修士学位論文

題目

プログラム依存グラフを用いたソースコードのパターン違反検出法

指導教員
井上 克郎 教授

報告者
山田 吾郎

平成 23 年 2 月 7 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 22 年度 修士学位論文

プログラム依存グラフを用いたソースコードのパターン違反検出法

山田 吾郎

内容梗概

イディオムとは，特定の処理を実現するための実装のパターンである．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．イディオムを実装する際，様々な理由により欠陥が混入し得る．

これまでに，このような欠陥を計算機を用いて検出する手法が考案されてきた．これらの手法は，検出プログラムに対して利用者の支援が必要か否かによって大まかに 2 種類に分類できる．利用者の支援が必要な手法は概して誤検出が少なく計算コストも低いが，検出できる欠陥の種類は利用者が与えた欠陥に限られてしまう点が問題がある．それに対し利用者の支援を必要としない手法では，検出プログラムが欠陥のパターンを自動で構築するため，利用者が知らない欠陥も発見することが可能である．

このような利用者の支援を必要としない手法では，欠陥のパターンの構築にかかるコストが非常に高く，また誤検出が多い．そこで既存手法では，高速化のため欠陥のパターン構築に利用する情報を制限してきた．また，誤検出を減少させるため欠陥とみなす条件である確信度という値の閾値を厳しく設定してきた．しかし，これらにより検出できない欠陥が生じてしまうと考えられる．

本研究では，欠陥の検出対象にプログラム依存グラフを用い，さらに欠陥候補に関連するメトリクスを利用することで問題の解決を試みた．プログラム依存グラフはプログラム中の文を頂点とし，文間に生じる複数の関係を辺としてもつ有向グラフである．これらの情報を考慮することで検出可能な欠陥が増加すると考えられる．また，従来手法に比べ欠陥の検出条件である確信度の閾値を緩めた．これにより誤検出が増加するが，新たに提案したメトリクスによるフィルタリングを行うことで誤検出の増加を抑えながら，従来では検出できなかった欠陥の検出が可能になると考えられる．

評価実験として，本手法を 8 個のオープンソースプロジェクトに適用した．それぞれのプロジェクトについて検出結果をフィルタリングし，並べ替えを行なったところ合計 6 個の欠陥と，7 個の修正を必要とする可能性がある候補が含まれていた．その中の 1 個は従来手法で原理的に検出できない箇所であり，13 個全てが従来の検出条件で用いられた確信度では検出できない箇所であった．

主な用語

プログラム依存グラフ
欠陥検出
パターン違反

目 次

1　まえがき

イディオム [14] とは，特定の処理を実現するための実装のパターンである．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 [8]．

イディオムの実装は常に正しく行われるとは限らない．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 [9]．

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のパターンをイディオムとして抽出し，パターン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 [10]．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，Nguyen らは複数オブジェクトの使用方法のパターンをイディオムとして抽出し，それに従わないコード片を漸増的に検出する手法を考案した [11]．Nguyen らはこの手法を 9 つのプロジェクトの適用し，計 10 個の欠陥を発見した．

このような利用者の支援を必要としない手法では，欠陥のパターンの構築にかかる計算コストが非常に高く，誤検出が多いという問題がある．既存手法では，計算コストの問題に対して，欠陥のパターンの構築に用いるデータ構造の簡略化により現実的な時間での検出を達成した．また誤検出については，検出の条件となる確信度という値の閾値を厳しく設定することで欠陥候補の数を絞ることで対処してきた．しかし，そのいずれもが検出可能な欠陥の減少を伴うと考えられる．

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違反を検出，さらに違反に関する複数のメトリクスでフィルタリングすることで，これら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・データ依存を表現するグラフである．従来に比べ多くの情報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．また，欠陥とみなす条件を従来手法に比べ緩く設定しておき，複数のメトリクスを用いてフィルタリングを行うことで従来では見つからなかった欠陥が検出可能になる．

提案手法は以下の 4 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コード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，相関ルールマイニングというデータマイニング手法を適用する．これにより頻出部分グラフに違反しているプログラム依存グラフを検出し，欠陥候補とする．

ステップ 4 検出された欠陥候補に対し，複数のメトリクスによってフィルタリングを行い，利用者に提示する．

ステップ 1，**ステップ 2** は肥後らの作成したツール **Scorpio**[1] [15] を元に，本手法に適したように調整を行った手法を用いる．

評価実験として，本手法を 8 個のオープンソースプロジェクトに適用し，GrouMiner との比較を行なった．各プロジェクトの検出結果に対してフィルタリングと並べ替えを行い，それぞれの上位 15 候補を調べたところ，合計で 6 個の欠陥と 7 個の修正を必要とする箇所を含んでいた．その中の 1 個は従来手法では使用するデータ構造では検出できない箇所であり，また 13 個全ての候補が従来の検出条件に用いた確信度では見つからないものであった．

以降，2 節ではパターン違反を用いた欠陥検出について，既存手法で共通する概念の説明を行う．また既存手法の具体的な説明と，それらに共通して見られる問題点を提示する．3 節では提案手法の詳細を解説し，4 節でオープンソースプロジェクトを対象とした適用実験について述べ，その考察を行う．最後に 6 節でまとめと今後の課題を述べる．

[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

2　背景

1 節で述べたように，イディオムの実装が誤って行われる可能性がある．これまでに，イディオムの欠陥を検出するための手法がいくつか提案されてきた．このような欠陥検出手法は，検出対象以外の情報の有無によって大きく 2 種類に分類できる．

まず，検出対象以外に情報が与えられる手法には FindBugs[6] や PMD[12] があげられる．これらは，開発者および利用者がイディオムから起こり得る欠陥を事前に欠陥のパターンとして用意しておき，欠陥のパターンに該当する部分を欠陥候補として検出する手法である．これらの手法は欠陥のパターンが既知であるため，未知の手法に比べると高速である．しかし，用意された欠陥のパターンに該当する欠陥しか検出できない点が問題である．したがって，あるソフトウェアに固有のイディオムについて，その欠陥を検出するためには利用者がパターンを用意する必要がある．この場合，そもそもイディオムが明文化されているとは限らず，パターンを用意することそのものが困難である場合も考えられる．

一方，検出対象のみを用いた欠陥検出手法もいくつか研究が行われており，2.4 節で説明を行う．これらの手法はイディオムの抽出および欠陥のパターンの構築を自動で行うため，ソフトウェア固有のイディオムも検出できる利点をもつ．

以降ではこのような手法を自動欠陥検出とよぶことにする．自動欠陥検出手法は大きく次の 3 つの段階によって行われる．

1. 検出対象のモデル化
2. イディオムに該当する**パターン**をモデル上で抽出
3. パターンに違反したモデルを欠陥候補として検出

ここでパターンとは，モデル上でのイディオムに対応する概念である．つまり，検出対象に対するイディオムと，モデル上でのパターンとは同じ関係を表している．これらの関係を図 1 にあげた．

本節では以後，自動欠陥検出の概要について説明する．まず 2.1 節にてモデル化の概要を俯瞰したのち，2.2 節にてパターンの抽出方法を，2.3 節にて欠陥の検出方法の概要を述べる．さらに自動欠陥検出の具体例として 2.4 節でこれら手法の既存研究を取り上げ，2.5 節で関連研究での問題点を提示する．

2.1　検出対象のモデル化

自動欠陥検出では，検出対象から直接イディオムを抽出するのではなく，一旦モデル化を行い，モデル上でのイディオムであるパターンを抽出することが一般的である．モデル化の目的は 2 つ考えられる．

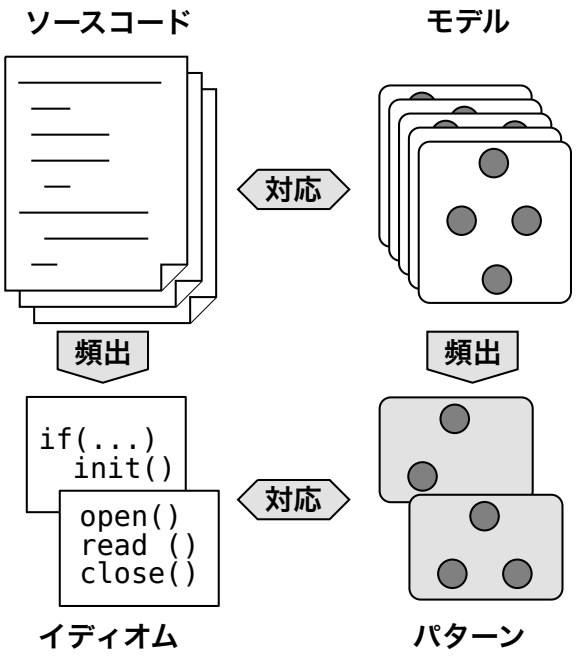


図 1: ソースコード，イディオム，モデル，パターンの対応関係

まず 1 つ目は，計算の高速化である．イディオムおよび欠陥のパターンが与えられる手法に比べ，自動欠陥検出ではイディオムから抽出する必要がある．このような手法は計算コストが非常に高い．そこで，最小限の情報のみを残しモデル化を行うことで計算コストを低下させる．例えば PR-Miner[10] はメソッド定義中で呼び出されているメソッド群の集合という単純なモデルを用いることで高速な検出を実現している．

2 つ目は，検出の目的に沿った抽象化である．検出の目的に必要な情報をできるだけ多く取得する一方，目的に関係のない情報を取り除くことで，より高い精度・効率での検出が可能になると考えられる．例えば JADET[13] は，単一オブジェクトの使用方法に違反したコードを検出する目的でモデルを考案した．このモデルは 1 つのオブジェクトの使用箇所のみを抽出する代わりに，制御フローなどを考慮した情報を含む特徴をもつ．

2.2 パターンの抽出

構築されたモデルからのパターン抽出には頻出パターンマイニングが用いられる．頻出パターンマイニングとはデータマイニングの一種であり，何らかのデータ構造の集合から頻出する部分構造を抽出する手法全体の総称である [5]．抽出された頻出する部分構造がパターンである．

データの集合をトランザクションとよぶとき，トランザクションの集合に対しデータマイニングを行うことで，一定回数以上出現する部分集合などをパターンとして抽出する．また，

トランザクション中でパターンに該当する部分をパターンのインスタンスとよぶ．逆に，インスタンスを抽出したものがパターンとも言える．自動欠陥検出では，構築したモデルがトランザクションにあたる．

2.1 節で取り上げた PR-Miner は単一メソッド中で呼び出されているメソッド群がトランザクションにあたり，アイテムセットマイニング [4] とよばれる，頻出する部分集合を抽出する頻出パターンマイニングを用いる．また，JADET ではトランザクションが単一オブジェクトの使用方法を表すグラフであり，頻出する部分グラフを抽出する手法を用いている．

2.3 欠陥の検出

抽出したパターンからの欠陥検出は**相関ルール** [1] とよばれる規則から，パターン違反とよばれるモデルを検出することで行われる．以下 2.3.1 節で相関ルールの構築方法を，2.3.2 節で相関ルールからのパターン違反検出方法を述べる．

2.3.1 相関ルールの構築

相関ルールとは，トランザクション T においてパターン P_1 が存在したとき，パターン P_2 も存在すると言う規則で，$P_1 \Rightarrow P_2$ と書く．

パターン違反の検出を目的とする場合，P_1，P_2 には以下のような包含関係を満たす 2 つを選ぶ．

$$P_1 \subset P_2 \tag{1}$$

P_1，P_2 がこのような関係にあるとき，P_1 を P_2 のサブパターンとよぶことにする．例えば，トランザクションが集合の場合はサブパターン P_1 は P_2 の真部分集合であり，グラフの場合は P_1 は P_2 の部分グラフとなる．

ここで，式 1 の関係を満たす 2 つのパターンは次の関係を満たす．ただし $T(P)$ はパターン P が出現するトランザクションの集合とする．

$$T(P_2) \subseteq T(P_1) \tag{2}$$

これは，以下の理由から必ず成立する．

- P_1 は P_2 の真部分集合であるから，P_2 が出現するトランザクションには，P_2 の部分集合として必ず P_1 も出現する．これは自明である．
- P_2 を構成する要素と P_1 を構成する要素の差分となる要素が存在する．これらの要素が P_1 の出現するトランザクションで必ず出現するとは限らない．したがって，逆に P_1 が出現するトランザクション全てに P_2 が出現するとは限らない．

以上より $T(P_2)$ は $T(P_1)$ の部分集合になる．

2.3.2 パターン違反の検出

パターン違反は相関ルールの**確信度**という値を用いて，パターンが多数出現するという仮定の下で検出する．確信度は相関ルール $P_1 \Rightarrow P_2$ が成立する強さを表す値であり，P_1 が出現したトランザクションで P_2 も出現する条件付き確率で定義される．

$$\text{確信度} = \frac{|T(P_1) \bigcap T(P_2)|}{|T(P_1)|} \tag{3}$$

したがって確信度は 0.0 ～ 1.0 の間で表される．

相関ルールの確信度が 1.0 ではない十分大きな閾値を越えるとき，サブパターン P_1 のみが出現するトランザクション $T(P_1) - T(P_2)$ を，P_2 に対するパターン違反とする．その理由は P_2 を構成する要素と P_1 を構成する要素の差分となる要素の欠落である．なお，閾値は利用者が決定する．

以降で，確信度に制限を設ける理由を述べる．まず確信度が 1.0 の相関ルールを除く理由は，確信度 1.0 が $T(P_1) - T(P_2) = \phi$ を意味するからである．

次に下限を設定する理由を述べる．確信度が小さいとき $T(P_2)$ に比べ $T(P_1) - T(P_2)$ の比率が大きくなる．このようなとき，$T(P_1) - T(P_2)$ は違反というよりは，そのようなトランザクションもパターンとして存在すると考えるほうが妥当である．逆に，確信度が高いと $T(P_1) - T(P_2)$ が非常に少なくなり，欠落が稀に起っているといえ，実際に違反であると考えるほうが妥当になる．

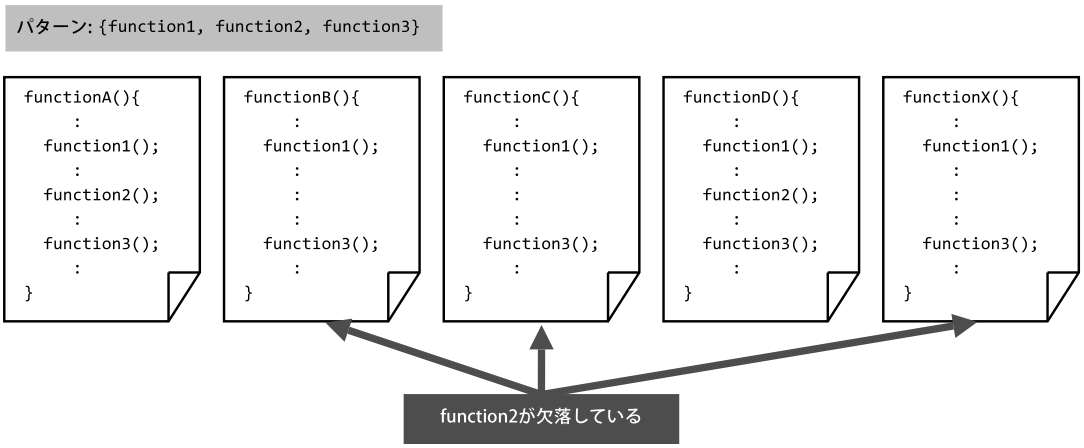
下限を設ける理由について PR-Miner の例を用いて説明する．図 2 に模擬コードによる説明をあげた．

図 2(a) では，パターン $\{function1, function2, function3\}$ がトランザクション $\{functionA\}$, $\{functionD\}$ の計 2 箇所で見つかっており，そのサブパターン $\{function1, function3\}$ のみが $\{functionB\}$, $\{functionC\}$, $\{functionX\}$ の 3 箇所で出現している．$\{function1\}$, $\{function3\}$ のみを使う場合もあれば，その間に $\{function2\}$ を呼び出す用例もあると考えることができる．

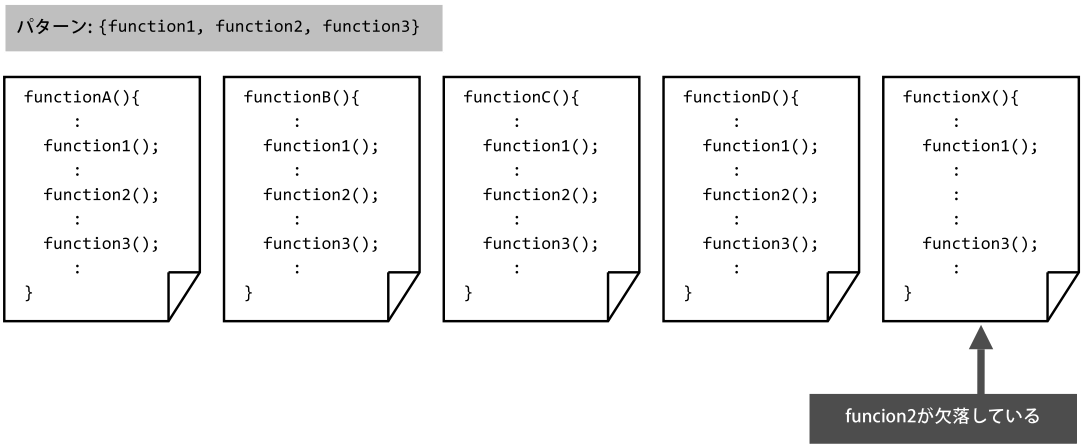
一方，図 2(b) では，パターン $\{function1, function2, function3\}$ がトランザクション $\{functionA\}$, $\{functionB\}$, $\{functionC\}$, $\{functionD\}$ の計 4 箇所で見つかっているのに対し，そのサブパターン $\{function1, function3\}$ のみが出現するトランザクションは $\{functionX\}$ の 1 箇所である．この場合 $\{functionX\}$ でのサブパターンは $\{function2\}$ の欠落による欠陥が生じている可能性が高いと考えることができる．

2.4 既存手法

本節では，自動欠陥検出に関連する研究を述べる．



(a) 欠陥である可能性が低い場合



(b) 欠陥である可能性が高い場合

図 2: 模擬コードによるパターン違反の例

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 [9]．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あげる．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手法といえる．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 [10]．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パターンとして抽出する．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．本手法とパターンの抽出対象は異なるものの，パターン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 [7]．

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 [13]．具体的には，全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，特定の一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，時系列を考慮して抽出する．最後にそれらのペアに基づき，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 [11]．Groum も Wasylkowski らの 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が，PDG に近似することを目的に様々な情報が付加されている．Groum の特徴は次の 3 点である．

- 複数のオブジェクトを考慮できる
- データ依存関係をもつ
- スコープを考慮する

このように，以前の手法に比べ多くの情報を用いた Groum という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．

2.5 既存手法での問題点

2.4 節で過去に行われた自動欠陥検出の手法を述べたが，大きく 2 つの問題が考えられる．

文の抽象化による問題 2.1 節で述べたように，いずれの手法もモデル化の段階で使用する情報を制限している．このことが，検出できる欠陥の減少や誤検出の増加につながると考えられる．

Nguyen らの考案したモデルである Groum は，文中で使用されている変数などの意味情報をもちながら複数オブジェクト間の相関関係を表現できる情報量の多いモデルである．しかし，オブジェクトのみに着目しておりプリミティブ型変数は無視されているため，それらを含むパターンは抽出できない．また，複数オブジェクト間のデータ依存関係の取得にヒューリスティックを用いているため，必ずしも正確なデータ依存辺ではなく，誤検出が増加する可能性がある．

確信度の閾値を高く設定したことによる問題 2.3.2 節で説明したように，従来の自動欠陥検出手法はパターンが多くの箇所で出現するという前提に立っている．というのも，少数しか出現しないパターンを含めると誤検出が膨大になるためである．そのため，実験では確信度の閾値が高く設定されている．しかし，パターンが多く出現しなければならないということは，少数しか出現しないパターンを検出できないことを意味している．

例えば Li らの PR-Miner や，Nguyen らの GrouMiner はともに適用実験では閾値を 0.9 に設定している．このとき，相関ルール $G_A \Rightarrow G_B$ が欠陥候補として検出されるためには，G_B のインスタンスが 10 以上存在していなければならない．インスタンス数が 9 以下の場合，そもそもパターンとして抽出されないため検出不能である．

3　提案手法

本研究では，2.5 節でであげた問題を解決するために，プログラム依存グラフをモデルとして用いた自動欠陥検出方法，および欠陥候補に関連するメトリクス群を提案する．

まず，モデル化の段階で情報を制限する問題を，プログラム依存グラフを用いることで解決する．プログラム依存グラフとはプログラム中の文を頂点とし，文間に生じる関係を表す有向グラフであり，メソッド単位で定義される．文間の関係は制御依存，データ依存を表す複数の辺により表現されており，従来手法で提案された簡略化されたグラフ構造に比べ多くの情報をもっている．そのため，従来手法では情報の欠落により検出できなかった欠陥を検出できる可能性がある．

次に，確信度の閾値を高く設定する問題に対して，欠陥候補に関するメトリクスを複数用い，フィルタリング・順位付けを行うことで解決をはかる．既存手法に比べ確信度の閾値を小さい値に設定し，代わりにその他のメトリクス群を併用してフィルタリングを行うことで，従来であれば検出できなかったインスタンス数の少ないパターンによる欠陥候補の検出を可能にしながら，誤検出の数を抑えることができると考えられる．

手法の概要を図 3 にあげた．手法は次の 4 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コード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，相関ルールマイニングというデータマイニング手法を適用する．これにより頻出部分グラフに違反しているプログラム依存グラフを検出し，欠陥候補とする．

ステップ 4 検出された欠陥候補を 5 つのメトリクスによってフィルタリングし，利用者に提示する．

ステップ 1，**ステップ 2** は肥後らの作成したツール Scorpio[15] に基づいて行う．以降では，まず 3.1 節で Scorpio を用いたパターンの抽出手法を述べ，続く 3.2 節で相関ルールによる欠陥候補の検出方法を説明する．最後に 3.3 節で検出された欠陥候補のフィルタリング・並べ替えの基準になるメトリクスを紹介する．

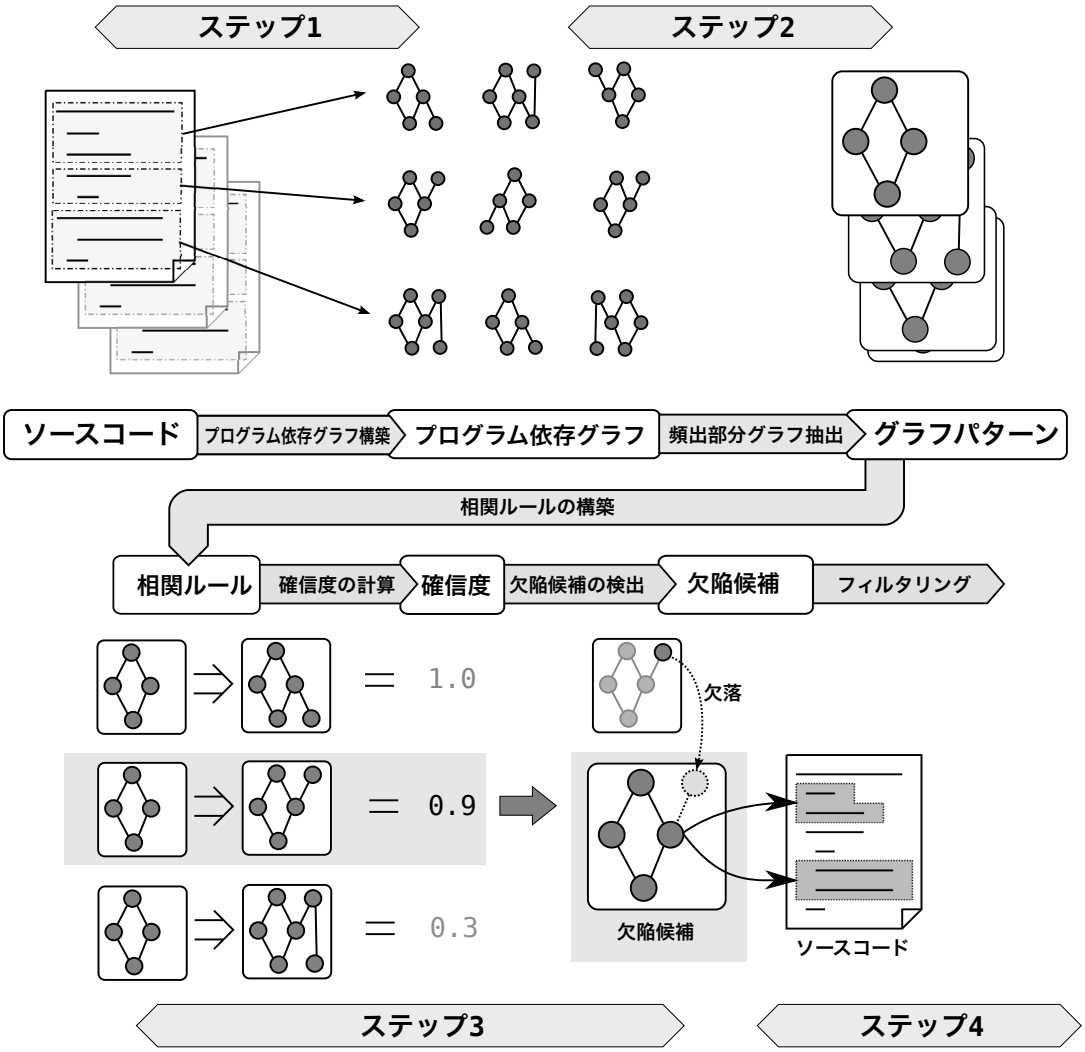


図 3: 提案手法の概要

3.1 Scorpio を用いたイディオムの抽出

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中で頻出する部分グラフである．以下では抽出した頻出部分グラフを**グラフパターン**とよぶ．逆に，プログラム依存グラフ中で，あるグラフパターンに対応する部分グラフをそのグラフパターンの**インスタンス**とよぶ．これらの関係を図 4 にあげた．

グラフパターンの抽出には肥後らの開発したツール Scorpio を欠陥検出に適した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制御フローグラフとよばれるグラフを構築し，そのグラフを基にプログラム依存グラフを構築する．そして，構築したプログラム依存グラフからグラフパターンを抽出する．図 5(b) は図 5(a) から構築される制御フローグラフを，図 5(c) は図 5(a) から構築されるプログラム依存グラフをそれぞれ表

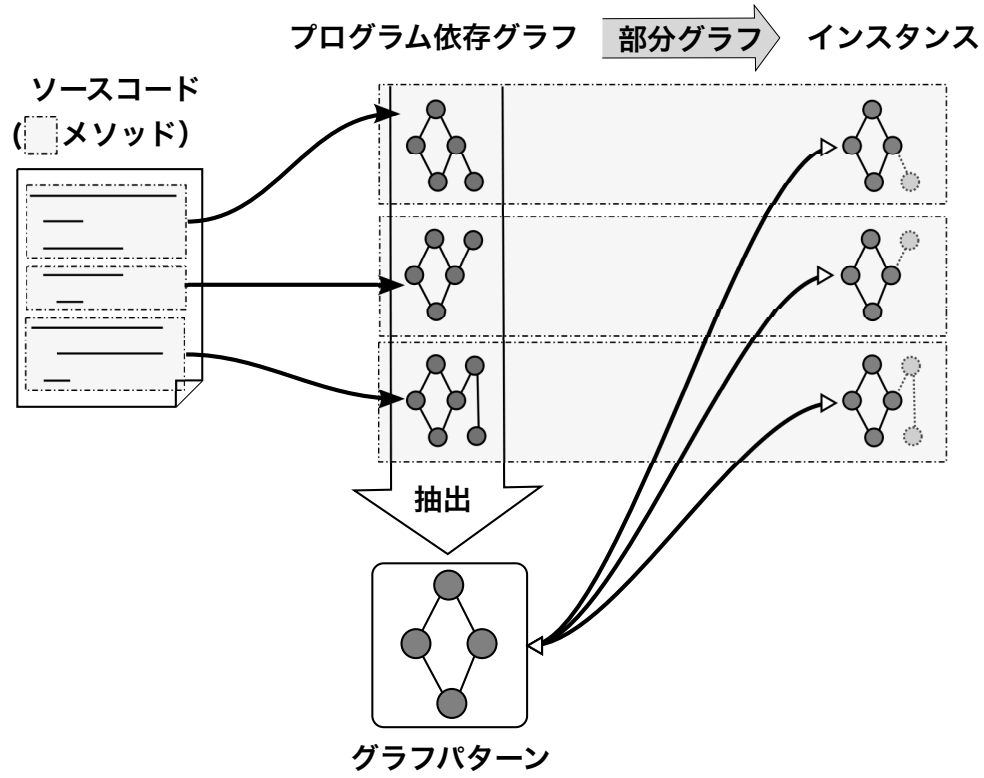


図 4: ソースコード，グラフパターン，インスタンスの関係

している．

また，Scorpio でのプログラム依存グラフの頂点・辺の詳細な定義は以下のとおりである．

頂点 文と条件節中の式を表す．例えば Java 言語での for 文の条件節からは 3 つの頂点が生成される．

辺 制御依存辺・データ依存辺の 2 種類の辺がある

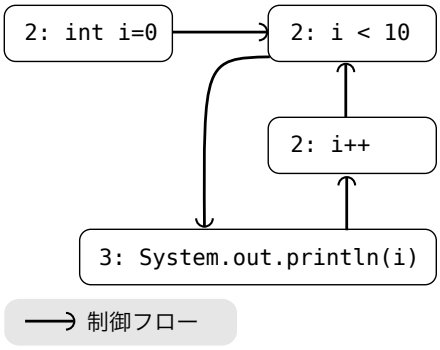
制御依存辺 頂点 A, B について，頂点 A が条件式であり頂点 B が実行されるか否かが頂点 A の結果に依存していることを表す．このとき，頂点 A から頂点 B へ辺を引く．

データ依存辺 頂点 A, B について，頂点 A で定義された変数が頂点 B で参照されていることを表す．このとき，頂点 A から頂点 B へ辺を引く．

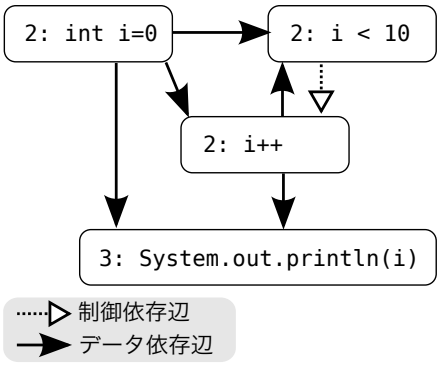
以降，3.1.1 で制御フローグラフの構築方法を，3.1.2 節でプログラム依存グラフの構築方法を説明し，3.1.3 節でグラフパターンの抽出方法を述べる．

```
1: void method(){
2:   for (int i = 0; i < 10; i++){
3:     System.out.println(i);
4:   }
5: }
```

(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 5: 制御フローグラフとプログラム依存グラフの例

3.1.1 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 で用いる制御フローグラフの頂点・辺の定義は以下のとおりである.

頂点 文と条件節中の式. 図5(a)では，文`System.out.println(i)`，式`i < 10`，`int i = 0`，`i++`の計 4 つが頂点となる.

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.

3.1.2 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．

3.1.3 プログラム依存グラフからのグラフパターン抽出

本節では，3.1.2 節で述べたプログラム依存グラフからグラフパターンを抽出する手法について述べる．本研究ではグラフパターンがソースコード上でのイディオムに対応する．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もつ．

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．このペアはグラフパターンの候補となる．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もつ場合．

条件 2 $(p1, pr2)$ 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 $(p1, pr2)$ 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 で検出したグラフパターンの候補 $(s1, s2)$ が他の候補 $(s1', s2')$ に含まれていた場合 ($s1 \subseteq s1'$かつ $s2 \subseteq s2'$)，その候補を検出されたグラフパターンの候補の集合から削除する.

ステップ 4 同部分グラフをもつ候補からグラフパターンを形成する．例えば，2 つの候補 $(s1, s2), (s2, s3)$ があった場合，グラフパターンとして $\{s1, s2, s3\}$ が形成される.

3.2 グラフパターンからの欠陥候補の検出

本節では，3.1 節で述べた手法により抽出したグラフパターンから，どのように欠陥候補を検出するか説明する.

グラフパターンからの欠陥候補の検出は，2.3 節で説明した相関ルールの確信度 $Conf$ を用いる．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，グラフパターン G_A が存在したとき，グラフパターン 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．詳細を以下で説明する.

手順 1 相関ルールを構築する．グラフパターン G_A に対し，G_A の頂点を全て含むグラフパターン 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．$N(G)$ がグラフパターン G のノード集合を表すとすると，G_A と G_B の関係は次のように定義される.

$$N(G_A) \subset N(G_B) \tag{4}$$

手順 2 相関ルール $G_A \Rightarrow G_B$ の確信度を計算する．確信度はある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 中でグラフパターン G が出現するプログラム依存グラフの集合を $O(G, H)$ と表すと，確信度 $Conf$ は以下の式により求まる.

$$Conf(G_A \Rightarrow G_B) = \frac{|O(G_A, H) \bigcap O(G_B, H)|}{|O(G_A, H)|} \tag{5}$$

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，かつある程度 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．したがって欠陥候補の集合 $Candidates$ は，グラフパターン G のインスタンスが出現するプログラム依存グラフの集合を $PDG(G)$ と書くとき，次のように表される．

$$Candidate = PDG(G_A) - PDG(G_B) \tag{6}$$

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．ある程度近いと判断する閾値は，使用者が設定した値である．

3.3 欠陥候補のフィルタリングに用いるメトリクス

3.1，3.2 節で説明した手法により欠陥候補が検出される．しかし，本研究では確信度の閾値を従来手法に比べ低い値に設定するため，比較的多くの誤検出が出現すると考えられる．

そこで，開発者が検出結果から効果的に欠陥を見つけるために，欠陥候補の相関ルールに関連するメトリクスを複数用意した．これらのメトリクスでフィルタリング・並べ替えを行うことで明らかな誤検出などを取り除くことができる．メトリクスは欠陥候補の相関ルールに対して定義されるものと，相関ルール $G_A \Rightarrow G_B$ の中で用いられるグラフパターン G_A, G_B それぞれについて定義されるものの 2 種類にわけられる．

定義したメトリクスを表 1 にあげた．以下では定義の詳細について説明を行う．いずれも例として相関ルール $G_A \Rightarrow G_B$ を用い，3.2 節での表記に基づいて記述する．

3.3.1 リフト値: Lift

相関ルールに対して定義される，G_A と G_B との間の独立度を表すメトリクス値である．この値は確信度を G_B のインスタンス数で割ることで求まる．リフト値 $Lift(G_A \Rightarrow G_B)$ は

メトリクス名	定義対象	値の範囲	解釈
リフト値 ($Lift$)	相関ルール	0 より大きい実数	高いほどよい
頂点欠落数 ($Missing$)	相関ルール	0 より大きい整数	低いほどよい
違反 PDG 数 ($Candcount$)	相関ルール	1 位上の整数	低いほどよい
頂点重複度 (Dup)	グラフパターン	1 以上の実数	低いほどよい
平均ギャップ長 (Gap)	グラフパターン	0 以上の実数	低いほどよい

表 1: フィルタリングに用いるメトリクスの一覧

次のように定義される．

$$Lift(G_A \Rightarrow G_B) = \frac{\text{確信度}}{|O(G_B, H)|} \tag{7}$$

リフト値が小さければ，式 7 での分母，つまり G_B のインスタンス数が大きいことを意味する．この場合，相関ルールの確信度が高い理由は G_A，G_B 間に強い相関関係があるからではなく，G_B のインスタンス数が多いからである．したがって相関ルールのリフト値が低い場合，誤検出の可能性が高い．

3.3.2 頂点欠落数: Missing

G_B に対し G_A で欠落している頂点の数を表すメトリクスであり，相関ルールに対して定義される．

3.2 節で説明したように，欠陥候補の相関ルールには G_A の頂点集合が G_B の頂点集合の真部分集合になるという条件がある．そのため，G_A の頂点数が少なく G_B の頂点数が多い場合，つまり頂点欠落数が大きいとき G_A が偶然 G_B に含まれてしまい，相関ルールが形成されやすくなる．したがって，頂点欠落数が大きい場合，誤検出の可能性が高い．

頂点欠落数 $Missing(G_A \Rightarrow G_B)$ は次のように定義される．ただし $N(G)$ はグラフパターン G に含まれる頂点の集合を表す関数である．

$$Missing(G_A \Rightarrow G_B) = N(G_B) - N(G_A) \tag{8}$$

3.3.3 違反 PDG 数: Candcount

欠陥候補となるプログラム依存グラフの数であり，相関ルールに対して定義される．同様の頂点の欠落が多くプログラム依存グラフに出現する場合，それは欠陥ではなく正しい実装の一種と捉えるほうが妥当であるという考えられる．したがって，違反 PDG 数が大きいほど誤検出である可能性が高い．

違反 PDG 数 $Candcount(G_A \Rightarrow G_B)$ は次のように定義される．ただし，$PDG(G)$ はグラフパターン G が出現するプログラム依存グラフの集合を表すものとする．

$$Candcount(G_A \Rightarrow G_B) = PDG(G_A) - PDG(G_B) \tag{9}$$

3.3.4 頂点重複度: Dup

同一グラフパターンに属するインスタンス間で頂点の共有の程度を表すメトリクスである．これはグラフパターンに対して 1 以上の実数で定義される．計算方法の概要は図 6 のようになる．

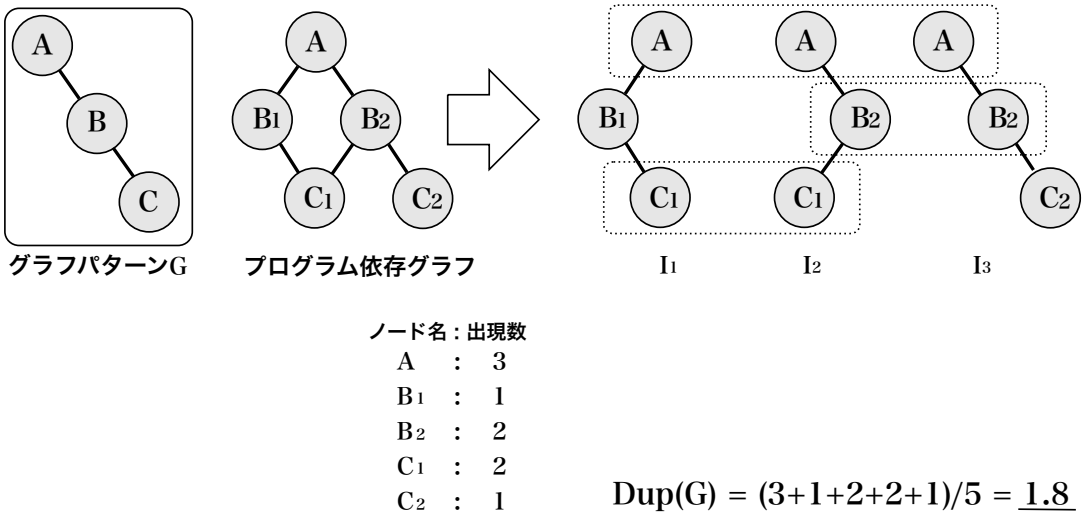


図 6: 頂点重複度の計算

このメトリクスは，似た処理が連続して現れるソースコードから抽出されたグラフパターンを除外するために考案した．例として swtich 文の場合を考える．それぞれの分岐処理が同じハッシュ値をもつ頂点になっていると，swtich 文の前後の頂点と各分岐処理がグラフパターンのインスタンスとなる．その結果，分岐処理の数だけインスタンスをもつグラフパターンとして抽出されてしまう．このようなグラフパターンは誤検出の原因になると考えられる．したがって，頂点重複度が高いグラフパターンに起因する欠陥候補は誤検出である可能性が高い．

頂点重複度の詳細な算出方法は次のように定義される．プログラム依存全体の集合 H においてグラフパターン G のインスタンスの集合を $I(G,H)$ とし，$I(G,H)$ 中で頂点 N が出現する回数を $C_N(n, I(G,H))$ で表すとき，G 中に現れる頂点の重複しない集合を $Nall(G)$ と書く．これらを用いて頂点重複度 $Dup(G)$ は次のように書ける．

$$Dup(G) = \frac{\sum_{n \in Nall(G)} C_N(n, I(G,H))}{|Nall(G)|} \tag{10}$$

なお $Nall(G)$ は，グラフパターンの出現 I 中に出現する頂点の集合を $N_I(I)$ で表すと次のように書ける．

$$Nall(G) = \bigcup_{i \in I(G,H)} N_I(i) \tag{11}$$

3.3.5 平均ギャップ長: Gap

グラフパターンについて，それぞれのインスタンスがソースコード上でどれだけ疎なのかを表すメトリクスである．この値は，あるグラフパターン中でのインスタンスのギャップ値

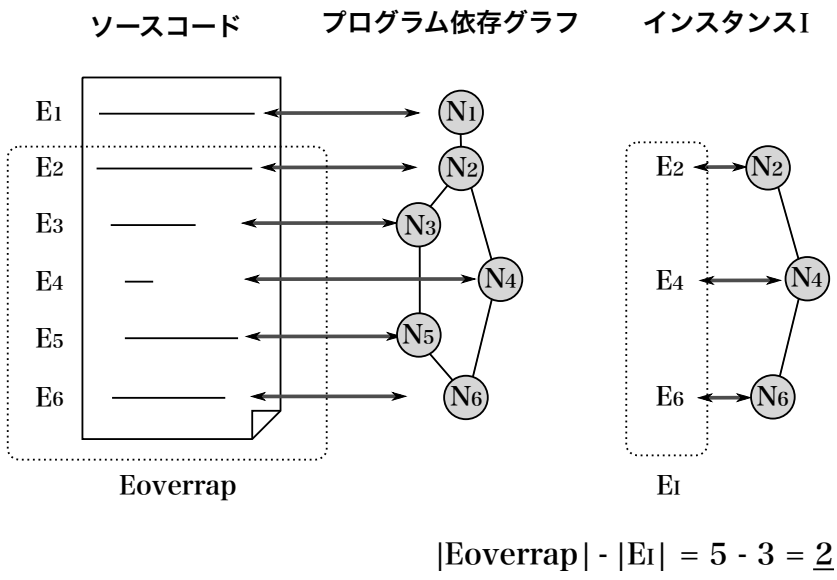


図 7: インスタンス単位でのギャップ値の計算

の平均である．単一インスタンスのギャップ値は図 7 にあげたように計算できる．

グラフパターンのインスタンスは，全てプログラム依存グラフ中で隣接する頂点である．しかし頂点に対応する文は必ずしも連続しない．広範囲にコード片が分散していると，偶然に一致した可能性が高くなる．したがって，平均ギャップ長が高いグラフパターンに起因する欠陥候補は誤検出である可能性が高い．

直観的に，この値はグラフパターンに対応するコード片の中に，どれだけグラフパターンに対応しないコード片が存在するかを表す値といえる．

平均ギャップ長 $Gap(G)$ は次のように定義される．

$$Gap(G) = \frac{\sum_{I \in I(G,H)} |E_{overrap}(I) - E_I|}{|I(G,H)|} \tag{12}$$

ここで，E_I とはインスタンス中の頂点 $N_I(I)$ に対応するソースコード要素の順序つき集合 (系列) であり，次のように定義される．ただし，$N_I(I) = [N_1, N_2, ..., N_n)]$ とし，頂点 N に対応するソースコード中の要素を $E(N)$ で表す．

$$E_I = [E(N_1), E(N_2), ..., E(N_n)] \tag{13}$$

また，$E_{overrap}(I)$ とはインスタンス I を含むプログラム依存グラフ全頂点に対応するソースコード要素を $[E_1, E_2, ..., E_k]$ としたとき次のように定義される．

$$E_{overrap}(I) = [E_p, ...E_q](E_p = E(N_1), E_q = E(N_n)) \tag{14}$$

4 評価実験

本手法が2.5節で提示した既存手法における問題点を解決ができたか評価するため，GrouMinerとの比較実験を行なった．

実験対象および検出された欠陥候補数と実行時間を表2にあげた．これらはNguyenらがGrouMinerの評価に用いたオープンソースプロジェクトである．しかし，GrouMinerが対象としたプロジェクトのうち，Fluid VCはソースコードが入手できなかったため対象から除外している．いずれも確信度の閾値を0.6に設定した．なお，実験は全てCPUがCore2Duo 1.86GHz，RAMを8GB搭載した計算機上で行なった．

4.1 実験準備

実験で用いる閾値を以下の手順で決定した．

1. 全ての欠陥候補から，各メトリクスの上位50%以上に存在する欠陥候補の集合を得る．ただし，50%の位置に存在する欠陥候補の値と同じ値をもつ候補は，50%以降に存在しても含める．また，並べ替えの順は表1での解釈に従う．つまり高いほどよいと考えられるメトリクスについては，降順にて並べ替える．

2. 各メトリクス毎の欠陥候補の集合の積集合を求める．つまり，各プロジェクトについて，全てのメトリクスで上位50%に位置する欠陥候補を得る．

3. 各プロジェクトの積集合を全て以下の基準で分類する．

   欠陥 欠陥 (バグ) と考えられる欠陥候補

   不吉な匂い 欠陥ではないが，修正の必要性を示唆していると考えられる欠陥候補．不吉な匂い (Bad Smell) とはFowlerが提唱した概念である [3].

   誤検出 上2つに該当しない欠陥候補．

   分類は，著者がソースコード，およびドキュメントなどから目視で行なった．

4. 各プロジェクトの積集合について分類した欠陥候補のうち，“欠陥” もしくは “不吉な匂い” に該当する候補を全てのプロジェクトから列挙し，それぞれのメトリクスの最大値および最小値を求める．その値を実験に用いる閾値とする．

見つかった欠陥および不吉な匂い全てを表3にあげた．IDは欠陥候補の識別子である．また，DupとGapはグラフパターンに対して定義されるため，欠陥候補の値にCand，相関

プロジェクト名	ファイル数	メソッド数	グラフパターン数	欠陥候補数	実行時間[秒]
Apache Ant 1.7.1	1123	8789	2722	950	155
Apache log4J 1.2.15	296	2281	672	143	36.7
AspectJ 1.6.3	1664	4427	1012	368	74.1
Apache Axis 1.1	1107	5693	1953	689	119
Columba 1.4	1549	8305	2378	1343	189
jEdit 3.0	251	2047	1810	632	77.0
Jigsaw 2.0.5	713	5188	1864	723	99.7
Struts 1.2.6	365	2943	487	137	46.6

表 2: 実験対象詳細

プロジェクト名	ID	Conf	Lift	Missing	Candcont	Dup Cand	Dup Right	Gap Cand	Gap Right
Apache Ant	856⇒ 857	0.67	0.33	1	1	1	1	1	1.5
1.7.1	498⇒ 499	0.67	0.33	1	1	1	1	3	2.5
	157⇒ 158	0.67	0.33	1	1	1	1	0	0
AspectJ 1.6.3	119⇒ 120	0.67	0.33	2	1	1	1	2	1
	299⇒ 300	0.67	0.33	2	1	1	1	0	0
Apache	45⇒ 48	0.67	0.33	1	1	1	1	0	0
Axis 1.1	190⇒ 191	0.67	0.33	1	1	1	1	1	1
	592⇒ 593	0.67	0.33	1	1	1	1	2	1
Columba 1.4	348⇒ 349	0.67	0.33	1	1	1	1	1	1
	732⇒ 733	0.67	0.33	2	1	1	1	0	6
jEdit 3.0	117⇒ 118	0.67	0.33	4	1	1	1	0	0.5
	350⇒ 76	0.75	0.25	2	1	1	1	1	0
Jigsaw 2.0.5	339⇒ 340	0.75	0.25	2	1	1	1	0	0
	380⇒ 381	0.67	0.33	2	1	1	1	2	1.5

表 3: 各プロジェクトから見つかった欠陥および不吉な匂い

ルールの右辺の値に Right と表記している．なお，Struts 1.2.6 と Apache log4J 1.2.15 については誤検出しか見つからなかったため表には掲載していない．

この結果から，それぞれのメトリクスについてフィルタリングの閾値は表 4 のようになる．$Candcount, DupCond, DupRight$ について “1 のみ” と表記したのは，これらの値は小さいほうが欠陥らしいことを示す値であり，かつ 1 以上で定義されるメトリクスだからである．

4.2 実験手順

4.1 節で決定した閾値に基づき次の手順で評価を行なった．

1. 欠陥候補を 4.1 節で求めた閾値によりフィルタリングする．
2. 各プロジェクトについて，フィルタリングした欠陥候補を $Missing$, $Lift$, $Candcount$, $DupCond$, $DupRight$ の 4 つのメトリクスについて優先順位を変えて並べ替える．優先順位と並べ替えの方向は 1 に従う．つまり $Lift$ のみが降順，あとは昇順である．並べ替えに用いるメトリクスから $Candcount, DupCond, DupRight$ を除いたのは，フィルタリング後にこれらのメトリクスの値が全て 1 になるからである．
3. 各プロジェクトを並べ替えた後，表 3 での欠陥が 15 位以内に位置しているか調べる．15 位以内というのは Nguyen らが GrouMiner の評価で用いた条件である．

なお，実際に用いた優先順位は次の 2 通りである．

- $Missing > GapRight > GapCond > Lift$
- $GapRight > GapCond > Missing > Lift$

以下で $Lift$ を常に最下位に設定した理由と，2 種類の Gap を連続して位置づけた理由を説明する．

メトリクス名	閾値
Lift	0.25 以上
Missing	4 以下
Candcount	1 のみ
Dup Cond	1 のみ
Dup Right	1 のみ
Gap Cond	3 以下
Gap Right	6 以下

表 4: フィルタリングに用いるメトリクスの閾値

まず $Lift$ を最下位に設定した理由であるが，これは $Lift$ の値が偏っていたためである．8 プロジェクト全ての欠陥候補のフィルタリング後の総数は 166 であり，そのうち 145 が 0.33 であった．偏った値で並べ替えを行なっても順位がほとんど変わらないと考えられるため，最下位とした．

また 2 種類の Gap である $GapCand, GapRight$ を連続した順位に位置づけた理由は，定義上これら 2 つのメトリクスは相関関係が大きいためである．3.2 節で説明したように，相関ルール $G_A \Rightarrow G_B$ は，$N(G_A) \subset N(G_B)$ の関係を満たす．これは全インスタンスにおいて成立するため，$Gap(G_A)$ が大きくなると $Gap(G_B)$ も大きくなる．

4.3 実験結果

評価実験の結果と，比較対象となる GrouMiner の検出結果を表 5 にあげた．GrouMiner は確信度で並べ替えを行い，その上位 15 件で評価を行なった値である．また，並べ替え 1 が $Missing > GapRight > GapCond > Lift$ であり，$GapRight > GapCond > Missing > Lift$ が並べ替え 2 である．

閾値決定実験で発見した欠陥および不吉な匂いは 14 個であり，並べ替え 1 ではそのうち 13 個が，並べ替え 2 では 8 個が出現しているのがわかる．さらに並べ替え 2 で発見した欠陥および不吉な匂いはいずれも並べ替え 1 で発見されたものであった．このことから，$Missing$ が欠陥らしさを表すメトリクスとして有用である可能性が考えられる．

以下で 2.5 節で既存手法の問題点を解決ができたか評価を行う．問題点を再掲する．

問題 1 モデルを抽象化しすぎるため情報が欠落し，抽出できないイディオムが増える．結果として検出可能な欠陥の低下が起こる

問題 2 確信度の閾値を高く設定するため，欠陥として検出されるにはパターンが多数出現する必要があり，少数しか出現しないイディオムに関する欠陥を検出できない

まず問題 1 についてである．GrouMiner が用いた Groum を含め，オブジェクトの使用に特化したモデルでは取得できないイディオム，および欠陥候補を抽出できている点では解決できたといえる．

例として Apache Axis 1.1 中に出現した ID 299 ⇒ 300 の欠陥を図 8 にあげた．この例では，図の下部のソースコード中で現れる変数 `sslFactory` の null チェックが，上部のソースコード中で欠落している．このコード片が現れるメソッドの名前，引数，およびそのメソッドを定義しているクラスの継承関係が 3 つのコード片で全て同一である．また，3 つのメソッド全体がイディオムとして抽出されており，リファクタリングの必要性も考えられる．この

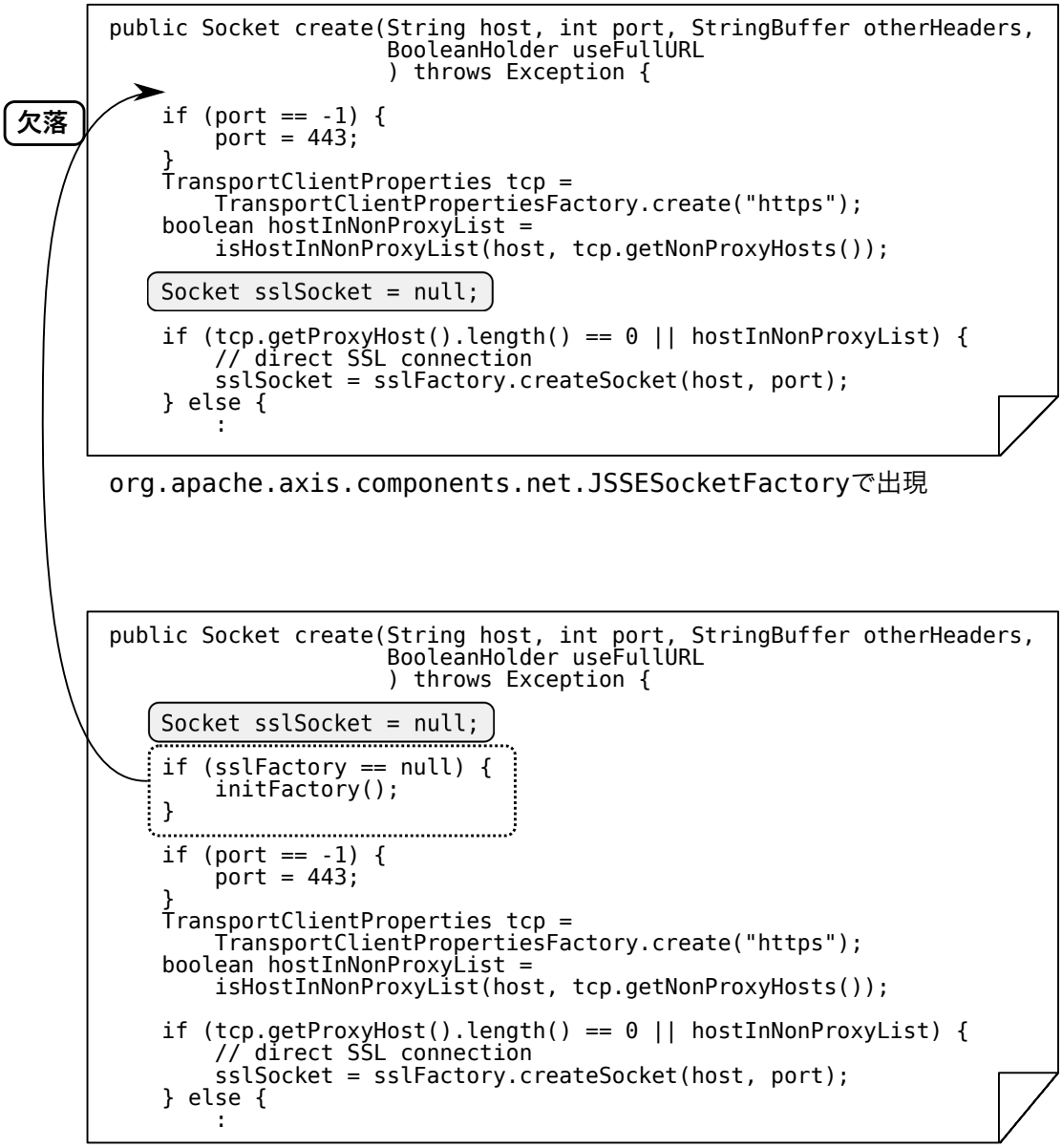


図 8: Apache Axis 1.1 で発見された欠陥

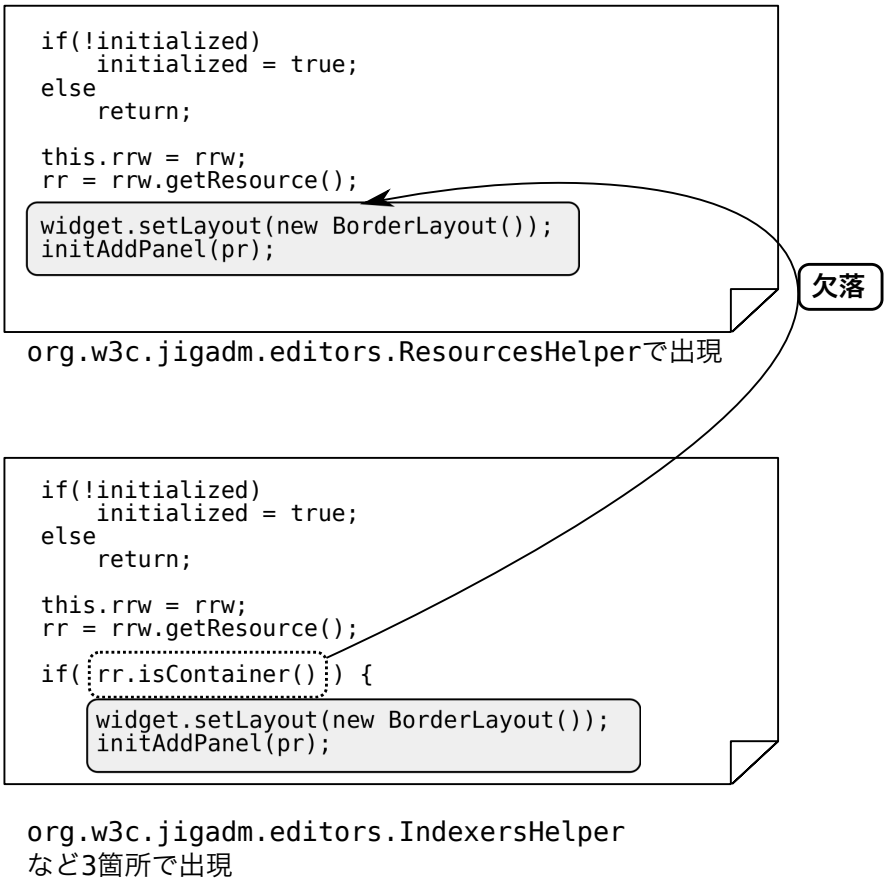


図 9: Jigsaw 2.0.5 で発見された不吉な匂い

例は，網掛けで囲った部分の位置が 3 箇所の出現の間で異なるため，従来の手法では抽出できないイディオムである．

また，もう 1 つの例として Jigsaw 2.0.5 で出現した ID339 ⇒ 340 の不吉な匂いを図 9 にあげた．この例では，図中の下部に示したソースコード中で破線で囲った `rr.isContainer()` の呼び出しが欠落している．原理上これは GrouMiner でも検出可能である．しかし，破線に続く網掛けで囲った後半部分は前半から依存関係が一切ないため，イディオムとして抽出できない．

次に問題 2 について述べる．表 3 にあげた欠陥および不吉な匂いの確信度は全て 0.67 か 0.75 のいずれかであった．これらの値は従来手法が評価で用いた閾値では除外されてしまう候補である．

ここで，確信度を低下させたことによる影響について述べる．表 6 に本手法の確信度 0.9 以上の場合と 0.6 以上でのフィルタリング前・後，さらに GrouMiner についての欠陥候補数をあげた．なお GrouMiner の確信度の閾値は 0.9 である．本手法では確信度の閾値を 0.9

から 0.6 に下げたことで欠陥候補が大幅に増加していることがわかる．いずれも GrouMiner に比べ非常に多く，全て閲覧することは困難である．

しかしフィルタリングを行うことで，Columba 1.4 を除き GrouMiner より候補数が少なり，全候補の閲覧も可能な範囲に収まった．並べ替え 1 では表 3 での 14 個の欠陥および不吉な匂いうち 13 個が上位 15 候補に位置しており，フィルタリングおよび並び替えは有効であると考えられる．

プロジェクト名	候補数	並べ替え 1		並べ替え 2		GrouMiner	
		欠陥	不吉な匂い	欠陥	不吉な匂い	欠陥	不吉な匂い
Apache Ant 1.7.1	24	2	1	1	1	1	0
Apache log4J 1.2.15	11	0	0	0	0	0	1
AspectJ 1.6.3	14	0	1	0	1	1	2
Apache Axis 1.1	27	1	3	1	1	0	2
Columba 1.4	50	0	1	0	0	1	0
jEdit 3.0	11	1	0	1	0	1	0
Jigsaw 2.0.5	26	2	1	1	1	1	1
Struts 1.2.6	3	0	0	0	0	0	0
合計	166	6	7	4	4	5	6

表 5: 各プロジェクト上位 15 候補中における欠陥および不吉な匂い

プロジェクト名	本手法	フィルタリング		GrouMiner
	(確信度 0.9 以上)	前	後	
Apache Ant 1.7.1	34	950	24	145
Apache log4J 1.2.15	8	143	11	32
AspectJ 1.6.3	26	368	14	244
Apache Axis 1.1	32	689	27	145
Columba 1.4	144	1343	50	40
jEdit 3.0	36	632	11	47
Jigsaw 2.0.5	41	723	26	115
Struts 1.2.6	5	137	3	33

表 6: 欠陥候補数の比較

5　考察

5.1　評価の妥当性について

欠陥および不吉な匂いの判断を著者が行なった点が妥当でない可能性がある．この点については 3 つの解決方法が考えられる．

まず 1 つめは全ての対象について，版管理システムの履歴全体に渡り解析を行い，Bugzilla[2] などの欠陥追跡システムや更新の際のコメントなどから，欠陥などとして報告されているか調べる方法である．しかし，欠陥追跡システムが使用されているとは限らず対象が限られてしまうことや，コメントに欠陥修正か否か記載されていない可能性があるため，全てのプロジェクトに対して使える方法ではない．

2 つめはプロジェクトの開発者にインタビューを行い確認する方法である．開発者から直接意見を聞くことでより正確に評価を行うことができると考えられるが，最新の版でない限り該当箇所の開発者に連絡がとれるか否かわからないため，これも全対象で用いることはできない．

3 つめは，一般のソフトウェア開発者複数を対象に検出した候補を評価してもらう方法である．筆者 1 人で行うよりは正確な評価を行えると考えられるが，開発者の熟練度に対する依存が大きい点が問題になると考えられる．

また，GrouMiner との比較において，検出された欠陥および不吉な匂いの数で比較を行い，また図 8 や図 9 では，GrouMiner のアルゴリズムから推測して検出できないと判断した．これは GrouMiner が入手できなかったためであり，より正確に比較を行うためには GrouMiner を用いて全ての欠陥候補を比較する必要がある．

フィルタリングに用いた閾値の妥当性であるが，これは対象とした 8 プロジェクトが複数ドメインに渡っていること，また閾値の決定の基となった欠陥候補及び不吉な匂いの数が極端に少なくないことから，特定のプロジェクトに過剰適合した値ではないと考えられる．また，過剰適合でないことを示すために，今回の対象とは異なるプロジェクトに対しても同様の評価を行う必要がある．

5.2　異なる手法を併用した欠陥検出について

自動欠陥検出手法はいずれも異なるモデルを用いているため，単純に欠陥の検出能力に優劣はつかない．これは異なる手法を併用することで，より多くの欠陥などを検出できるとも言い換えることができる．ただし，併用により誤検出も増える．そこで，例えばある欠陥候補がどれだけの手法から検出されたかを表すメトリクスなど，併用ならではのメトリクスを用いたフィルタリング方法が有効な可能性がある．また，いずれの手法も計算コストが高い

ため，計算にかかる時間の増大は免れない．しかし，共通の処理を一括して行うよう実装することで実行時間の増加を抑えれると考えられる．

本研究で提案したメトリクス群は，いずれも他の自動欠陥検出でも利用可能であると考えられる．したがって，本手法に用いたようなフィルタリングや並べ替えを他の手法に適用することで，それらの手法で従来の閾値では検出できなかった欠陥および不吉な匂いが検出できるようになる可能性がある．

5.3 実行時間について

表 2 にあげたように，解析に要した時間は最長でも 189 秒であり，十分実用に耐え得る実行速度であると考えられる．

実行時間は概ねグラフパターン数に依存しているといえる．しかし，AspectJ 1.6.3 ではグラフパターン 368 で 74.1 秒かかっており，グラフパターンが 689 の Apache Axis 1.1 が 119 秒であることと比べると遅いといえる．これは AspectJ のファイル数が多く，ファイルの解析にかかる時間が長いためと考えられる．

6　あとがき

ソフトウェアを開発する際，イディオムという特定の処理を実装するパターンを利用することがある．しかし，イディオムが常に正しく実装されるとは限らない．このような誤実装を自動で検出する研究が行われてきたが，高速化などのために精度が低下していると考えられた．そこで本研究ではよりプログラム依存グラフを用い，より高精度な検出を目的とする手法を提案した．実際に，従来手法では検出できない欠陥などを発見することができ，有効性が確認された．

今後の課題としては，5 節で述べたようにより正確な評価を，対象を増やして行う必要がある．

また，グラフパターンに対して定義される頂点重複度 Dup と平均ギャップ長 Gap についてはグラフパターン抽出中でのフィルタリングが可能である．これを実装することでより高速・省メモリでの検出が可能となると考えられる．

謝辞

本研究において, 常に適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授に心より深く感謝いたします.

本研究において, 適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下 誠 准教授に深く感謝いたします.

本研究において, 逐次適切な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 石尾 隆 助教に深く感謝いたします.

本研究で使用させて頂いたツール，Scorpio の開発者である大阪大学大学院情報科学研究科コンピュータサイエンス専攻 肥後 芳樹 助教に深く感謝いたします.

本研究において, 数多くの御指導および御助言を頂きました奈良先端科学技術大学院大学情報科学研究科 情報システム学専攻 吉田 則裕 助教に深く感謝いたします.

最後に, その他様々な御指導, 御助言等を頂いた大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に深く感謝いたします.

参考文献

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

[2] Bugzilla. `http://www.bugzilla.org/`.

[3] M. Fowler. *Refactoring: improving the design of existing code.* Addison Wesley, 1999.

[4] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proc. of FIMI 2003*, 2003.

[5] D. J. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining.* The MIT Press, 2001.

[6] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM SIGPLAN Notice*, Vol. 39, No. 12, pp. 92–106, 2004.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] PMD. `http://pmd.sourceforge.net/`.

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

[14] 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.

[15] 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.