変数に着目した部分プログラム抽出による コードレビュー手法の提案

M2013SE005 加藤大典
指導教員：蜂巣吉成

1 はじめに

ソフトウェア開発において，ソースコードの欠陥修正や品質改善を目的としてコードレビューが行われている．コードレビューでは，コーディング担当者 (以下,『コーダ』) が作成したソースコードをレビュー担当者 (以下,『レビュア』) が目視で検査することによって修正内容を明らかにする．レビュアから修正内容の報告を受けたコーダはコードを修正する．

レビューによって発見されたコードの問題は，発見された箇所以外にも同様に存在していることが考えられる．しかしながら，問題が発見される度に疑いのあるすべてのコードから同様の問題を探し出し修正していくという作業は，プロジェクトの遅れの原因となる．

本研究では，発見された問題に対する修正内容からコード変換パターンを作成し，コードに適用するという方法により修正を行う．コード変換パターンを作成する際には，発見された箇所と同様の問題をもつ他の箇所との間にあるコードの差分を吸収するために，修正内容の抽象化を行う．コード変換パターンは適切な抽象度で記述される必要がある．抽象度が高すぎる場合は意図しない箇所が変換され，低すぎる場合は意図した箇所が変換されないことから，コード変換パターンの記述は知識と経験を要する作業である．そこで本研究では，この作業を直感的に行えるようにするために，コードレビューに適した表現を用いてコード変換パターンの作成が可能となる方法を提案する．コードレビューにおいて，レビュアはある変数に着目しその変数の出現を追うことで検査を行っていることから，ある変数の出現をノード，その出現間の制御フローをエッジとする部分プログラムのグラフを表現として用いる．

2 対象とするコードレビュー

2.1 コードレビューの様式

本研究が対象とするコードレビューについて述べる．開発において使用されているプログラミング言語は C 言語とする．コードレビューの目的にはコードの欠陥修正や品質改善，内容理解，プロジェクトチームでの知識共有などがある [1]．本研究ではその中から欠陥修正や品質改善といった，修正作業を伴うものを扱う．すなわち対象とするコードレビューは，

1. レビュアがコードを検査し結果をコーダに報告
2. コーダが報告を元にコードを修正

という 2 つの手順からなるものとする．レビュアは目視によってコードを検査し，発見した問題をコーダに報告する．報告は自然言語で行われる．例えば，ソースコード 1 がレビューされるとする．

ソースコード 1 レビューを行うソースコード

```
1  ...
2  strcpy(path, "~/.profile");
3
4  if ( isSymbolicLink(path) ) return ERROR;
5
6  fd = open(path, O_RDONLY, 0);
7  if (fd < 0) return ERROR;
8  err = read(fd, buffer, size);
9
10 // ファイル内容へのアクセス
11 ...
```

4 行目はシンボリックリンクでないことを検査する処理である．このコードに対しては,「オープンしたファイルが，シンボリックリンク検査済みのファイルからすり替えられたものでないことを検査する処理を追加せよ」というレビュー結果報告が考えられる．

2.2 コードレビューのもつ課題

1 節でも述べたとおり，レビューで発見された問題は別の箇所にも同様の問題が存在していることが考えられる．これには次のような原因がある．

1. レビュアによる見落とし
2. レビュー対象ソースコードの限定
3. レビューとコーディングの同時進行

2.3 解決のアプローチ

本研究では，発見された問題に対する修正内容から抽象化されたコード変換パターンを作成するというコードの修正方法を用いる．このコード変換パターンをコードに適用することで同様の問題をもつ他の箇所も修正することができる。コード変換パターンの作成はコードレビューに適した表現を用いて行う．そのために，レビュアがレビューを行う際の思考に着目する．上野らの研究 [2] では，コードレビューにおいてレビュアがコードを読む際の特徴的行動として次の 3 つが述べられている．

1. スキャンパターン
2. 宣言文確認パターン
3. 変数出現確認パターン

1 は「レビュー開始から一定時間，コード全体を眺めるように読み，全体の流れを把握する」，2 は「ある変数が初めて出現した際，その変数の宣言部を確認する」，3 は「ある変数が出現した際，その変数の直前の出現を確認する」というものである．これより，レビュアはプログラム全体の流れを把握したあと，流れに沿って各変数を順番に着目し，その出現箇所間の関係を意識しながらコードを検査していると言える．これは，思考の中で変数に着目し，その変数に関する部分プログラムを抽出することで検査しているとも言える．よって，本研究では対象

のプログラムから1つの変数に関する部分プログラムを抽出する．

部分プログラムの抽出方法として，プログラムスライスがある．これは特定の変数の出現に着目し，制御・データ依存関係を持たないコードを省略する部分プログラム表現である．しかしながら，省略後も平均してコード全体の1/3の量を持つことから [3]，部分プログラムとしての抽象度は低い．よって，コードレビューにおいて同様の問題点をもつコード間の差分を吸収する表現としては適切でない．そこで，本研究ではより抽象度の高い表現として『コードレビューグラフ』を提案する．

3 提案するコードレビュー手法

3.1 概要

本研究では，プログラムスライスと比較してより抽象度の高い部分プログラム表現である『コードレビューグラフ』を提案する．コードレビューグラフにより，コードの修正をグラフ操作に置き換える．さらに，このコードレビューグラフを操作するための表現として,『グラフ操作パターン』を提案する．図1にコード修正の概略を示す．

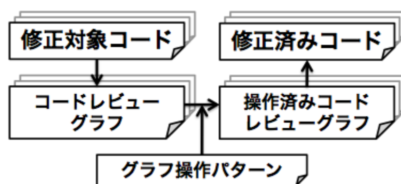


図1 グラフ操作によるコード修正

コードレビューの流れは次のとおりである．

1. レビュアによる変数選択と修正点の指摘
2. レビュアによるコードレビューグラフの作成
3. コーダによるグラフ操作パターンの作成
4. コーダによるグラフに対する変換の適用

1ではレビュアが変数の流れに着目しながら対象のプログラムを検査する．修正点を発見した場合は修正すべき問題に関する変数1つを『修正対象変数』として選択し，どのように変更すべきかについてコメントを記述する．2ではレビュアが対象のプログラムから修正対象変数におけるコードレビューグラフを作成する．3ではコーダが対象のプログラム，レビュアによるコメント，コードレビューグラフを参照しながらグラフ操作パターンを作成する．4ではコーダが対象のプログラム中のすべての変数におけるコードレビューグラフに対して，グラフ操作パターンによる変換を適用する．

3.2 コードレビューグラフ

制御フローグラフを

$$CFG = (S, CF)$$

S：プログラム中の文を表すノードの集合

CF：制御フローを表すエッジの集合

とするとき，変数 x のコードレビューグラフ CRG は

$$CRG = (RS, RF)$$
$$RS = \{p \mid p \in S, hasVar(p, x)\}$$
$$RF = \{\langle p, q\rangle \mid p, q \in RS, isReachable(p, q, CF)\}$$

と表される．ただし，$hasVar(s, x)$ は「ノード s に対応する文に 変数 x が出現する」，$isReachable(p, q, F)$ は「エッジの集合 F によるノード p から q への 経路が存在する」ことを表す関数とする．コードレビューグラフには，補助的に制御フローの開始と終了を表すノードを追加する．これはプログラム中の文に対応しない仮想的なノードである．例えば，ソースコード2に対し，変数bにおけるコードレビューグラフを作成すると図2になる．

ソースコード2 ソースコード

```
1 a = 2;
2 b = 3;
3 if( c > 0 ){
4    b = b + a;
5 }
6 x = a * a;
7 y = b * b;
```

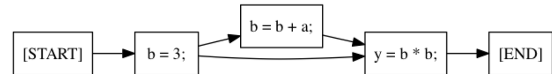


図2 コードレビューグラフ

3.3 グラフ操作パターン

グラフ操作パターンは2つの抽象コードレビューグラフをもち，それぞれのグラフで操作前と操作後を表すことによりグラフ操作を表現する。グラフ操作パターン GOP は抽象コードレビューグラフ $ACRG$ を用いて

$$GOP = (ACRG_{before}, ACRG_{after})$$
$$ACRG_{before} = (ARS, RF)$$
$$ACRG_{after} = (ARS', RF')$$

ARS：抽象化された RS の要素の集合

$$ARS' = ((ARS \setminus ARS_{remove}) \cup NS)$$

ARS_{remove}：削除されるコードを表すノードの集合

NS：新しく追加されるコードを表すノードの集合

$$RF = \{< p, q > \mid p, q \in ARS, isReachable(p, q, CF)\}$$
$$RF' = \{< p, q > \mid p, q \in ARS', isReachable(p, q, CF)\}$$

と表される．

$ACRG_{before}$ と $ACRG_{after}$ はそれぞれ『Beforeグラフ』と『Afterグラフ』と呼称する．グラフ操作パターンを作成するには，まずBeforeグラフから作成する．Beforeグラフの作成には修正対象となるコードのコードレビューグラフを用いる．まず，コードレビューグラフから修正すべき箇所の部分グラフを抽出する．次に抽出した部分グラフについて，ノードを抽象化し抽象コードレビューグラフに変換する．ノードの抽象化にはノード属性を用いる．ノード属性には次のように定義済みの5つの属性と，ユーザ定義がある．

- Declare：対象変数の宣言
- Assign：対象変数への代入

- Use：対象変数の使用
- Start：コードレビューグラフの開始ノード
- End：コードレビューグラフの終了ノード
- ユーザ定義

ユーザ定義属性は，定義済みのノード属性のみで抽象コードレビューグラフを構築すると抽象度が高すぎる場合に用いるものであり，そのノードの特徴を具体的な条件で指定する．例えば，目的のノードがファイルオープンを行っているノードであれば，open 関数や fopen 関数が出現するコードであることを指定する．次に Before グラフを元とした After グラフを作成する．After グラフは Before グラフに対して，ノードを削除するか，NS としてノードを追加することで作成する．追加する場合は，そのノードに対応するコードを与える必要がある．コードレビューグラフにグラフ操作パターンによる操作を適用するには，Before グラフを用いたマッチングを行い，マッチした箇所に Before グラフと After グラフの差分に基づいてノード追加・削除の操作を行う．

4 評価

4.1 実装

コードレビューグラフが自動的に作成できることを確認するために，ツールの実装を行った．現状のツールでは，コードは 1 つの関数ブロック内にあるコードのみに限られており，複数の関数にまたがるコードは対象外である．また，制御文は if 文による分岐と while 文によるループのみに対応している．

ツールに対しソースコードを入力として与えると，コードレビューグラフが DOT 形式で出力される．ノードには ID が与えられ，ID に対応するコードを別途テキストファイルとして出力する．本稿に掲載しているコードレビューグラフの図は，理解性を高めるために，出力されたグラフを手作業で編集し，対応コードをラベルとして表示するようにしたものである．

4.2 提案手法の適用

提案手法を用いて典型的なコード修正が行えることを確認する．IPA 独立行政法人 情報処理推進機構の Web サイトで公開されている「セキュア・プログラミング講座」[4] を参考に，掲載されていたセキュリティに関する問題をもつコードを作成し，提案手法による修正を適用した．ここではファイルレースコンディション攻撃対策への適用例について述べる．これは 2 節で挙げた，ソースコード 1 に対し「ファイルのすり替えを防止せよ」というレビュー報告に対する修正である．コードを修正するには，処理を (1) lstat 関数によってファイル属性値を取得，(2) 取得したファイル属性値によりシンボリックリンクを検査，(3) 取得したファイル属性値を fstat 関数によって取得するファイル属性値と比較，という流れにすればよい．よって，ソースコード 1 からシンボリックリンクを検査する関数を削除し，新たに「lstat 関数によってファイル属性値を取得しシンボリックリンクを検査する処理」(ソースコード 3) と，「そのファイル属性値を fstat 関数によって取得するファイル属性値と比較する処理」(ソースコード 4) を追加する．

ソースコード 3 追加する処理のコード A

```
1 struct stat lstat result;
2 if (lstat(path, &lstat result) != 0)
3     return − 1;
4 if ((lstat result.st mode & S IFMT) == S IFLNK)
5     return ERROR;
```

ソースコード 4 追加する処理のコード B

```
1 if (fstat(fd, &fstat_result) != 0) {
2   close(fd);
3   return −1;
4 }
5 if (lstat result.st ino != fstat result.st ino
6      || lstat result.st dev != fstat result.st dev) {
7   close(fd);
8   return − 1;
9 }
```

修正対象変数を変数 path とすると，コードレビューグラフは図 3 のようになる．検査処理を削除し，そこへソースコード 3 のコード A，オープン後にソースコード 4 のコード B を追加したいので，図 4 のグラフ操作パターンを作成する．Before グラフではノードに「$1」「$2」という名前が付けられ，After グラフでは$2 が後方参照されている．「has "〜"」は「字句〜の出現がある」というユーザ定義ノードを表しているとする．「A」と「B」はそれぞれコード A(ソースコード 3) とコード B(ソースコード 4) に対応する．

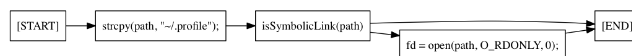
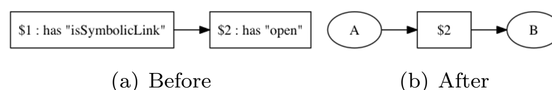


図 3 図 1 のコードレビューグラフ



(a) Before (b) After

図 4 グラフ操作パターン

これを図 3 に適用すると，操作後のコードレビューグラフは図 5 のようになる．これに基づきソースコード 1 を修正するとソースコード 5 のようになる．

図 5 操作後のコードレビューグラフ

ソースコード 5 修正後のソースコード

```
1 ...
2 strcpy(path, "~/.profile");
3
4 struct stat lstat result;
5 if (lstat(path, &lstat result) != 0)
6     return − 1;
7 if ((lstat result.st mode & S IFMT) == S IFLNK)
8     return ERROR;
```

```
9
10 fd = open(path, O_RDONLY, 0);
11
12 if (fstat(fd, &fstat_result) != 0) {
13   close(fd);
14   return −1;
15 }
16 if (lstat result.st ino != fstat result.st ino
17     || lstat result.st dev != fstat result.st dev) {
18   close(fd);
19   return − 1;
20 }
21
22 if (fd < 0) return ERROR;
23 err = read(fd, buffer, size);
24
25 // ファイル内容へのアクセス
26 ...
```

この他にも，表 1 の修正項目について提案手法による修正を適用した．

表 1 提案手法の適用

修正項目	適用の可否
シンボリックリンク攻撃対策	可
ファイルレースコンディション攻撃対策	可
バッファオーバーフロー対策	不可
ページング抑制	可
ディレクトリトラバーサル攻撃対策	可
整数オーバーフロー対策	不可
入力値の検証処理追加	可
入力値の無害化処理追加	可

4.3 考察

典型的なコード修正に対する提案手法の適用において，「バッファオーバーフロー対策」と「整数オーバーフロー攻撃対策」に対しては，提案手法は有効ではなかった．これはバッファオーバーフローと整数オーバーフローはデータ依存関係により引き起こされるからである．着目している変数以外の変数を省略しているコードレビューグラフ上では依存関係を追うことができず，防衛的な対策をとることしかできない．例えば、バッファオーバーフロー対策であれば，バッファサイズを超えて処理を行わない安全なライブラリ関数を使用することにより，バッファサイズ以上のデータが代入されないようにすることができる．しかしながらこの対策ではバッファサイズを超えたデータが途中で切り捨てられ，プログラムが仕様通りに動作することを妨げる．データの切り捨てを防ぐためには，データ依存関係のある変数のサイズを調べ，バッファにそれ以上のサイズを設定する必要がある．

残りの項目に関しては提案手法が有効であった．特に「ページング抑制」と「入力値の検証処理追加」については，ユーザ定義属性を用いないグラフ操作パターンによって修正を行うことができた．

5 関連研究

抽象化されたコード変換を行うことができるプログラムのマッチング及び変換環境には TEBA[5] や Coccinelle[6] がある．変換にはパターン変換記述 (TEBA) や Semantic Patch Language(Coccinelle) の記述法を習得する必要があるが，細かな条件指定による柔軟な変換が記述可能である．このような既存の変換環境では操作対象はプログラムであり，汎用的である反面，操作前と操作後の状態がイメージしづらい．これが変換の抽象化が難しい原因であると考えられる．本研究では操作対象をソースコードからグラフへ置き換えることにより，操作対象そのものがプログラムの振る舞いを視覚化するビューとなった．これにより操作前と操作後の状態をイメージしやすくなったと考えられる．

6 おわりに

本研究では，コードレビューにおいて，コードレビューに適した部分プログラム表現を用いてコード変換パターンを作成し，コードを修正することが可能となる方法を提案した．今後の課題は，グラフ操作パターン作成におけるコードレビューグラフからの部分グラフ抽出基準やユーザ定義属性の定義方法を拡充すること，グラフ操作パターンによるコードレビューグラフの操作が自動化された環境を提供すること，レビューにより発見された箇所以外の問題が正しく修正できるか評価することである．

参考文献

[1] Alberto Bacchelli, Christian Bird, “Expectations, Outcomes, and Challenges of Modern Code Review,” Proceedings of the 2013 International Conference on Software Engineering, ICSE ’13, pp. 712–721, 2013.

[2] 上野秀剛, 中村匡秀, 門田暁人, 松本健一 “プログラマの視線を用いたレビュー性能の要因分析,” ソフトウェア工学の基礎 XIII 日本ソフトウェア科学会 FOSE 2006, pp. 103–112, 2006.

[3] David Binkley, Loyola C. Maryland, Nicolas Gold, Mark Harma, “An Empirical Study of Static Program Slice Size,” ACM Transactions on Software Engineering Methodology, Vol. 16, No. 2, pp. 8, 2007.

[4] IPA 独立行政法人 情報処理推進機構, “セキュア・プログラミング講座,” https://www.ipa.go.jp/security/awareness/vendor/programmingv2/, 2007.

[5] 吉田敦, 蜂巣吉成, 沢田篤史, 張漢明, 野呂昌満, “属性付き字句系列に基づくプログラム書換え支援環境,” 情報処理学会論文誌, Vol. 53, No. 7, pp. 1832–1849, 2012.

[6] Gilles Muller, Julia Lawall, Jesper Andersen, Julien Brunel, Rene Rydhof Hansen, Yoann Padioleau, Nicolas Palix, “Coccinelle: A Program Matching and Transformation Tool for Systems Code,” http://coccinelle.lip6.fr/, 2009.