存グラフ中に G_B も存在する条件付き確率である．プログラム依存グラフ全体の集合を H とするとき，H 中で頻出部分グラフ G が出現するプログラム依存グラフの集合を $O(G,H)$，集合の数を $S(O(G,H))$ と表すと，確信度 $C(G_A \Rightarrow G_B)$ の計算は以下の式により行われる．

$$C(G_A \Rightarrow G_B) = \frac{S(O(G_A,H) \cap O(G_B,H))}{S(O(G_A,H))} \qquad (1)$$

このように，確信度は確率であるため，0 ～ 1 の実数値をとる．

手順 3　確信度の値が 1 ではなく，かつ十分 1 に近い閾値以上の時，G_A の出現するプログラム依存グラフのうち，相関ルールを満たさないものを欠陥候補とする．すなわち，G_A は出現するが，G_B が出現しないプログラム依存グラフが候補となる．欠陥の原因は，G_B の頂点の集合と G_A の頂点の集合との差分の頂点の集合が，欠陥候補となる G_A において欠落していることである．十分に近いと判断する閾値は，ユーザが設定した値である．

(3) について，“確信度の値が 1 ではなく，かつ十分 1 に近い値” の表す意味を述べる．相関ルール $G_A \Rightarrow G_B$ において，G_A は G_B の部分グラフである．言い換えると，G_B の頂点と G_A の頂点との差分となる頂点が，G_A において欠落している．確信度は G_A が出現するプログラム依存グラフにおいて G_B が出現しない割合を表している．0 に近い値では欠落のある頻出部分グラフ G_A が多く出現し，1 に近い値では欠落が稀に出現することを表し，1 では欠落は出現しないことを表す．大半のイディオムの実装は正しく行われるという仮定から，より稀に出現する欠落がある頻出部分グラフほど優先して検査すべき欠陥候補と考えられる．

直観的に提案手法は次のように説明される．図 3 は，イディオム $A(open() \rightarrow read() \rightarrow close())$ が多数のソースコード中に出現するのに対し，メソッド呼び出し $close()$ が欠落したイディオム $A'(open() \rightarrow read())$ のみが出現するソースコードが 1 つのみある状況を表している．この状況では，$close()$ の呼び出しが欠落しており，それによって欠陥が生じていると考えられる．相関ルールの確信度は，イディオム A' のみ出現するソースコードの稀少さを表しているといえる．イディオム A' のみ出現するソースコードが少数であるほど，イディオム A が正しい実装であり，一方でイディオム A' が誤った実装である可能性が高いと考えられる(注2)．

4. 使 用 例

提案手法を Java 言語を用いて試作し，適用実験を行った．対象にはオープンソースプロジェクトの MASU [6] [10] を用いた．MASU の規模は表 1 に記した．なお，欠陥候補とみなす閾値は 0.90 に設定した．

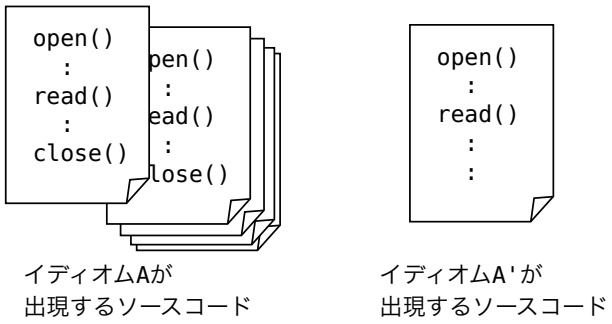


図 3　イディオムと欠陥候補の例

リビジョン	行数	ファイル数	メソッド数
1354	100574	521	4345

表 1　MASU の規模

頻出部分グラフ	欠陥候補	欠陥候補のうち欠陥を含むもの
893	27	2

表 2　確信度の閾値 0.90 での検出結果

その結果，893 個の頻出部分グラフから合計 27 個の欠陥候補が検出され，そのうち 2 個の欠陥候補が 1 つの欠陥を指していた (表 2)．つまり，異なる 2 個の頻出部分グラフが頂点を共有しており，共有している頂点が欠落していた．欠陥候補のうち片方を図 4 に挙げる．なお，図 4 に挙げた欠陥候補の確信度は 0.93 であった．

図 4 では，図 4(a) が欠陥であり，図 4(b) が正しい実装を表している．欠陥の原因は，図 4(b) の中で破線で囲った部分が，4(a) 内で欠落していることによる．破線部は 2 つの役割に分かれている．1 つめはメソッド `resolve()` が不正なスレッドから呼び出されないかチェック，2 つめは引数のチェックを行う役割である．これらの欠落により，例えば引数に不正な値が渡されて，プログラムが予期せぬ動作をする可能性が考えられる．一連のチェックは，すべてのメソッド `resolve()` の中でまず初めに行われるべきである．これは MASU 固有のイディオムであり，プロジェクト固有のイディオムから欠陥を発見できたといえる．

5. 関 連 研 究

本節では，イディオムからの欠陥検出に関連する研究を述べる．

Li らはシーケンシャルパターンマイニングを用いて，コピーアンドペースト後の識別子の変更漏れによる欠陥を検出する手法を考案した [4]．シーケンシャルパターンマイニングとは，順序を考慮する系列 (シーケンス) から，頻出する部分系列を抽出する手法である．まず，シーケンシャルパターンマイニングによりコピーアンドペーストから生成されたコード片の抽出を行い，次に抽出されたコード片について，識別子の変更率を基準に欠陥の候補を挙げる．直観的には，ペーストされたコード片中で，一部変更漏れがあるコード片を欠陥候補とするといえる．この手法を C 言語で記述された Linux カーネルのソース

(注2)：注意すべきなのは，イディオム A' はイディオム A に含まれるため，イディオム A が出現するソースコード中にはイディオム A' にも含まれることである．したがって，相関ルールの確信度は “誤った実装に対する正しい実装の割合” ではなく，“誤った実装と正しい実装を合わせた数に対する正しい実装の割合” となる．

```
public BinominalOperationInfo resolve(/* 省略 */) {
    if (this.alreadyResolved()) {
        return this.getResolved();
        }
/* 省略 */
    final int fromLine = this.getFromLine();
    final int fromColumn = this.getFromColumn();
    final int toLine = this.getToLine();
    final int toColumn = this.getToColumn();
/* 省略 */
    return this.resolvedInfo;
}
```

(a) UnresolvedBinominalInfo.java から検出した欠陥

```
public CatchBlockInfo resolve(/* 省略 */) {
   MetricsToolSecurityManager.getInstance().checkAccess();
   if ((null == usingClass) || (null == usingMethod)
        || (null == classInfoManager) || (null == methodInfoManager)) {
       throw new NullPointerException();
   }
    if (this.alreadyResolved()) {
        return this.getResolved();
    }
/* 省略 */
    final int fromLine = this.getFromLine();
    final int fromColumn = this.getFromColumn();
    final int toLine = this.getToLine();
    final int toColumn = this.getToColumn();
/* 省略 */
    return this.resolvedInfo;
}
```

欠陥候補にて
欠落していた部分

(b) UnresolvedCatchBlockInfo.java 他 13 箇所で出現する正しい実装

図 4　MASU から検出された欠陥

コードなどに適用し，Linux カーネルからは 49 個の欠陥を発見した．

同じく Li らはアイテムセットマイニングを用い，得られた頻出部分集合の違反から欠陥候補を検出する手法を考案した [5]．PR-Miner はこの手法を実装したツールであり，Linux カーネルのソースコードに対し適用し， 16 個の欠陥を検出した．アイテムセットマイニングとは，複数の要素をもつもの (アイテムセット) の集合から，頻出する部分集合を抽出するデータマイニングの一種である．Li らの手法では，まずアイテムセットマイニングをメソッド定義中のメソッド呼び出し文などから構成されるアイテムセットの集合に適用し，同時に出現することの多いメソッド呼び出しの集合をイディオムとして抽出する．そして，抽出した頻出部分集合のうち一部が欠落した頻出部分集合が少数であれば，それを欠陥候補とする．本手法とイディオムの抽出対象は異なるものの，イディオムから欠陥候補を検出する方法は同じである．しかし，順序を考慮しないために，欠陥の検出精度が下がるという指摘が Kagdi らによりなされている [2]．

Wasylkowski らはオブジェクトの使用方法を表す Object Usage Model というグラフ表現提唱し，それ従わないコード片が少数であった場合に欠陥候補とする手法を考案した [8]．具体的には，すべてのメソッドについて，メソッド定義中でオブジェクトに関した文を要素とする，制御フローを考慮したグラフを構築する．次にそれぞれのグラフについて，特定の 1 オブジェクトに着目したグラフを制御構造を保ちながら抜き出す．このグラフが Object Usage Model である．Object Usage Model の集合から，2 つのメソッド呼び出しのペアで頻出するものを，時系列考慮して抽出する．最後にそれらペアをに基づき，Object Usage Model で欠落したペアを探し，欠陥候補とする．この手法で Wasylkowski らは AspectJ のソースコードから 2 個の欠陥を発見した．Object Usage Model はメソッドの実行順序を考慮するため，PR-Miner よりも検出精度が向上すると考えられるが，一方で単一オブジェクトの使い方に特化したことにより，検出できない欠陥も生じる可能性がある．

Nguyen らは，Groum というオブジェクトの使い方を表すグラフ表現を提唱し，その違反から欠陥を検出する手法を考案した [7]．Groum も Wasylkowski ら Object Usage Model と同じくオブジェクトの使い方を表現したグラフ表現である．Groum の特徴は次の 3 点である．(1) 複数のオブジェクトを考慮できる．(2) データ依存関係をもつ． (3) スコープも考慮する．こ

のように，以前の手法に比べ多くの情報を用いたモデルを用いることによって，これまでは検出できなかった欠陥を検出できるのが利点である．情報量の増加により，計算時間の増加は免れないが，ヒューリスティックにより計算量を削減し，現実的な時間内での実行を可能にしている．Nguyen らはこの手法をいくつかのオープンソースプロジェクトのソースコードに適用し，欠陥の検出に成功した．この手法は Wasylkowski らの手法に比べ多くの情報を用いているものの，複数オブジェクト間の依存関係がヒューリスティックを用いて検出されているなど，計算量を削減するための手順で精度が落ちていると考えられる．例えば，本来であれば実行順序関係を持たない 2 文間に辺が引かれることが挙げられる．本手法では，プログラム依存グラフを用いることで，実行順序関係をもたない 2 文間に辺が引かれることはない．

6. まとめと今後の課題

本研究では，プログラム依存グラフから頻出する部分グラフを抽出し，抽出した頻出部分グラフに基づいて欠陥検出を行う手法を提案した．プログラム依存グラフを用いることで，従来手法では検出できなかった欠陥が検出できると考えられる．手法を実現したツールを試作し，オープンソースプロジェクトに適用したところ，実際に欠陥を検出することができ，これにより欠陥の検出能力が確認できた．

今後の課題は大まかに 2 つある．1 つめは評価実験，もう 1 つはプログラム依存グラフの欠陥検出に向けた最適化である．

本研究では，1 つのプロジェクトに対してのみ手法を適用したのみであり，十分な評価を行えたとは言いがたい．現在，評価実験として複数のオープンソースプロジェクトに対しての手法の適用を考えている．しかし，最新のリビジョンから見つかった欠陥候補が，本当に欠陥であると断定することは難しい．そこで，以下のような手順での評価を考えている．(1) ソフトウェアプロジェクトのリポジトリから，コミットログを参考にして欠陥の修正が行われたリビジョンを特定する．これを対象プロジェクト中の欠陥の母数とする．(2) 本手法を修正が行われたリビジョンと，その直前のリビジョン両方に適用する．ここで，直前のリビジョンで欠陥候補となった頻出部分グラフが，直後のリビジョンでパターン違反でなくなっていれば，欠陥として確定できる．(3) 確定した欠陥が，(1) で特定した欠陥の母数に対しどの程度の割合があるか評価する．

プログラム依存グラフの構築，さらにそこからの頻出部分グラフの抽出は非常に高コストな計算である．しかし，本手法ではプログラム依存グラフのすべての情報を用いているわけではない．そこで，評価実験の結果などから余分な情報を削減し，より高速な検出を実現したいと考えている．

謝辞

本研究を進めるにあたり，貴重なご意見をいただきました大阪大学 松下誠 准教授に感謝いたします．また，本研究で用いた Scorpio の提供をはじめ，様々なご協力をいただきました同大学 肥後芳樹 助教に感謝いたします．

本研究は一部，文部科学省「次世代 IT 基盤構築のための研究開発」の委託に基づいて行われた．また，日本学術振興会科学研究費補助金基盤研究（A）（課題番号:21240002）の助成を得た．

文　献

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of VLDB*, pp. 487–499, 1994.

[2] H. Kagdi, M. L. Collard, and J. I. Maletic. Comparing approaches to mining source code for call-usage patterns. In *Proc. of MSR 2007*, pp. 123–130, 2007.

[3] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Proc. of ISESE 2004*, pp. 83–92, 2004.

[4] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, Vol. 32, No. 3, pp. 176–192, 2006.

[5] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. of ESEC/FSE 2005*, pp. 306–315, 2005.

[6] MASU. `http://sourceforge.net/projects/masu/`.

[7] T.T. Nguyen, H.A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proc. of ESEC/FSE 2009*, pp. 383–392, 2009.

[8] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proc. of ESEC/FSE 2007*, pp. 35–44, 2007.

[9] パターンワーキンググループ. ソフトウェアパターン入門. ソフト・リサーチ・センター, 2005.

[10] 三宅達也, 肥後芳樹, 井上克郎. メトリクス計測プラグインプラットフォーム MASU の開発. ソフトウェアエンジニアリング最前線 2008, pp63-70, pp. 63–70, 2008.

[11] 肥後芳樹, 楠本真二. 実規模ソフトウェアへの適用を目的としたプログラム依存グラフに基づくコードクローン検出法. ソフトウェアエンジニアリング最前線 2009, pp. 97–104, 2009.