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.

ソースコードの自動進化に向けて

村上　寛明†　　堀田　圭佑†　　肥後　芳樹†　　楠本　真二†

† 大阪大学大学院情報科学研究科　〒 565–0871 大阪府吹田市山田丘 1–5
E-mail: †{h-murakm,k-hotta,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　ソースコードの変更作業は難しく，誤った方法で変更してしまったり，変更すべき箇所を見逃してしまったりという人的過失が起こることが多い．このような過失が起こった場合には，追加の変更作業が必要になるだけではなく，そのような連続した変更作業自体がソフトウェアの品質にとって望ましくない．著者らは，ソースコード変更作業における人的過失を解決する手段の 1 つとして，ソースコードの自動変更手法についての研究を行っている．本稿ではその第一歩として，Java メソッドにおいて次の変更でどのようなプログラム要素が削除および追加されるのかを予測する手法を提案する．

キーワード　ソフトウェア進化，コード自動変更，ソースコード解析

Towards Automated Code Evolution

Hiroaki MURAKAMI†, Keisuke HOTTA†, Yoshiki HIGO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University
Yamadaoka 1–5, Suita-shi, 565–0871 Japan
E-mail: †{h-murakm,k-hotta,higo,kusumoto}@ist.osaka-u.ac.jp

Abstract　Changing source code is not an easy task. Developers sometimes change source code in wrong ways or overlook code fragments to be changed. Those mistakes require additional cost to change source code in the correct ways and such repeated changes themselves are not good to software quality. We are conducting challenging research on automated code change as a countermeasure for human errors in source code changes. As a first step of this research, in this paper, we propose a technique to predict what kinds of program elements are deleted and added in the next change on Java methods.

Key words　Software evolution, Automated code change, Source code analysis

1. ま え が き

ソフトウェアの保守において，ソースコードの変更作業は高い割合を占める活動である [1]．しかし，ソースコードの変更作業は難しい．誤った方法で変更をしてしまったり，変更すべき箇所を見逃してしまったりという人的過失が発生する場合が多い．このような過失が起こってしまった場合には，追加の変更作業が必要になるだけではなく，そのような連続した変更作業自体がソフトウェアの品質に望ましくない [2]．そのため，ソースコードの変更作業を支援し，作業の自動化を促進する技術は有益である．

これまでにさまざまな変更支援手法が提案されている．例えば，変更予測モデルを構築し，頻繁に変更されるモジュールを予測する手法がある [3]〜[5]．システムの中でもあまり変更が行われないインターフェースに特化した変更予測手法も存在する [6]．これらの手法を用いることで，たとえば，バグが頻発しそうなモジュールに対して集中的にレビューを行うといった，限られた人的資源を有効に活用することが可能になる．また，同時に修正されるプログラム要素（ファイルや関数）を予測する手法も存在する [7]〜[12] ．これらの手法を用いることにより，開発者が機能追加やバグ修正のためにある 1 つのプログラム要素を修正した場合に，同時に修正されるべき要素を開発者に提示することができる．このような支援を行うことにより，変更漏れの発生を予防することができる．

ソースコードの変更の中でもバグ修正については特に研究が活発である．例えば，複数のバグ間において対処する優先度を決定する手法 [13], バグの原因となる箇所を特定する手法 [14], [15]，バグ修正が正しく行われたことを確認するための手法 [16] が存在する．

さらには，ソースコードに対して自動的にバグ修正を行う手法（バグ修正用のパッチを生成する手法）も提案されている．Perkins らが開発したツール ClearView は，対象プログラムを

動的解析することにより，満たされていない不変条件を自動検出する．そして，その不変条件が満たされるようなパッチを自動生成する [17]．Wei らは，不変条件だけではなく，事前条件や事後条件も利用し，それらが満たされるようにプログラムを自動的に修正する手法を提案し，ツール AutoFix-E を開発している [18]．Wei らの手法は，事前条件や事後条件も利用するので，Perkin らの手法に比べて多くのバグに対応できるが，事前条件や事後条件が記述されたクラスやメソッドでなければ適用することができない．それに対して，Perkin らの手法では，Daikon [19] というツールを利用することで自動的に不変条件を特定するので，対象クラスに不変条件が予め記述されている必要がない．Jin らは，並列処理における原子性違反に関するバグを修正するためのパッチを自動的に生成する手法を提案している [20]．Weimer らが開発した GenProg というツールは，遺伝的プログラミング (注1) を用いてバグ修正用のパッチを自動生成する [21], [22]．さらに，文献 [23] では，GenProg を用いた大規模な実験を行っており，105 のバグのうち 55 個が自動的に修復できたと報告している．

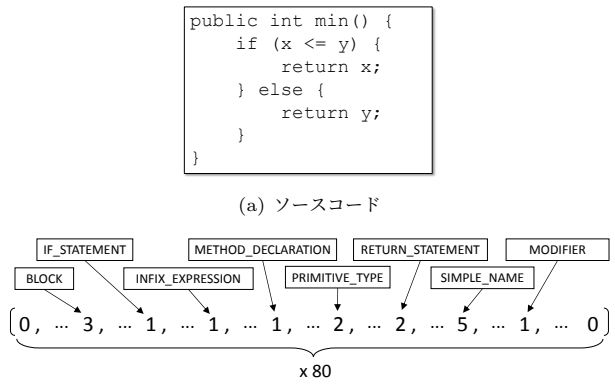
ソースコードの変更は，バグ修正だけではなく，機能追加や機能変更等を目的としても行われる．我々の研究グループでは，バグ修正や機能追加といった変更の種類に依存しないソースコードの自動変更についての研究を行っている．本手法が実現すれば，開発者がソースコード変更時に行う作業量が軽減される．我々の手法によって提示された変更が良ければそのまま変更の許可を行い，もし悪ければ自分自身で変更を行うという変更プロセスになる．開発者行自身が行う作業量が軽減されるために効率的にソースコードの変更作業を行うことが可能になる．さらに，ソースコードを自動的に変更することが可能になれば，全く人の手を介さずにソースコードを進化させていくことも可能になるかもしれない．

本報告ではこの研究の第一歩として，Java メソッドを対象として，次の変更においてどのようなプログラム要素が削除および追加されるのかを予測する手法を提案する．本報告が提案する変更予測手法には下記の制限がある．

- 変更予測の対象は Java メソッドの内部に加わる変更のみである．Java 言語以外のプログラミング言語や，Java 言語であってもクラスの追加等メソッドの外部に加わる変更は対象外である．
- 複数のメソッドに同時に加わる修正を予測することはできない．本手法が対象にしているのは，各メソッドが次にどのように変更されるかである．
- 本手法は変更後のソースコードを生成することはできない．本手法は，追加および削除されるプログラム要素を予測するのみである．

2. 提案手法

提案手法は，入力として与えられた対象ソフトウェアの開発履歴情報を分析し，与えられたメソッドに次の修正が加えられたときにどのようなプログラム要素が追加または削除されるのかを予測するモデルを構築する．提案手法によって出力されたモデルを，予測の対象となるメソッド (以降，予測対象メソッド) に適用することで，それらに対して次にどのようなプログラム要素が追加されるのか，あるいはどのようなプログラム要素が削除されるのかを予測することができる．提案手法ではモデルの構築に際し，それぞれのメソッドをプログラム要素の出現数を要素とするベクトル (以降，状態ベクトル) として表現する．本節では．はじめにメソッドの状態ベクトルについて触れた後，提案手法の処理の内容について述べる．



(a) ソースコード

(b) 状態ベクトル

図 1　状態ベクトルの例

2.1 状態ベクトル

提案手法では，それぞれのメソッドを状態ベクトルを用いて表現する．状態ベクトルとは，メソッド中に存在するプログラム要素の情報を用いて作成されるベクトルである．ここでプログラム要素とは，`if` 文などの文の他に，識別子名，変数宣言，メソッド宣言などが含まれる．それぞれのプログラム要素の出現数が状態ベクトルの 1 つの要素となる．

本研究における実装では，プログラム要素の特定に抽象構文木を使用し，抽象構文木の頂点の種類 1 つを 1 つのプログラム要素とする．なお本実装では抽象構文木の構築に JDT(Java Development Tools) を用いた．JDT は Java を対象として抽象構文木を構築する機能を提供しており，83 種類の頂点が定義されている．本実装ではこれらの頂点のうち，コメントに関する 3 種類の頂点を除外する．従って，本実装における状態ベクトルの次元は 80 となる．

図 1 に状態ベクトルの例を示す．図 1(a) に示すメソッドから状態ベクトルを生成すると，図 1(b) のようになる．このメソッド中には，80 の構文要素のうち 8 のみが出現している．従って，80 の要素のうち 8 の要素は 1 以上の値を持ち，残りの 72 の要素の値は 0 となっている．なお，図 1(b) ではベクトルの先頭と末尾を除き，値が 0 である要素の記述を省略している．

以降，状態ベクトルの要素となるそれぞれのプログラム要素を，$A_1, \ldots, A_{80}$ と表記する．

2.2 用　　語

本小節では，以降の説明に用いる諸用語の定義を与える．

はじめに，あるメソッド m の状態ベクトルを $\vec{v_m}$ =

(注1)：Genetic Programming

$(x_{1_m}, \cdots, x_{80_m})$ とし，メソッド m のソースコードの文字列表記を s_m とする．次に，対象リポジトリ中のすべてのコミットからなる集合を C とし，あるコミット $c \in C$ の前後のリビジョンをそれぞれ rb_c，ra_c とする．また，あるリビジョン r に存在するすべてのメソッドからなる集合を M_r と表記する．

さらに，あるコミット c について，コミット前のリビジョン rb_c に存在するメソッド m_{rb_c} とコミット後のリビジョン ra_c に存在するメソッド m_{ra_c} を考える．このとき，m_{rb_c} と m_{ra_c} が同一のメソッドであれば，$m_{rb_c} = m_{ra_c}$ と表記する．なお，提案手法では連続するリビジョン間に存在する 2 つのメソッドが同一であるかの判定に，CRD の類似度に基づく手法 [24] を用いる．この手法は，ソースコード中に存在する各ブロックについて，それらの位置情報を表す CRD [25] を算出し，それを用いてリビジョン間でのブロックの対応付けを行う手法である．この手法の特長として，ブロックが他のファイルに移動した場合でもその移動を追従することが可能であるという点がある．

2.3 提案手法の概要

提案手法が行う処理は大きく以下の 2 つのステップから成る．

STEP1: 学習用データの抽出

STEP2: 予測モデルの構築

STEP1 は対象ソフトウェアの開発履歴情報が蓄積されたリポジトリを入力とし，予測モデルを構築するために使用する学習用データを抽出する．次に STEP2 では，STEP1 で得られた学習用データをもとに，予測モデルを出力する．以降の小節でそれぞれの処理について述べる．

2.4 STEP1: 学習用データの抽出

このステップでは，入力として与えられたリポジトリからすべての学習用データを取得する．ここで学習用データとは，以下の条件をすべて満たす 2 つの状態ベクトル $\vec{v_{m_b}}$，$\vec{v_{m_a}}$ の組である．

$$\exists c \in C(m_b \in M_{rb_c} \wedge m_a \in M_{ra_c}) \tag{1}$$

$$m_b = m_a \tag{2}$$

$$s_{m_b} \neq s_{m_a} \tag{3}$$

すなわち，学習用データ $d = (\vec{v_{m_b}}, \vec{v_{m_a}})$ とは，あるコミットで変更が加えられたメソッドの変更前後それぞれのソースコードから得られた状態ベクトルの組である．このステップでは，与えられたリポジトリ中の各コミットを分析し，取得可能な学習用データをすべて取得する．以降，リポジトリから得られたすべての学習用データからなる集合を TD と表記する．

2.5 STEP2: 予測モデルの構築

このステップでは，STEP1 で得られた学習用データの集合 TD から，予測対象メソッドに次の修正が加わった後の状態ベクトルを予測するモデルを構築する．すなわち，与えられた予測対象メソッド m の状態ベクトル $\vec{v_m}$ から，メソッド m に次の修正が加わった後の状態ベクトル $\vec{v_{m'}}$ を予測するモデルを構築する．この予測モデルを用いて次の修正が加わった後の状態ベクトルを予測することで，次の修正によって生じるそれぞれのプログラム要素の出現数の増減を知ることができる．以降，予測対象メソッドの状態ベクトルを $\vec{v_t} = (x_{1_t}, \cdots, x_{80_t})$ とし，予測結果として出力される状態ベクトルを $\vec{v_p} = (y_{1_p}, \cdots, y_{80_p})$ と表記する．

提案手法では，予測モデルの構築に線形重回帰モデルを用いる．なお，提案手法では状態ベクトルの個々の要素それぞれについて 1 つずつモデル式を構築する．従って，状態ベクトルの次元数と同じ数のモデル式が構築される．

それぞれのモデル式は，予測対象メソッドの状態ベクトルのすべての要素 $(x_{1_t}, \cdots, x_{80_t})$ から選択されたいくつかの要素を説明変数とし，状態ベクトルの 1 つの要素 $(y_{i_p}$，ただし $i \in \{1, \ldots, 80\})$ を目的変数とする．すなわち，i をそれぞれ 1 から 80 までの整数としたとき，それぞれのモデル式は以下のように表現される．

$$y_{i_p} = \beta_0 + \sum_{j=1}^{80} \beta_j x_{j_t} \tag{4}$$

ただし，

$$\begin{cases} \beta_j \neq 0 \ (\text{要素 } A_j \text{が説明変数として選択されている}) \\ \beta_j = 0 \ (otherwise) \end{cases} \tag{5}$$

提案手法では，与えられた学習用データをもとに，それぞれのモデル式について，用いる説明変数の選択並びに係数の決定を行う．それぞれのモデル式について独立して処理を行うため，使用する説明変数や係数はモデル式によって異なる．本研究における実装では，これらの処理を R を用いて行う．具体的には，係数の決定を lm 関数を用いて行い，説明変数の選択を step 関数による変数増減法を用いて行う．

3. 実　　験

3.1 準　　備

本実験の目的は，状態ベクトルを用いてソースコードの変更予測を行えるのか否かを確かめることである．本実験では，状態ベクトルのすべての要素を用いた予測と文以上の要素のみを用いた予測を行う．文以上の要素とは，ソースコードを抽象構文木で表現した際，`STATEMENT` より上位に属する要素である．図 2 は図 1(a) のソースコードを抽象構文木で表現した図である．この場合，図 2 における文以上の要素は白い頂点を指す．状態ベクトルのすべての要素数は 80 であるが，文以上の要素数は 40 である．

本実験の目的を達成するため，以下に示す 2 つの調査項目を設定した．

RQ1: 状態ベクトルのすべての要素を用いた予測はどのくらいの精度で行えるか．

RQ2: 状態ベクトルの文以上の要素のみを用いた予測はどのくらいの精度で行えるか．

実験対象はオープンソースソフトウェアの OpenYMSG, ArgoUML, Ant である．これらのソフトウェアは Java で記述されており，かつ Subversion でバージョン管理されている．実験対象の詳細を表 1 に示す．

3.2 手　　順

実験は以下のステップで行われる．

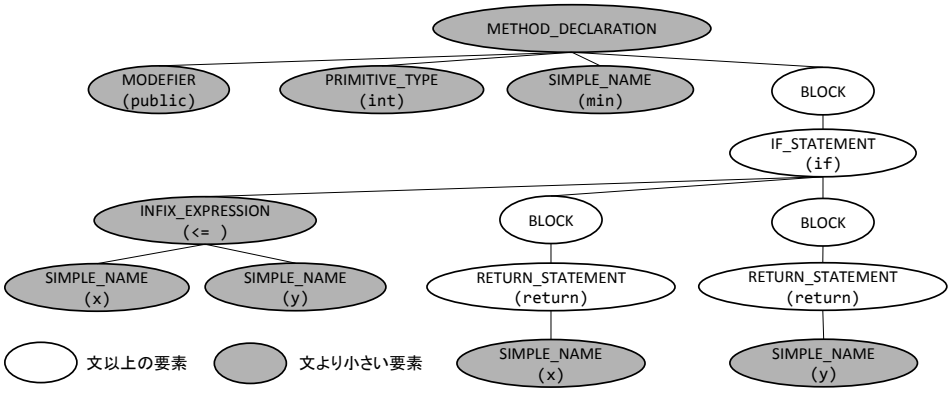


図 2　抽象構文木

表 1　実験対象ソフトウェア

名称	開始リビジョン（日付）	終了リビジョン（日付）	対象リビジョン数	終了リビジョンの総行数
OpenYMSG	1 (2007/04/17)	195 (2010/12/07)	155	163,545
ArgoUML	1 (1998/01/27)	19,893 (2012/07/10)	3,918	355,411
Ant	267,549 (2000/01/13)	1,233,420 (2012/01/20)	8,284	255,169

STEP1: 実験対象ソフトウェアのコミットの集合をリビジョン数が等しくなるように 5 つに分割する．ここで，分割されたコミットの集合をそれぞれ C_1, C_2, C_3, C_4, C_5 とする．

STEP2: 各コミットの集合における変更前後の状態ベクトルについて，変化した要素数が 5 以下である状態ベクトルの組を抽出する．本ステップを設定した理由は，変化した要素の数が多い状態ベクトルの組を学習用データとして用いると，精度の高いモデルを構築することが難しいと判断したためである．

STEP3: $n(2 \leqq n \leqq 5)$ に対し，$C_1, \cdots, C_{n-1}$ を学習用データとしてモデル構築を行い，構築したモデルを用いて C_n で行われるであろう変更を予測する．なお本ステップでは，状態ベクトルのすべての要素を用いた予測および文以上の要素のみを用いた予測を行う．

STEP4: 予測した C_n と実際の C_n を比較する．$n(2 \leqq n \leqq 5)$ であるため，1 つの実験対象ソフトウェアにつき 4 つの結果を得ることができる．

3.3 結　　果

図 3 は，状態ベクトルのすべての要素を用いた予測を Ant に対して行った結果である．図 3 の各グラフの横軸は予測した状態ベクトルと実際の状態ベクトルを比較したときの異なる要素の数であり，縦軸はその状態ベクトルの数である．例えば図 3(a) に着目すると，すべての要素の数を正しく予測できた状態ベクトルの数は約 14,000 であることが分かる．各コミットの集合において，全状態ベクトル数に対するすべての要素を正しく予測できた状態ベクトル数の割合は 60%〜70%である．OpenYMSG や ArgoUML に対しても，ほぼ同様の結果を得ることができた．

図 4 は，状態ベクトルの文以上の要素のみを用いた予測を Ant に対して行った結果である．図 4 における各グラフの横軸および縦軸が表しているものは図 3 と同様である．各コミットの集合において，全状態ベクトル数に対するすべての要素を正しく予測できた状態ベクトル数の割合は 80%〜90%である．OpenYMSG や ArgoUML に対しても，ほぼ同様の結果を得ることができた．

3.4 調査項目に対する解答

各調査項目について以下のように解答する．

RQ1 に対する解答: 状態ベクトルのすべての要素を用いた予測精度は **60%〜70%**である．本実験では状態ベクトルのすべての要素が一致して初めて予測できたとみなしているため，60%〜70%という値は高精度であると著者らは考えている．

RQ2 に対する解答: 状態ベクトルの文以上の要素のみを用いた予測精度は **80%〜90%**である．状態ベクトルのすべての要素を用いた予測に比べて精度が高くなった理由は，`SIMPLE_NAME` や `PRIMITIVE_TYPE` などの文より小さい要素が予測の対象から取り除かれたためである．`SIMPLE_NAME` や `PRIMITIVE_TYPE` などの要素はソースコード上において頻繁に表れるため，これらの要素の出現数を正確に予測するのは難しいと著者らは考えている．

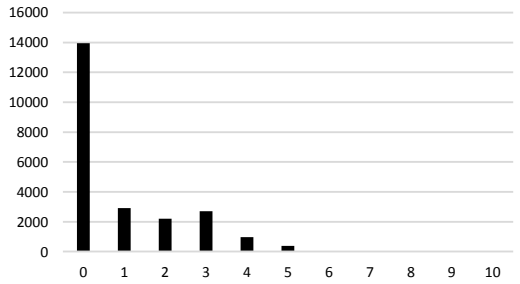
4. 妥当性への脅威

4.1 コミットの集合の分割

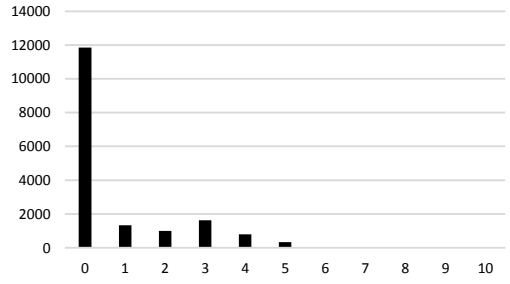
本実験では，実験対象ソフトウェアのコミットの集合をリビジョン数が等しくなるように 5 つに分割した．しかし，バージョンが上がるタイミングで分割したり，開発日数が等しくなるように分割するなど本研究とは異なる区切りで分割した場合，あるいは分割数を 5 以外に設定した場合，本研究で得られた結果とは異なる結果が導かれる可能性がある．しかし，コミットの集合の分割方法および分割数のすべての組み合わせについて実験を行うことは現実的ではない．

4.2 モデルの構築

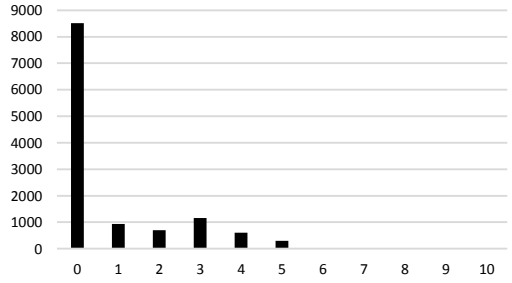
本実験では精度の高いモデルを構築するため，変化した要素数が 5 以下である状態ベクトルの組を学習用データとした．こ



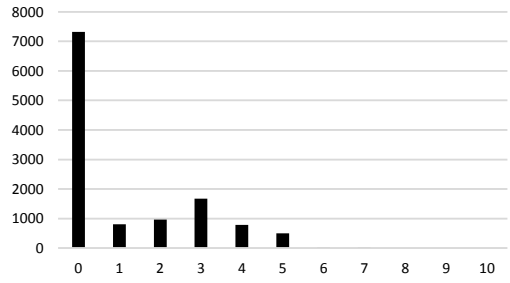
(a) C_2 に対する結果



(b) C_3 に対する結果

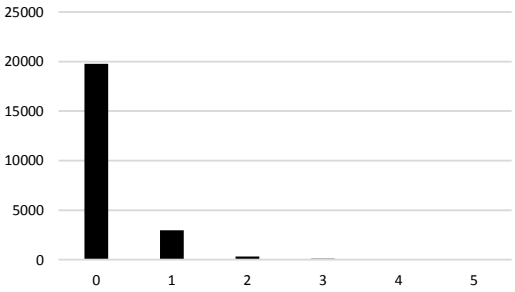


(c) C_4 に対する結果

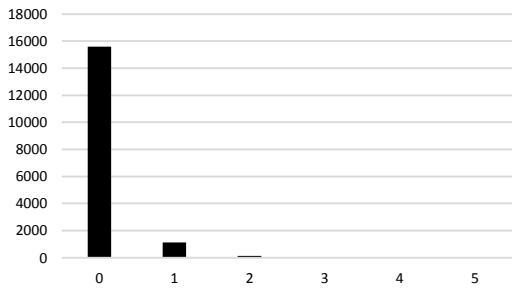


(d) C_5 に対する結果

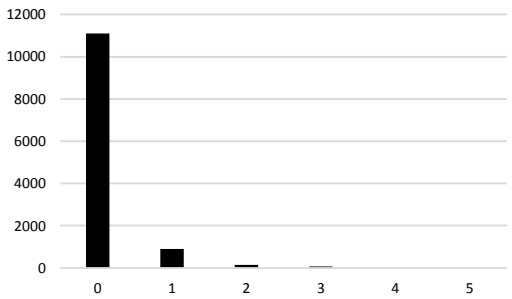
図 3　状態ベクトルのすべての要素を用いた場合における Ant に対する予測結果



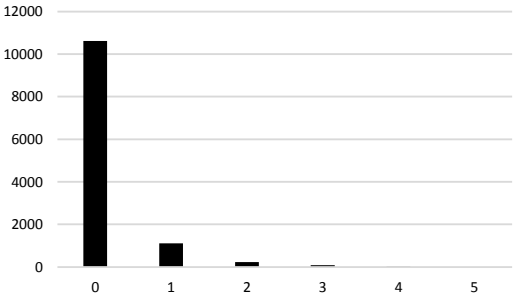
(a) C_2 に対する結果



(b) C_3 に対する結果



(c) C_4 に対する結果



(d) C_5 に対する結果

図 4　状態ベクトルの文以上の要素のみを用いた場合における Ant に対する予測結果

の閾値を変更してモデルを構築すると，本研究で得られた結果とは異なる結果が導かれる可能性がある．この閾値を小さくすると，より厳密な予測を行うことができるが，学習用データの減少に伴いモデルの精度が低くなる恐れがある．一方閾値を大きくすると学習用データは増えるが，一度に多くの変更が加えられたメソッドの状態ベクトルも学習用データに含まれてしまうためモデルの精度が低くなる恐れがある．本研究の最終目標はソースコード自動進化の実現であるため，できるだけ精度の高いモデルを構築することが求められる．

4.3　実験対象ソフトウェア

本研究では 3 つのオープンソースソフトウェアを対象にして実験を行った．しかし実装の都合上，実験対象ソフトウェアはすべて Java で記述されており，かつ Subversion で管理されていなければならないという制限がある．そのため Java とは異なる言語で記述されたソフトウェアに対して実験を行った場合，あるいは Subversion とは異なるバージョン管理システム（CVS，Git など）で管理されたソフトウェアに対して実験を行った場合，本研究で得られた結果とは異なる結果が導かれる

可能性がある．

5. あとがき

本稿では，Java メソッドを対象として，次の変更においてどのようなプログラム要素が追加及び削除されるのかを予測する手法を提案した．3 つの Java 言語で記述されたオープンソースソフトウェアを用いた実験の結果，文よりも大きなプログラム要素のみを予測した場合では，80%〜90%，全ての要素を予測した場合には 60%〜70%で予測が可能であった．

今後は，提案手法を拡張し，予測に基づいてソースコードを生成できるように研究を進めて行く予定である．ソースコードを生成するためには，どのようなプログラム要素が追加されるかだけではなく，どこに追加されるのかも予測しなければならない．現在のところは，プログラムミューテーションや遺伝的プログラミングを応用することを予定している．

謝　辞

本研究は，日本学術振興会科学研究費補助金基盤研究 (S)(課題番号：25220003)，挑戦的萌芽研究 (課題番号：24650011)，および文部科学省科学研究費補助金若手研究 (A)(課題番号：24680002) の助成を得て行われた．

文　献

[1] R.C. Seacord, D. Plakosh, and G.A. Lewis, Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices, Addison-Wesley Longman Publishing Co., Inc., 2003.

[2] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, “Change Bursts As Defect Predictors,” Proceedings of the 21st International Symposium on Software Reliability Engineering, pp.309–318, ISSRE ’10, 2010.

[3] E. Arisholm, L.C. Briand, and A. Foyen, “Dynamic Coupling Measurement for Object-Oriented Software,” IEEE Transactions on Software Engineering, vol.30, no.8, pp.491–506, Aug. 2004.

[4] M. Dagpinar and J.H. Jahnke, “Predicting Maintainability with Object-Oriented Metrics - An Empirical Comparison,” Proceedings of the 10th Working Conference on Reverse Engineering, pp.155–164, WCRE ’03, 2003.

[5] W. Li and S. Henry, “Object-oriented Metrics That Predict Maintainability,” Journal of Systems and Software, vol.23, no.2, pp.111–122, Nov. 1993.

[6] D. Romano and M. Pinzger, “Using Source Code Metrics to Predict Change-prone Java Interfaces,” Proceedings of the 27th International Conference on Software Maintenance, pp.303–312, ICSM ’11, 2011.

[7] H. Kagdi, “Improving Change Prediction with Fine-grained Source Code Mining,” Proceedings of the 22nd International Conference on Automated Software Engineering, pp.559–562, ASE ’07, 2007.

[8] H. Kagdi and J.I. Maletic, “Combining Single-Version and Evolutionary Dependencies for Software-Change Prediction,” Proceedings of the 4th International Workshop on Mining Software Repositories, pp.17–20, MSR ’07, 2007.

[9] R. Robbes, D. Pollet, and M. Lanza, “Logical Coupling Based on Fine-Grained Change Information,” Proceedings of the 15th Working Conference on Reverse Engineering, pp.42–46, WCRE ’08, 2008.

[10] P. Tonella, “Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis,” IEEE Transactions on SoftwareEngineering, vol.29, no.6, pp.495–509, June 2003.

[11] A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll, “Predicting Source Code Changes by Mining Change History,” IEEE Transactions on Software Engineering, vol.30, no.9, pp.574–586, Sep. 2004.

[12] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, “Mining Version Histories to Guide Software Changes,” Proceedings of the 26th International Conference on Software Engineering, pp.563–572, ICSE ’04, 2004.

[13] J. Anvik, L. Hiew, and G.C. Murphy, “Who Should Fix This Bug?,” Proceedings of the 28th International Conference on Software Engineering, pp.361–370, ICSE ’06, 2006.

[14] J.A. Jones and M.J. Harrold, “Empirical Evaluation of the Tarantula Automatic Fault-localization Technique,” Proceedings of the 20th International Conference on Automated Software Engineering, pp.273–282, ASE ’05, 2005.

[15] D. Saha, M.G. Nanda, P. Dhoolia, V.K. Nandivada, V. Sinha, and S. Chandra, “Fault Localization for Data-centric Programs,” Proceedings of the 19th SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp.157–167, ESEC/FSE ’11, 2011.

[16] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, “How Do Fixes Become Bugs?,” Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp.26–36, ESEC/FSE ’11, 2011.

[17] J.H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M.D. Ernst, and M. Rinard, “Automatically Patching Errors in Deployed Software,” Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, pp.87–102, SOSP ’09, 2009.

[18] Y. Wei, Y. Pei, C.A. Furia, L.S. Silva, S. Buchholz, B. Meyer, and A. Zeller, “Automated Fixing of Programs with Contracts,” Proceedings of the 19th International Symposium on Software Testing and Analysis, pp.61–72, ISSTA ’10, 2010.

[19] M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, and C. Xiao, “The Daikon System for Dynamic Detection of Likely Invariants,” Science of Computer Programming, vol.69, no.1-3, pp.35–45, Dec. 2007.

[20] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, “Automated Atomicity-violation Fixing,” Proceedings of the 32nd Conference on Programming Language Design and Implementation, pp.389–400, PLDI ’11, 2011.

[21] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “GenProg: A Generic Method for Automatic Software Repair,” IEEE Transactions on Software Engineering, vol.38, no.1, pp.54–72, Jan. 2012.

[22] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, “Automatically Finding Patches Using Genetic Programming,” Proceedings of the 31st International Conference on Software Engineering, pp.364–374, ICSE ’09, 2009.

[23] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, “A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each,” Proceedings of the 34th International Conference on Software Engineering, pp.3–13, ICSE ’12, 2012.

[24] Y. Higo, K. Hotta, and S. Kusumoto, “Enhancement of CRD-based Clone Tracking,” Proc. of the 13th International Workshop on Principles on Software Evolution, pp.28–37, IWPSE ’13, Aug. 2013.

[25] E. Duala-Ekoko and M.P. Robillard, “Clone Region Descriptors: Representing and Tracking Duplication in Source Code,” ACM Transactions on Software Engineering and Methodology, vol.20, no.1, pp.3:1–3:31, June 2010.