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.
SS2001-34 (2002-01)

バグ報告データを用いたプログラムコード検証方法の提案

松村 知子　　門田 暁人　　松本 健一

奈良先端科学技術大学院大学情報科学研究科
〒630-0101 奈良県生駒市高山町 8916－5
0743-72-5312

E-mail: {tomoko-m, akito-m, matumoto}@is.aist-nara.ac.jp

あらまし 長年にわたって使用され更新されている大規模なレガシーソフトウェアでは，プログラムコード間の関係が複雑化したために，仕様書や設計書に記載されない潜在的なコーディング規則が多数存在する．これらの規則に違反したコーディングを行うと，バグ混入の原因となる．本稿では，規則に違反した個所のソースコードを自動的に検出・警告する方法を提案する．提案方法では，バグ報告データから「潜在コーディング規則に違反したコードパターン」を抽出し，パターン記述言語で表現する．そして，バグが潜在すると思われる部分のコードをパターンマッチングにより特定する．ある大規模レガシーソフトウェアに提案方法を適用した結果，約 30%のバグは潜在コーディング規則の違反により混入したものであり，そのうちの 82%のバグはパターン化が可能であり，それらのバグが潜在する個所を全て検出できることを確認した．

キーワード レガシーソフトウェア，ソフトウェア保守，コード検証，パターンマッチング，バグ報告データ

A Method for Program Code Inspection Using Bug Report

Tomoko MATSUMURA, Akito MONDEN, and Ken-ichi MATSUMOTO

Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama-Cho, Ikoma, Nara 630-0101
+81-743-72-5312

E-mail: {tomoko-m, akito-m, matumoto}@is.aist-nara.ac.jp

Abstract Thorough an ongoing process of maintaining large legacy software, a number of “implicit coding rules”, which are never described in specification documents or design documents, are unexpectedly generated as software becomes more complicated than it used be. Violating such rules cause fault injection. This paper proposes a method for automatically detecting and informing code fragments that violate implicit coding rules. In this method, patterns of typical code violating rules are derived from past bug reports. The patterns are described by a pattern description language, and, a potential faulty code fragments are extracted by a pattern matching technique. The result of a case study showed that 30% of faults in a large legacy system were due to the violation of implicit coding rules, 82% of these faults were descriptable by the pattern language, and, all these faults were extracted by the proposed method.

keywords legacy software, software maintenance, code inspection, pattern matching, bug report

1. はじめに

今日，大規模レガシーソフトウェアの信頼性の向上と保守コストの低減は，多くの企業にとって重要な課題となっている [11, 17]. レガシーソフトウェアとは，10～30 年の長期にわたって開発・保守が繰り返されている遺産的なソフトウェアのことである．長期にわたる保守の過程では，度重なる機能変更や追加，修正のために Code Decay[2]（コードの劣化：ソフトウェアの構造が複雑になることを指す）が起こり，ソフトウェアに変更を加えることが次第に困難となる．そのため，機能追加や変更の際に，要求される信頼性を達成することが困難になったり，達成できたとしてもより多くの人員や時間を要するようになる．また，長期の保守の過程では保守作業者が入れ替わることも多く，複雑化したソフトウェアに関する知識や経験が流出することが保守をより困難にしている[12].

本稿では，保守を困難にしている要因の一つとして，保守対象のレガシーソフトウェアに熟知した者だけが知っているコーディング規則（以降では，「潜在コーディング規則」と呼ぶ）に着目する．機能拡張・変更が繰り返されてきたレガシーソフトウェアには，「大域変数 A に値を代入しないで関数 B を処理するとある機能が動作しない」，「関数 X の呼び出し直後に関数 Y を呼び出すと異常が発生する」といった潜在コーディング規則が数多く存在する．このような規則は，設計書や仕様書には記載されず，保守作業者の頭の中に暗黙的に記憶される．そのため，保守作業者の退職時や入れ替わり時に規則が潜在化しやすい．

潜在コーディング規則の存在は，レガシーソフトウェアの信頼性の低下，及び，保守コストの増大の大きな要因となっている．潜在コーディング規則に違反することはバグ（フォールト）混入の原因となるため，保守作業者は，全ての規則に違反していないかどうかを確認しながら保守を行う必要がある．しかし，第 2 章で述べるように，我々の調査したレガシーソフトウェアでは，保守工程で発見されたバグの約 30%は潜在コーディング規則の違反により混入したものであった．

本稿では，潜在コーディング規則に違反していると思われる部分のソースコードを自動的に検出・警告する方法を提案し，ある大規模レガシーソフトウェアのサブシステムに適用したケーススタディについて報告する．保守作業者がソフトウェアの機能拡張や変更を行った直後に，潜在コーディング規則に違反している個所を自動的に検出・警告できれば，規則の違反によるバグの混入を未然に防止できると期待される．

提案方法では，「潜在コーディング規則に違反したソースコード」をパターンとして予め記述しておく（以降，これを「バグコードパターン」と呼ぶ）．そして，対象となるレガシーソフトウェアとバグコードパターンのマッチングを行い，マッチした部分のソースコードを保守作業者へ提示する．バグコードパターンは，多数の保守作業者が過去に作成したバグに関する文書（本稿では，「バグ報告データ」と呼ぶ）から抽出する．ここでいうバグ報告データとは，バグが存在したソースコード上の位置，バグが混入した原因，及び，バグの除去方法等についての情報であり，多くの企業においてこれらの情報が記載された文書が作成されている．抽出したバグコードパターンの記述には，文献[16]で提案されているパターン記述言語を拡張したものを用いる．

以降，第 2 章では，潜在コーディング規則とバグコードパターンについて述べる．第 3 章では提案手法について述べ，第 4 章ではそのケーススタディを行う．第 5 章では関連研究について述べ，第 6 章ではまとめと今後の課題を述べる．

2. 潜在コーディング規則とバグコードパターン

2.1. 潜在コーディング規則の特徴とその発生原因

潜在コーディング規則は，保守作業者が守るべきコード間の関係を示すものであり，次のような特徴を持つ．

- いわゆる「コーディング規約」としてソフトウェア開発・保守組織において予め定められたものではなく，長期にわたる保守の過程で予期せず発生する規則である．
- 設計書や仕様書に明示的に記述されておらず，保守作業者の頭の中に暗黙的に記憶されている．そのため，保守作業者の退職時や入れ替わり時に規則が潜在化しやすい．
- 一般的なライブラリの使い方を定めた規則（ライブラリ仕様）ではなく，保守対象のソフトウェアごとに特化して発生する規則である．
- 潜在コーディング規則に違反したやり方でソフトウェアに変更を加えたとしても，直ちに故障が発生するとは限らない．特殊な状況でのみ故障が発生し，バグの早期発見が困難な場合がある．
- 一般的なチェックリストによるソースコードレビューを行ったとしても，潜在コーディング規則に違反している部分を発見することは困難である．一般的なチェックリストには，対象ソフ

トウェアに特化したチェック項目が載っていないためである．

- ソースコードの Refactoring[3]などの手段を用いて，潜在コーディング規則を解消することは必ずしも容易でない．一般に，大規模で古いソフトウェアであるほど，ソフトウェアの隅々までを熟知している保守作業者がおらず，現状を正しく表す仕様書や設計書も残っていないために，Refactoring にかかるコストは非常に大きくなり，かつ，多大なリスクを伴う．

以上のような潜在コーディング規則が発生する原因は，保守において繰り返される機能変更や追加である．例えば，機能追加によって，従来は一つの機能のみから参照されていたデータXが，新たに追加された別の機能でも参照される場合がある．このような場合，データXの取り扱いに関する潜在コーディング規則が発生する．この規則を知らずに一方の機能の変更時にデータに変更を加えると，他方の機能が正しく動作しなくなることがある．また，追加した機能が既存の機能との動作タイミングによって故障を発生させるケースもある．

潜在コーディング規則の存在は，レガシーソフトウェアの信頼性の低下，及び，保守コストの増大の大きな要因となる．大規模なレガシーソフトウェアではサブシステムごとに異なるグループ（時には別会社）が担当する場合があり，保守作業者間で潜在コーディング規則が共有されにくい．また，このような規則は設計書や仕様書に記述されることは無いので，保守担当者が入れ替わるとこれらの情報が消滅してしまう場合がある．このような潜在化した規則が増加すると，保守作業においてバグ混入を誘発する危険が大きくなる．そのため，慎重なレビューや膨大な再テストが必要になり，保守コストは増大する．

2.2. バグコードパターン

前述した潜在コーディング規則に対して，規則に違反している部分のソースコードをパターンとして表したものを「バグコードパターン」と呼ぶ．二つの例を表1に示す．

表 1　潜在規則とバグコードパターンの対応

潜在コーディング規則	バグコードパターン
機能 C を正常に動作させるためには，関数 B を処理する前に大域変数 A に必要な情報を代入する．	大域変数 A への設定無しに関数 B を呼び出す．
関数 X の呼び出し直後に関数 Y が処理されると異常が発生するため，関数 X と Y は続けて処理しない．	関数 X と Y を続けて呼び出す．

バグコードパターンに該当する部分のソースコードは故障の原因となり得るため，保守作業者はこれらのパターンが存在する個所を全て特定し，故障を生じさせないかどうかを確認する必要がある．

2.3. バグコードパターンの実情

あるレガシーソフトウェアの1サブシステムを対象として，バグと潜在コーディング規則の件数を調査した．バグ報告書からバグ混入の原因がコード上明確に特定されたケース 162 件中 48 件が潜在コーディング規則の違反によるものであった（29.6%・同一規則によるバグを含む）．従って，もしあらかじめ潜在コーディング規則がわかっていて，それに対応した調査が行われていれば，これらのバグ混入は未然に防止できたと思われる．このうち，2つの潜在コーディング規則の違反による4件のバグは，後日規則を解消するように再設計され，修正された．つまり，残りの 44 件のバグに関連する潜在コーディング規則は，現在も残っており，将来の保守作業でバグ混入の原因となる可能性がある．

3. 本研究での提案手法

3.1. コード検証の流れ

本研究では，バグコードパターンに一致する部分のソースコードを自動的に検出・警告するシステム（コード検証支援システム）を提案する．提案するシステムの大まかな流れは図1のとおりである．

(1) バグ報告データから潜在コーディング規則を抽出し，バグコードパターンを作成し，システムに保存する．
(2) 開発・保守作業者はバグを検出したいソースコードをシステムに入力する．
(3) 出力されたバグを発生させる可能性があるコード（疑惑コード）をバグ情報を使ってチェックする．
(4) バグが存在する場合対応を行い，そのバグ情報を報告する．
(5) システム上のバグ情報の保守を行う．(潜在コーディング規則が解消したものに関してはシステムから削除する等)

3.2. バグ報告データ

バグ報告データとは，プロセスデータの一つである．これは，通常テスト工程や運用工程でバグが発生したときに作成され，報告・修正・リリースなどの一連の作業の状況や結果が追加される．今回使用したバグ報告データには，バグ発生報告書，不具合修正報告書，ファイル更新通知の3種類があり，それぞれ以下のような情報が含まれる．

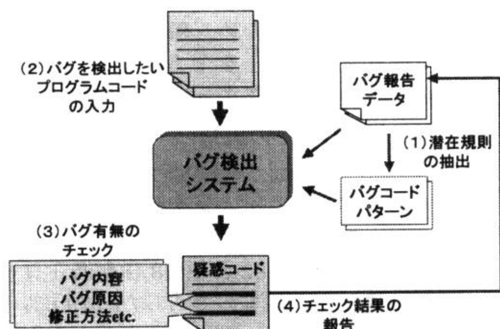

図1　コード検証システムの概要

バグ内容:ある画面の表示中に特殊なキー入力による割り込み処理が発生すると、元の画面に復帰するときに正しい画面が表示されない。

```
F(){
  int c;

  c=0;
  for( ; ; c++){
    if( F2(c)== 0 )
      break;
  ChangeView( );
  return(RET_OK);
}
```

バグ原因:復帰画面表示のために必要なページ番号が、変数Ctrl.Noへの保存されずに、ChangeView()関数を呼び出したため。

修正方法: ChangeView()関数を呼び出す前に、ページ番号を変数Ctrl.Noに保存する。

図2　バグ報告データの例

- バグ番号・バグ発生状況・バグ内容
- バグ原因・修正方法・影響調査・テスト仕様・混入時期・混入理由
- 更新情報・バックアップ情報

ここでは潜在コーディング規則を抽出するために主に不具合修正報告書を参照した.

3.3. バグ報告データからのバグコードパターンの抽出

ここで，バグコードパターンとして取り出すバグの条件は，以下のとおりである.

- バグの混入原因が明確で，かつ，ソースコード上のバグ存在個所が明確である.
- 潜在コーディング規則に違反していることより発生したバグである.
- 今後の保守において同様のバグが混入する可能性がある.
- バグ存在個所のソースコードをバグコードパターンとしてパターン記述言語で表現可能である.

対象となるバグの例を図2に示す．このバグ報告データから抽出される潜在コーディング規則は「ChangeView()関数を呼び出す前には，Ctrl.Noへ表示中のページ番号を代入しなければならない」,バグコードパターンとしては「Ctrl.No への代入無しに ChangeView()関数を呼び出す」というようになるだろう.

3.4. バグコードパターンの記述

バグコードパターンを記述するためには，柔軟で拡張性の高い記述言語が必要になる．バグコードパターンはシステムの内容やコーディング規則などにも依存し，特に複雑なパターンでは複数の関数やファイルにわたるパターンも存在する．そのため，本稿ではプログラム言語に基づく表現方法[16]を採用する．この表現方法は，コードの再利用やコード理解のためのソースコード検索を目的に作成されたが，記述言語がプログラム言語に類似するため記述が容易であり，拡張性も高い.

本研究では，この記述方法に以下のような拡張を行う（図3参照).

(1) 命令の不在
(2) 命令の存在を呼び出し関数まで拡張
(3) 大域変数
(4) 正規表現の使用（キーワード宣言）
(5) パターンの連鎖

(1)は関数・変数の組み合わせの規則に対して必要な処理が欠けているバグコードパターンに必要である．(2)は組み合わせの命令が関数呼び出しを経由する場合などに用いる．(3)は設定される変数が大域変数かどうかを明示するために用いる．[16]では変数・関数名の記述時にワイルドカードを使った表現を用いているが，(4)でこれを拡張し，grep などで用いられる正規表現を採用してさらに名称の多様な表現ができるようにする．これは大規模なシステムで変数・関数の命名規則などが厳格に決められている場合，非常に有効に用いることができる．(5)ではコードの一部分にパターンが収まらず，複数の関数やファイルにわたる関連をパターン記述するために用いる.

このように記述されたパターンに対しては，関数単位でマッチングを行う．キーワード宣言がある場合，それをあらかじめ検索してマッチングする関数を絞り込むことで，効率的な検索を行うことができる.

これらを使用して記述されたパターンの例を図4に示す．図4の例2）のパターンに対する照合処理結果,検出されるコード部分は図5のようになる.

3.5. 照合結果の提示と適用

今回の研究では，パターンマッチングのプロトタイププログラムを作成しただけで，結果の提示や適用を視野に入れたツールの開発は行っていないが，最終的には，以下のような機能を実現する必要があると考えている.

(1) 一致コードの提示：一致したコード部分を効果

パターン表現記号([16]から)

宣言	$d	$*d	$d_{name}
型	$t		$t_{name}
変数	$v	$*v	$v_{name}
関数	$f		$f_{name}
式	#	#*	#_{name}
命令	@	@*	@_{name}

$v	不特定の変数
$*v	不特定の変数集合
$v_{name}	特定の変数
@[stmt1| stmt2]	いくつかの命令のどれか
@<id_1>	特定の識別子を参照

拡張パターン表現記号

^@$f_1(#*);	関数f_1の呼出し命令が存在しない
$~f_1	関数f_1が下位関数を含めて呼び出されている
$f~_1	関数f_1とその呼び出し関数を含む
$vg	グローバル変数

図3　パターン表現記号一覧

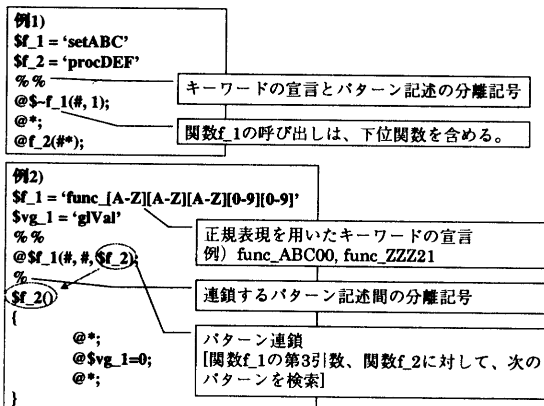

図4　パターン記述例

的に提示する機能や連鎖パターンの検索機能等.

(2) バグ情報の提示：そのパターンの源となったバグに関する情報の提示機能．具体的には，バグ番号やバグ内容，原因，修正内容などを提示し，さらに詳細な情報の獲得方法（ファイルの保管場所等）を提示する．

(3) 適用と保守：修正によって解消されたパターンの消去や新たなバグ発見情報の追加．潜在コーディング規則を解消するような根本的な対応が行われた場合は，このパターンをシステムから消去する．また，対症療法を採る場合，その発見されたバグに関する情報をパターンにリンクして取り出せるようにする．それによって，同一パターンでのバグの発生状況がわかり，それに対する対応方法の選択基準になる．

4. ケーススタディ

4.1. 目的

対象のソフトウェアに対して，バグを発生させる

```
void
func_x()
{
        ………〔略〕…………
        func_TYS10( type, size, func_PWD );
        ………〔略〕…………
}
………〔略〕…………
void
func_PWD ( a, b )
int     a;
char    *b;
{
        ………〔略〕…………
        glVal = 0;
        ………〔略〕…………
}
```

図5　パターン照合結果例

可能性があるコード部分を抽出するための提案手法の有効性を評価する．具体的には不具合修正報告書から取り出された潜在コーディング規則に対するバグコードパターンがパターン記述言語で表現できるか，記述されたパターンを用いたパターンマッチングによるコード部分の検出が可能か，報告されない潜在的なバグが実際にどのくらい含まれているかを評価する．これによって，従来検出困難だったバグの検出に対する有効性を評価する．

4.2. 評価事例ソフトウェア

今回研究の評価に用いたソフトウェアは，以下のようなものである．

- 組込み系・ハードウェア制御＋ユーザインターフェースを含むレガシーソフトウェアの1サブシステム
- 記述言語：C
- ファイル数：C ファイル　230　H ファイル　391（計 621）
- サイズ（行数）：C ファイル　約 356K　H ファイル　約 90K（計約 447K）（いずれも，最終リリース版）
- 開発・保守期間　1991 年～現在（1997 年 4 月～1999 年 5 月　※）

（※　このソフトウェアは多種のハードウェアに対して並行して開発が進められているため，全体的な開発期間は長いが，今回提供されたものは1ハードウェアに対するバージョンのみ．従って，実際の資料は括弧内の期間のもののみである．）

このソフトウェアに関して用いた資料は以下のとおりである．

- バグ発生報告書（紙媒体）
- 不具合修正報告書（紙媒体）
- ファイル更新通知（紙媒体）
- 最終リリースソースコード（電子媒体）
- マイナーリリースバックアップコード（1997

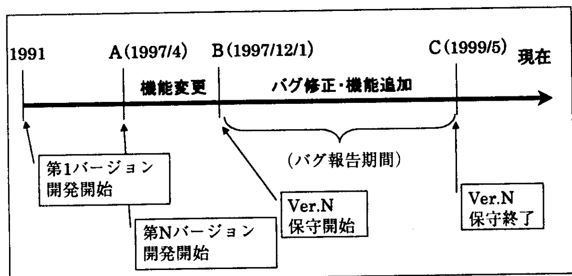

図6　ソフトウェア開発・保守期間

年9月～1999年5月）（電子媒体）

4.3. 評価方法

今回の実験で使用するソフトウェアに関しては，図6のB時点までに開発されたソースコードに対して，B～Cの期間に報告されたバグ報告データから抽出された潜在コーディング規則を使ってマッチングを行う（B以前のバグやレビュー情報は提供されていないため）．B～Cの期間に報告されるバグの中には，開発当初～Cの間に開発・保守されたコードによって混入した潜在コーディング規則が含まれるが，実際に照合によって抽出されるのは，開発当初～Bに混入したもののみがマッチすることになる．今回は，B以降の保守による混入バグは対象としない．

本来，提案手法では過去のバグ報告から抽出されたバグコードパターンを用いて，その後の変更コードからのバグを検索することを目的とする．しかし，今回はB以前に発生したバグに関する資料が入手できず，また多くのバグがB時点までの変更コードによって混入したため，このような方法をとった．B以前のバグ報告が存在していれば，検出できたバグと考えられる．

抽出されたコード部分に関して，以下の分類を行う．

A) 有効事例：バグ報告データによるバグ原因の状況と同じ状況が発生する可能性があり，バグの有無をチェックする必要があるコード部分．

B) バグ発見事例：バグ報告データによるバグ原因の状況でバグが発生する可能性があるコード部分．ただし，実際のシステムを動作して確認はできないため，コード上の可能性のみ．

C) バグ報告事例：バグ報告データによるバグ原因の状況でバグの発生報告が行われたコード部分．ここには，潜在コーディング規則パターンを取り出す元になるバグが含まれるため，必ず1件以上になる．

D) バグ未発見事例：バグコードパターンに一致するが，バグ原因となる状況においても，バグが発生する可能性がないコード部分．

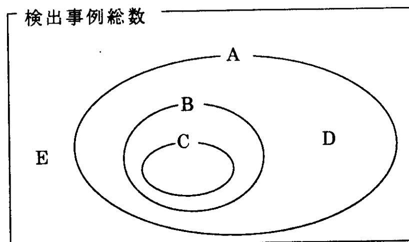

図7　検出事例分類関連

E) 無効事例：バグコードパターンに見かけは一致するが，実際に動作しなかったり条件文などによってバグを発生させない事例．

各ケースの包含関係は図7のとおりである．

4.4. 評価システム

パターンマッチングを行うために，以下のプロトタイププログラムを作成した．

- パターン解析・照合プログラム（C言語・シェルスクリプト）：字句・構文解析コードは，オープンソースから流用
- 評価作業用ツール（C言語）

検出されたコード部分は，ファイル名とマッチしたコード部分の開始行・終了行で表現される．これから重複するものを除いたものを検出事例総数とする．（連鎖パターンなどで，中継パターンでのマッチするコード部分が異なるケースも，同一パターンとする．）

4.5. 評価結果と考察

評価結果は，表2の通りである．表には記載しないが，今回バグ報告書から抽出されたバグコードパターンは45件，そのうち1997年12月1日時点では存在しない規則は6件，さらにコードパターンでは記述できないものが7件ある．従って，現状の提案方法で利用できるバグコードパターンは32件だった．これを使った照合によって，発生が報告されたバグは必ず検出され，さらにバグとして発生していない潜在バグも検出できた．また，多くの場合無効事例は無視できるほど小さいこともわかった．ただ，P1・P3のように報告されたバグのほかには同一のパターンに合致するバグは存在しないものも存在する．

この結果から，次のようなことが考察される．バグコードパターンとして表現できないものは約18%と小さく，また検出された無効事例は少ないので，パターン記述言語としては有効な方法であると思われる．表現できない7件に関しても，対象コードを変更コードに絞るなどして表現方法を拡張すれば，利用することは可能になる．また，いずれのパ

表 2 評価結果

パターン番号	検出事例総数 (A+E)	有効事例数 (A)	バグ発見事例数 (B)	バグ報告事例数 (C)	無効事例数 (E)	有効事例率 (A/(A+E))	バグ発見事例率 (B/(A+E))	発見バグ中の報告事例率 (C/B)
P1	1	1	1	1	0	1	1	1
P2	8	8	3	1	0	1	0.38	0.33
P3	1	1	1	1	0	1	1	1
P4	15	14	10	1	1	0.93	0.67	0.1
P6	14	10	4	2	4	0.71	0.29	0.5
P7	6	6	2	2	0	1	0.33	1
P10	24	11	8	1	13	0.46	0.33	0.13
P22	20	18	7	1	2	0.9	0.35	0.14
計	89	69	36	10	20	0.78	0.40	0.28

ターンも有効事例率が高く，無効事例のチェックのための不要な作業を強いられることは少なく，コードレビューにおける無駄なコストの増大はあまり無いと判断できる．

各バグ発見事例率も概ね高いといえる．実際に発生が報告されたバグに関しては 100%検出できた．これは，全体のバグ総数からのパターン化率が約 30%（2.3.参照）であることを考えると，すべての規則がパターン化できたならば 30%のバグは本手法でコードレビュー時に検出することができることを示す．さらに発見されたバグのうち報告されたバグは約 27.8%となり，潜在コーディング規則による潜在バグの検出の難しさと本提案手法の有効性を示すと考えられる．さらに，対象ソースコードに開発期間（A～B）のみでなく保守期間（B～C）のものを加えたり，コードレビューの時点でのコードを対象にしたりすることで，検出率は向上すると思われる．

5. 関連研究

ここでは，従来から研究されているコード検証支援手法について，いくつか紹介する．各手法の内容・特徴について述べ，本研究での提案手法と比較する．

5.1. パターンマッチング

パターンマッチングの手法を使ってコード検証を支援する研究には，以下のようなものがある．

小田等の C プログラムの落とし穴検出ツール Fall-in C[15]では，コンパイラや lint などでは検出できなかったり，適切なコメントができなかったりする字句・構文上の問題を検出し，警告を行う．

海尻のプラン認識を用いた方法[7]では，プロジェクトグループに固有のコーディング規則や標準的な書き方，デザインパターン的なものを検出する手法を提案している．

河合等の Sapid[18]を用いた方法[8]では，あるまとまったソースコードから多用される関数に関して規範パターンを取り出し，それに一致しないパターンを検出することを提案している．（例えば，fopen －fclose）これによって，パターンを探したりパターン記述をしたりという作業を自動化することができる．

これらは，いずれも一般的な問題の検出を対象としており，本研究で対象とするようなシステム依存のバグの検出への適用は考えられていない．

5.2. チェックリスト

設計書やソースコードに対するチェック項目をリスト化し，各工程で開発者・レビュア・保守担当者がチェックする手法である．

設計・コーディング（言語に依存する）に関する一般的な注意事項のリストに関しては，古くから研究が行われ，一般に普及しているリストが存在する[5, 6, 14]．また，これらのチェックリストを分類し，検証を行うコードに必要なもののみを精選することも提案されている[10]．

これらのチェックリストでは，システムに依存した問題はチェックできない．この点に関して，過去に発生したバグをチェックリストに追加し，そのリストを使ったレビュー方法が提案されている[13]．この方法では，システムに依存する問題がチェックでき，その頻度や発見難度・修正難度にしたがってリストを精選することで，チェック効率が良くなる．しかし，チェックを人手に頼るため，チェックのコストがかかったりチェック漏れが発生する可能性がある．また，リストから外れてしまったチェック項目によるバグは発見できない．

5.3. 不具合傾向モジュールの予測

ソースコードやバグ報告データから定量的に計測できるデータ（コード行数，関数呼び出し数，ループ数など）を予測因子として，モジュール単位で不具合傾向予測を行う．不具合を含む傾向にあるモジュールと無いモジュールに分類できると，不具合傾向にあるモジュールを重点的にチェックしたりテストしたりして，実際のバグの発見を支援することができる．

例えば，Khoshgoftaar 等は，新規・修正コード数，インストール回数，設計者が行った更新数などが予測に有効な因子になると言っている[9]. Graves 等は，修正履歴を用い，各モジュールにおける変更の回数と各変更が行われた時期を考慮した重み付けで予測精度が向上したと発表した[4]. これに対して，Andrews 等は，バグ履歴からコンポーネントやそれを含むモジュール・サブシステムの不具合傾向を判定する研究を行っている[1].

これらの方法では，モジュール単位での不具合有無の予測しかできないため，実際のバグの抽出やバグへの対応は従来どおりのテストやデバッグを必要とする．

6. あとがき

本稿では，コード検証時におけるバグの検出を支援する手法として，バグ報告データを用いたパターンマッチング法を提案し，その方法を用いて既存のソフトウェアからのバグ検出実験を行った．その結果，約 25%のバグ原因がパターン記述言語で表現可能で，それによってバグの原因となるコード部分が高い精度で検出できることを確認した．このことから，この手法がソースコードからのバグの検出に対して有効であると考えられる．また，抽出されたコード部分に対する検証を行う際，過去のバグ情報を参照できることで，検証すべき観点が明確であるため，確認作業も短時間で完了した．

今回は，ある一時点でのソフトウェアに対して照合作業を行い，バグを発見するという評価実験を行ったが，これを保守工程における変更ソースコードに対しても適用し，さらにバグの発見やコード検証に有効であることを確認したい．

また，今後の課題としては，この手法を開発プロセスへのコード検証手法として取り入れる場合の全体的な流れを想定し，それを支援する方法を確立したいと考えている．その際，以下のような点が課題になると考えられる．

- バグ報告データから潜在コーディング規則を系統的に取り出す方法．
- 既存のコード解析プログラムなどと組み合わせた高速なマッチングシステムの開発．
- 抽出されたコード部分の提示方法とバグ情報の参照支援ツールの開発．

参考文献

[1] A. A. Andrews, M. C. Ohlsson, and C. Wohlin, “Deriving fault architectures from defect history,” *J. of Software Maintenance: Research and Practice*, Vol. 12, No. 5, pp. 287 - 304, Sept.-Oct. 2000.

[2] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, “Does code decay? Assessing the evidence from change management data,” *IEEE Trans. on Software Engineering*, Vol. 27, No. 1, pp. 1 - 12, Jan. 2001.

[3] M. Fowler, “Refactoring: Improving the design of existing code”, Addison-Wesley, 1999.

[4] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, “Predicting fault incidence using software change history,” *IEEE Trans. on Software Enginerring*, Vol. 26, No. 7, pp. 653 - 661, July 2000.

[5] C. P. Hollocker, “Software reviews and audit handbook,” p. 162, John Wiley & Sons, 1990.

[6] W. S. Humphrey, “A discipline for software engineering,” Addison-Wesley, 1995.

[7] 海尻賢二, “プログラムパターンの認識及びリバースエンジニアリングツール,” 信学技報, SS2000-20, pp. 17 - 24, Sep. 2000.

[8] 河合茂樹, 山本晋一郎, 阿草清滋, “既存プログラムからの規範パターン獲得とそれに基づくコーディングチェッカ,” 日本ソフトウェア科学会 FOSE'97, pp. 99 - 106, Dec. 1997.

[9] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl, “Data mining for predictors of software quality,” *Int'l J. of Software Engineering and Knowledge Engineering*, Vol. 9, No. 5, pp. 547 - 563, 1999.

[10] F. Macdonald, and J. Miller, “A comparison of tool-based and paper-based software inspection,” *Empirical Software Engineering*, Vol. 3, No. 3, Autumn 1998.

[11] A. Monden, S. Sato, K. Matsumoto, and K. Inoue, "Modeling and analysis of software aging process," *F. Bomarius and M. Oivo (Eds), Lecture Notes in Computer Science*, Vol. 1840, pp. 140 - 153, 2000.

[12] A. Monden, S. Sato, and K. Matsumoto, "Capturing industrial experiences of software maintenance using product metrics," *Proc. 5th World Multi-Conference on Systemics, Cybernetics and Informatics*, Vol. 2, pp. 394 - 399, July 2001.

[13] 毛利幸雄, 菊野亨, 鳥居宏次, “レビューで用いるチェックリストの作成法の提案,” 第 11 回ソフトウェア信頼性シンポジウム論文集, pp. 7-11, Nov. 1990.

[14] G. J. Myers, “The art of software testing,” John Wiley, New York, 1979.

[15] 小田まり子, 掛下哲郎, “パターンマッチングに基づいたＣプログラムの落とし穴検出方法,” 情処学論, pp. 2427 - 2436, Vol. 35, No. 11, 1994.

[16] S. Paul, and A. Prakash, “A framework for source code search using program patterns,” *IEEE Trans. on Software Engineering*, Vol. 20, No. 6, pp. 463 - 475, June 1994.

[17] N. F. Schneidewind and C. Ebert, “Preserve or redesign legacy systems?,” *IEEE Software*, Vol. 15, No. 4, pp. 14 - 17, July/Aug. 1998.

[18] “Sapid Home Page”, http://www.sapid.org/