一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
SS2016-70(2017-03)

前処理命令による可変点を考慮した共変更ルール抽出

森　達也[†]　小林　隆志[†]　林　晋平[†]　渥美　紀寿[††]

[†] 東京工業大学　〒152–8552 東京都目黒区大岡山 2–12–1
[††] 京都大学　〒606–8501 京都市左京区吉田本町
E-mail: {tmori@sa., tkobaya@, hayashi@se.}cs.titech.ac.jp, atsumi.noritoshi.5u@kyoto-u.ac.jp

あらまし　プログラムを変更すると複数の依存箇所に変更が伝播するが，大規模なソフトウェア開発では開発者が必要な変更箇所を全て特定することは困難な作業となる．前処理命令を埋め込むことで実行環境の差異の吸収が可能なソフトウェアでは前処理命令が散在するため，必要な変更箇所の特定はより困難となる．本稿では，前処理命令の影響を受ける箇所に対する変更を考慮した共変更ルールの抽出手法を提案する．前処理命令の分岐条件と変更の関係を解析することで，改版履歴中の変更に属性を付与し，ルールを生成する．3 つの中規模 OSS を用いた評価実験の結果，提案手法によって得られた共変更ルールはリフト値の観点から質が高いことを明らかにした．また，既存手法と比較して推薦精度を表す *MRR* の最大値が向上することを示した．

キーワード　変更支援，ソフトウェアリポジトリマイニング，ソフトウェア保守，コンフィギュラブルソフトウェア

1. は じ め に

クロスプラットフォームソフトウェアの開発においては，OS の違いや CPU の命令セットアーキテクチャの違いなどの細かい差異を吸収する必要がある．また，プラットフォームの差異だけでなく，プロダクトライン型開発においても特定の機能を利用するかどうかを制御する必要がある [1]．本稿では，プログラムの一部を変更することによって，このような実行環境の差異の吸収や機能構成の変更が可能なソフトウェアをコンフィギュラブルソフトウェア (CS) と呼ぶ．CS の実現には，静的に構成を変更する方法と，動的に変更する方法の 2 つの実現手段があるが，本稿では前処理命令を用いて静的に構成変更を実現する CS を対象とする．

CS の開発においては，ソースコード中に前処理命令が散在するため，依存関係がより複雑となる．例えば，ファイル内に複数の前処理命令の記述が存在する場合には，前処理命令の条件によって有効になるコード断片が異なるため，同じファイルを変更した場合でも変更が波及する箇所が異なる．このように，CS の場合には開発者がソフトウェアの構成要素間の依存関係を全て把握することが困難となる．

必要な変更箇所に対して変更をし忘れてしまうとバグの原因となるため，開発者に必要な変更箇所を推薦する，プログラム変更支援が必要となる．プログラム変更支援では，改版履歴の分析に基づく手法が効果的である．改版履歴の分析に基づく変更支援手法では，Git や Subversion などのバージョン管理システムの改版履歴を分析対象とする．改版履歴に対して相関ルールマイニング [2] を適用することで，あるファイルが変更された際に高い確率で別のファイルも同時変更されることを示す共変更ルールという形で，ファイル間の依存関係が抽出できる．共変更ルールを用いることで，コミット時の変更漏れに対して十分な精度で推薦が可能であることが明らかになっている [3–6]．

共変更ルールに基づく変更推薦では，改版履歴中で頻繁に同時変更されたという情報のみに基いて推薦を行うため，ソースコード中の静的な解析は行わない．そのため，CS へ手法を適用する際に，前処理命令の情報を加味できていないという問題がある．そこで，本稿では前処理命令に対する静的な解析と改版履歴の分析を組み合わせた手法を提案する．変更されたファイルに対して静的解析を行い，変更箇所が前処理命令の影響を受けるかどうかを判定する．本稿では以後，ソースコード中の前処理命令の影響を受ける箇所を可変点コードと記述する．可変点コードに対する変更には，前処理命令の条件に用いられるマクロ変数名を情報として付与する．マクロ名の情報付与により，同じファイルに対する変更であっても，特定の可変点コードに変更があったことを考慮することが可能となる．このように，変更内容に関連のあったマクロ変数の情報を付与した改版履歴を生成することで，可変点コードに対する変更であることを考慮した共変更ルールを抽出する．このルールにより，特定の機能の修正のための同時変更や，関連の強い機能を同時に修正する場合に対して，より高精度な推薦が期待できる．

2. 関 連 研 究

Zimmermann らが提案した eROSE [3] は，同時に変更される可能性の高いファイル間，メソッド間の依存関係を相関ルールの分析方法を用いて共変更ルールという形で抽出しており，このルールに基づいて推薦を行う．Zimmermann らはコミット時の変更漏れに対する推薦という実験設定において，eROSE による推薦が正確であることを示した．しかし，共変更ルールの生成時には変更箇所などの静的な情報は加味されず，CS に対して手法を適用する場合には，前処理命令によって変化する依存関係を上手く扱えないという問題がある．

Dintzner らは，CS の改版履歴中の変更が機能に影響を及ぼす変更であることを自動で特定する FEVER [7] を提案した．

This article is a technical report without peer review, and its polished and/or extended version may be published elsewhere.
Copyright ©2017 by IEICE

FEVER では各コミットにおいて，KConfig に記述されたフィーチャーモデルに対しての変更や，Makefile に記述された機能とソースコード間の対応関係への変更，また，ソースコード中の前処理命令の範囲内に対する変更を解析することで，変更内容がどの機能に影響を及ぼすかを特定している．今西らは，前処理命令による制御構造と改版履歴中のソースコードの改変にどのような関係があるのかについて OSS を対象に調査を行っている [8]．調査の結果，改変に関連する前処理条件の全体構造において，改変頻度が高い構造は改変頻度が低い構造より分岐命令を複数含んでいることや，改変に関連する前処理条件の集合のうち複数リビジョンに渡っ て頻出する組は，複数の全体構造に含まれていることなどを明らかにした．Braz らは，特定のコンパイルスイッチの組み合わせで発生するコンパイルエラーを，改版履歴中の前処理命令の範囲内に対する変更を解析することで自動検出する ChangeConfigMX [9] を提案した．このように，改版履歴中の前処理命令の範囲内に対する変更を解析した研究は複数存在するが，変更推薦を行う上で前処理命令を考慮するという点が本稿と異なっている．

3. 可変点コードへの変更情報を加味した改版履歴の生成

3.1 概　　要

図 1 に，可変点コードに対する変更を考慮した改版履歴の生成手法の概略図を示す．改版履歴中において変更されたファイルに対し，変更差分解析，インクルード解析，関数コール解析の 3 種類の解析を行うことで，可変点コードに対する変更であるかを判定する．これらの解析の結果，可変点コードに対する変更であると判定されれば，変更に対してマクロ変数名を属性として付与する．本稿では，改版履歴中の変更に対しマクロ変数名を属性として付与することを，**変更に型をつける**と表現する．例えば，ファイル A が変更され，その変更箇所が X というマクロ変数を条件部にもつ前処理命令に関連するものであれば，X という型を付け A[X] と表記する．

元の改版履歴に含まれる型のついていない変更に加え，このように型付けを行った変更を追加することで，可変点コードに対する変更の情報を加えた詳細な改版履歴を生成する．この履歴を**型付きの改版履歴**と呼び，型付き改版履歴をマイニングして得られる共変更ルールを**型付きの共変更ルール**と呼ぶ．型付きの共変更ルールは A[X] ⇒ B[X] の形で表され，特定の前処理命令に関する変更であることを加味することができるため，推薦精度の向上が期待できる．

3.2 前処理命令に関する情報の抽出

ソースコード中の前処理命令の解析には，srcML [10, 11] を利用した．#if，#ifdef，#ifndef，#define の 4 種の命令の条件に使用されるマクロ変数名と，開始・終了行番号の情報を取得する．ただし，ヘッダファイルの多重インクルードを防ぐために用いられるインクルードガードは除外する．

3.3 可変点コードへの変更の特定

変更差分解析では，そのコミットにおいて変更された箇所が前処理命令の範囲内であるかどうかを特定する．3.2 節で述べた方法によって取得した前処理命令の出現位置の情報と，変更された箇所の行番号を比較することで，前処理命令の範囲内に変更があったかどうかを特定する．

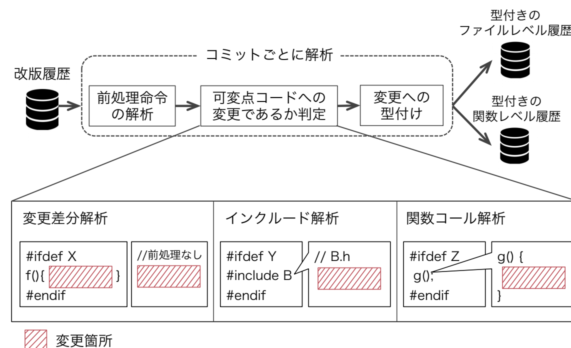

図 1　手法全体の概略図

変更された箇所が直接前処理命令に囲まれていなくても，変更されたヘッダファイルのインクルード文や，変更された関数の呼び出しが前処理命令に囲まれている場合が存在する．このようなケースに対しては，インクルード解析と関数コール解析を行い，可変点コードに対する変更であることを特定する．

インクルード解析では，そのコミットにおいて変更されたヘッダファイルが前処理命令の範囲内でのみインクルードされているかを特定する．この解析ではその変更がなされたコミットにおけるプロジェクト全体のスナップショットに対して，srcML を適用する．プロジェクト全体に対して srcML を適用する理由は，インクルード解析の対象となっているヘッダファイルがプロジェクト下のどのファイルの何行目でインクルードされているのかを網羅的に探索する必要があるためである．Algorithm 1 により，プロジェクト内において前処理命令の範囲内でのみインクルードされるヘッダファイルの一覧を取得する．本稿では，Algorithm 1 における $LIMIT$ は 10 に設定した．この中に変更されたヘッダファイルが含まれていれば，可変点コードに対する変更と判定する．

Algorithm 1 の補助関数は以下の通りである．

$\mathrm{inPP}(c, v)$: ヘッダファイル c がファイル v 内において前処理命令の範囲内でインクルードされていれば真となる述語．

$inc(c)$: ファイル c がインクルードするヘッダファイルの集合．

$inc^{-1}(c) = \{f \mid c \in inc(f) \land f$ はヘッダファイル $\}$: ファイル c をインクルードするヘッダファイルの集合．

関数コール解析ではインクルード解析と同様の方法により，そのコミットにおいて変更された関数が前処理命令の範囲内でのみ呼び出されるかを特定する．変更がなされたコミットにおけるプロジェクト全体のスナップショットに対して srcML を適用，生成された XML を解析し，前処理命令の範囲内でのみ呼び出される関数の一覧を取得する．この中に変更された関数が含まれていれば，可変点コードに対する変更と判定する．

3.4 変更に対する型付け

3.3 節にて説明した 3 種類の解析によって可変点コードに対する変更だと判定された場合には，その変更に対して前処理命令の条件に使用されるマクロ変数名を型として付与する．前処

Algorithm 1 前処理命令の範囲内でのみインクルードされるヘッダファイルの取得

Input: $inc : F \rightarrow 2^F$ ファイル f から f がインクルードするファイル集合への写像

Output: $result$: ヘッダファイルの集合

1: $result \leftarrow \emptyset, count \leftarrow 0$
2: $candidates \leftarrow \{f \mid inc^{-1}(f) \neq \emptyset\}$
3: **while** $candidates \neq \emptyset \wedge count < LIMIT$ **do**
4: $\quad nextCandidates \leftarrow \emptyset$
5: $\quad$ **for each** c **in** $candidates$ **do**
6: $\quad\quad$ **if** $\forall v \in inc^{-1}(c)\ .\ (\mathrm{inPP}(c, v) \vee v \in result)$ **then**
7: $\quad\quad\quad result \leftarrow result \cup \{c\}$
8: $\quad\quad\quad nextCandidates \leftarrow nextCandidates \cup inc(c)$
9: $\quad candidates \leftarrow nextCandidates$
10: $\quad count \leftarrow count + 1$

理命令に関係のある変更であると判定されなかった場合は，可変点コードに対する変更でないことを表す $*$ を型として付与する．ファイルレベルではファイルの変更に対して型を付与し，関数レベルでは関数の変更に対して型を付与する．前処理命令が入れ子構造になっている場合や，前処理条件に複数のマクロ変数が含まれる場合には，各マクロ変数を個別に型として付与する．例えば，ファイル Foo.c の関数 func において可変点コードに変更があり，関係する前処理命令が#if X && Y であった場合には，ファイルレベルでは Foo.c[X]，Foo.c[Y] のように型を付与し，関数レベルでは Foo.c#func[X]，Foo.c#func[Y] のように型を付与する．本手法における型付けは，前処理条件の真偽には着目せず，マクロ変数名のみを用いる．そのため，否定演算子の有無や#if ブロックと#else ブロックの違いは区別しない．

4. 実 験

評価実験により以下の Research Question に回答する．

RQ1 可変点コードに対する変更は改版履歴中でどの程度起こっているか

RQ2 インクルード解析と関数コール解析に効果はあるか

RQ3 型付きの改版履歴から得られる共変更ルールにはどのような特徴があるか

RQ4 型付き共変更ルールの利用により推薦性能は向上するか

表 1 に示す 3 つの OSS を実験対象とする．表 1 における C/H ファイル数と総行数は最新リビジョンのものである．

表 1 対象プロジェクト

対象プロジェクト	総コミット数	開発期間	C/H ファイル数	総行数 (KLOC)
OpenLDAP	22,058	1998–2016	723	416
OpenSSL	17,370	1998–2016	1,029	362
OpenSSH	8,515	1999–2016	379	130

4.1 RQ1: 改版履歴中での可変点コードに対する変更割合

表 1 で示した 3 つの OSS の改版履歴に手法を適用し，ファイルレベルと関数レベルの型付き改版履歴を生成した．表 2 に，ファイルレベル，関数レベルそれぞれの型付き改版履歴における型付きの変更を含むコミットの割合を示す．ファイルレベルでは，C のソースファイルまたはヘッダファイルに対して変更のあったコミットの内，OpenLDAP では 53.2%，OpenSSL では 63.2%，OpenSSH では 47.5%のコミットで可変点コードに対する変更を含んでいた．関数レベルでは，関数に対して変更のあったコミットの内，OpenLDAP では 47.9%，OpenSSL では 53.4%，OpenSSH では 39.5%のコミットで可変点コードに対する変更を含んでいた．

表 2 型付きの変更を含むコミットの割合

	ファイルレベル		関数レベル	
プロジェクト	#コミット	#型付き	#コミット	#型付き
OpenLDAP	16,735	8,906 (53.2%)	14,649	7,015 (47.9%)
OpenSSL	10,661	6,740 (63.2%)	8,178	4,364 (53.4%)
OpenSSH	5,742	2,728 (47.5%)	4,488	1,773 (39.5%)

表 3 最新リビジョンにおけるヘッダファイルの数

プロジェクト	ヘッダファイルの総数	前処理命令内でのみインクルードされるヘッダファイル数	前処理命令内で一度でもインクルードされるヘッダファイル数
OpenLDAP	168	13	59
OpenSSL	189	14	103
OpenSSH	119	7	72

RQ1 への回答：CS の改版履歴では約 40%から 60%のコミットに可変点コードに対する変更が含まれており，実際に可変点コードに対する変更が多く行われている．

4.2 RQ2: インクルード解析と関数コール解析の効果

インクルード解析と関数コール解析を行う場合と行わない場合を比較し，型付きの変更がどれだけ増えたか調査した．ファイルレベルの改版履歴においては，インクルード解析と関数コール解析を行うことで，型付きの変更が OpenLDAP で 4.6%，OpenSSL で 6.7%，OpenSSH で 1.9%増加した．関数レベルでは関数コール解析を行うことで，型付きの変更が OpenLDAP で 3.1%，OpenSSL で 10.1%，OpenSSH で 2.8%増加した．OpenSSL の関数レベルの型付き改版履歴においては 1 割以上型付きの変更を増やすことができたため，関数コール解析に確かな効果があったと考える．その他の場合においては，型付きの変更数が大幅に上昇することはなかった．

関数レベルの OpenSSL 以外の場合に型付きの変更数がさほど増えなかった原因を考察する．表 3 は，各プロジェクトの最新リビジョンにおけるヘッダファイルの総数と，前処理命令の範囲内でのみインクルードされるヘッダファイルの数，前処理命令の範囲内で一度でもインクルードされるヘッダファイルの数を示している．全プロジェクトにおいて，前処理命令の範囲内でのみインクルードされるヘッダファイルが少数であることが分かる．本手法のインクルード解析では，前処理命令の範囲内でのみインクルードされるヘッダファイルへの変更に対して型付けを行う．この型付けの条件を厳密にしすぎたことが，型付きの変更数がさほど増えなかった原因であると考える．

表 4 は，各プロジェクトの最新リビジョンにおける関数の総数と，前処理命令の範囲内でのみ呼ばれる関数の数を示している．ヘッダファイルの場合と比較すると，前処理命令内でのみ呼ばれる関数が少ないわけではないことがわかる．しかしなが

表4　最新リビジョンにおける関数の数

プロジェクト	関数の総数	前処理命令内でのみ呼ばれる関数の数	左記の関数の内，定義全体が前処理命令に囲まれるものの数
OpenLDAP	5,706	950	807 (84.9%)
OpenSSL	7,137	1,612	897 (55.6%)
OpenSSH	2,599	429	390 (90.9%)

表5　共変更ルールの内訳

	ファイル		関数	
プロジェクト	型なし	型付き	型なし	型付き
OpenLDAP	33,507 (70.1%)	14,269 (29.9%)	129,483 (71.5%)	51,528 (28.5%)
OpenSSL	39,939 (33.1%)	80,637 (66.9%)	49,816 (42.4%)	67703 (57.6%)
OpenSSH	140,292 (98.2%)	2,530 (1.8%)	12,506 (60.3%)	8236 (39.7%)

ら，前処理命令の範囲内でのみ呼ばれる関数の56%から91%が，前処理命令の範囲内で定義されている．このことから，関数コール解析を行わなくても変更差分解析のみで変更に型がつく場合が多かったことが，型付きの変更数がさほど増えなかった原因であると考える．

RQ2 への回答：プロジェクトや変更の粒度によって効果の大きさに違いはあるが，変更された箇所が前処理命令の範囲内であるかのみを解析する場合より，2%〜10%程度可変点コードに対する変更であると判定できるケースが増えた．

4.3 RQ3: 型付きの改版履歴から得られる共変更ルールにはどのような特徴があるか

4.3.1 実験設定

RQ3 では，型付きの改版履歴から得られる共変更ルールの特徴を明らかにする．共変更ルールの中に可変点コードに対する変更を要素に持つルールがどの程度含まれているかという数の観点と，得られたルールの質の観点から考察を行う．数の観点においては，得られたルールの中から型のついていない変更を要素に持つルールを除外し，∗ 以外の型が付与された変更を要素に持つルール数の内訳を調査する．質の観点においては，一般的に相関ルールの質を表す尺度として用いられる確信度とリフト値を調査する．得られたルールの内，右辺の変更と同じファイルもしくは関数を左辺にもつルールは除外した．

共変更ルールの生成には，相関ルールマイニングのアルゴリズムの1つである Apriori アルゴリズムを用いた．Apriori アルゴリズムでは，最小支持度 (*minsup*) と最小確信度 (*minconf*) の2パラメータの設定が必要である．本節の実験では，全改版履歴中の全コミットをマイニングの対象とし，7回以上同時変更されている組み合わせからルールを生成するように *minsup* を定めた．*minconf* の設定は 0.1 とした．また，同時変更数の多いコミットはノイズとなる共変更ルールが生成される原因となるため [12]，影響が大きいと考えるファイルレベルの履歴をマイニングする際は，同時変更数が 100 以上のコミットを取り除いた．

4.3.2 実験結果

表5に，3つのプロジェクトの型付き改版履歴から得られた，ファイルレベルと関数レベルのルール数の内訳をそれぞれ示す．OpenSSH のファイルレベル以外では，得られたルールのうち 30%〜70%程度が可変点コードに対する変更を要素に持つ共変更ルールであった．プロジェクトや改版履歴の粒度によって差はあるが，可変点コードへの変更を加味した共変更ルールを多く得ることができた．対して，OpenSSH のファイルレベルでは，可変点コードに対する変更を要素に持つものはわずか 1.8%であった．これは，∗ を型にもつルールの数が非常に多かったことが原因であった．今回の実験では，ノイズとなるルールの生成を防ぐためファイルレベルにおいて同時変更数 100 以上のコミットをマイニングの対象から除いたが，OpenSSH では同時変更数が 100 以下であるコミットの中にも取り除くべきコミットが見受けられた．OpenLDAP と OpenSSL に比べて OpenSSH はプロジェクトの規模が小さいため，このようなコミットを取り除けず，∗ を型に持つルールが多く生成されたものと考える．

図2に，各プロジェクトのファイルレベルにおいて得られた共変更ルールの確信度とリフト値の関係を示す．ルールを 0.25 刻みの確信度で分類し，それぞれのリフト値の分布を箱ひげ図で表したグラフである．OpenLDAP と OpenSSL では，どの確信度の分類においても型付きの共変更ルールの方がリフト値が高いことが分かる．また，型付きのルールでは確信度が高いルールの中にリフト値が非常に高いものが多く含まれていることが分かる．この傾向は，関数レベルの共変更ルールにおいても同様であった．共変更ルールにおけるリフト値は，ルールの右辺が表す変更が起こりにくい程高くなる．起こりにくい変更は開発者が変更を忘れる可能性が高く，リフト値の高い共変更ルールを推薦に使用することは，開発者が変更し忘れる可能性の高いファイルや関数の推薦に大きく貢献することが期待できる．この観点から，型付きの共変更ルールは変更推薦において有用であると考える．

OpenSSH のファイルレベルでは確信度が 0.5 以上の場合において，型付きルールのリフト値の中央値は元の改版履歴から得られたルールに比べて低くなった．これは，表5に示す OpenSSH のファイルレベルにおける型付きルール 2530 個の内，右辺と左辺に異なるファイルをもつものが 191 個と非常に少なかったためだと考える．

RQ3 への回答：提案手法により抽出された共変更ルールには可変点コードに対する変更を要素に持つものが多い．また，確信度及びリフト値の観点から質の高いものが多い．

4.4 RQ4: 型付きの共変更ルール利用による推薦性能の向上

4.4.1 実験設定

この実験では，型付き改版履歴から得られた共変更ルールを用いた2種類の推薦手法を既存手法と比較し，変更推薦の性能を向上できるかを明らかにする．提案手法1では，型付きの改版履歴から抽出された共変更ルールの内，ルールの全ての要素に型がついているルールのみを推薦に使用する．提案手法2では，提案手法1で使用するルールに加え，右辺の要素に型がついていない $A[X] \Rightarrow B$ のようなルールも推薦に使用する．既存手法では，元の改版履歴から得られる型なしの共変更ルールを用いて推薦を行う．

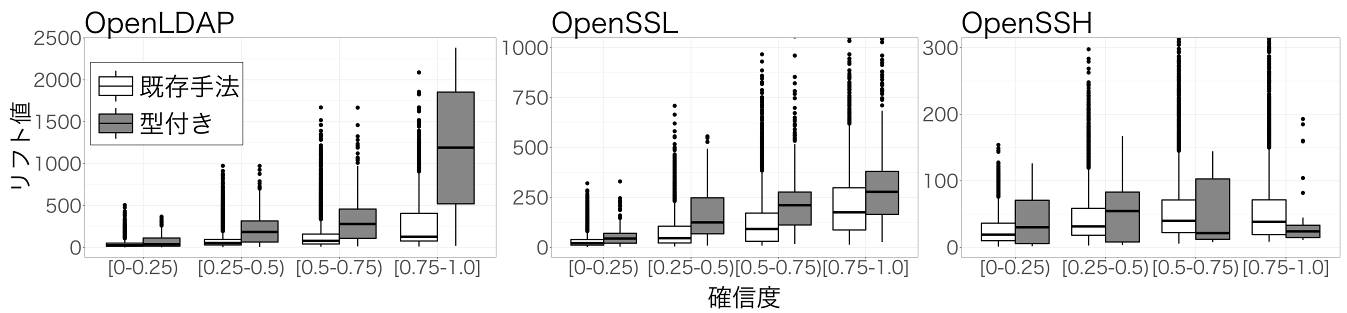

図 2　ファイルレベルにおける共変更ルールの確信度とリフト値の関係

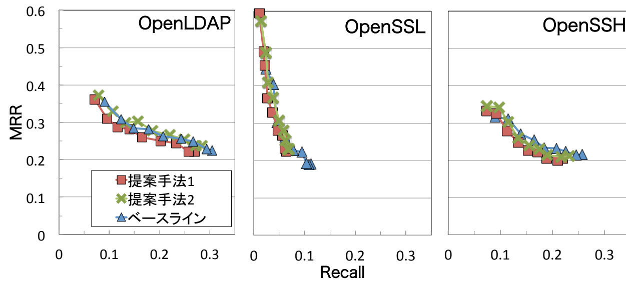

図 3　関数レベルの MRR-Recall グラフ

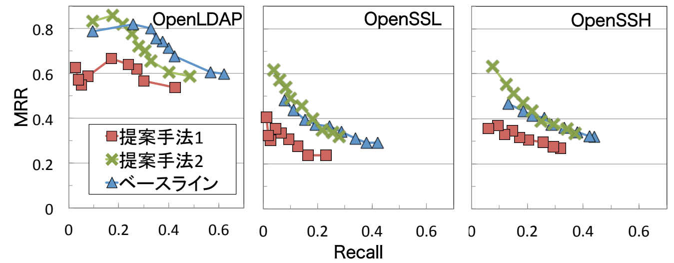

図 4　ファイルレベルの MRR-Recall グラフ

推薦性能を評価するため，先行研究 [4,6] と同様にコミット時の変更漏れに対する推薦という実験設定を用いる．推薦フェーズでは，leave-one-out 交差検証によって評価対象のコミットを評価する．コミット内のファイルや関数を 1 つ欠損させることによって，そのファイルや関数を変更し忘れたという状況を仮想的に作り出す．提案手法では，左辺の集合の要素が型の情報まで含めて一致した場合のみ，右辺を推薦候補とする．また，推薦の正解を既存手法と合わせるため，推薦に使用するルールの右辺に型がついていても，型の種類に関係なく正解と同じファイルや関数を推薦することができれば，正解とする．評価尺度も，[4,6] と同様に *MRR*，*Recall* を用いた．*MRR* は推薦の精度を表す尺度であり，*Recall* は変更漏れに対して推薦を正解することのできた網羅率を表す．

ファイルレベルの改版履歴を用いた実験では，共変更ルールの生成時のパラメータとして，*minsup* を OpenLDAP で 0.0004，OpenSSL で 0.0006，OpenSSH で 0.001 に設定した．関数レベルの改版履歴を用いた実験では，*minsup* を OpenLDAP で 0.0004，OpenSSL で 0.001，OpenSSH で 0.0007 に設定した．*minconf* については，ファイルレベルの改版履歴，関数レベルの改版履歴ともに，全てのプロジェクトで 0.1 から 0.9 まで 0.1 刻みで変化させて実験を行った．評価対象として扱うコミットには，ファイルレベル，関数レベルともに，全てのプロジェクトで同時変更数が 2 以上である最新のコミット 500 個を用いた．

4.4.2　結　　果

図 3 に関数レベルの改版履歴を用いた場合の推薦結果を示す．縦軸が *MRR*，横軸が *Recall* を表している．グラフの各プロットは *minconf* が 0.1〜0.9 のときの結果を示している．全プロジェクトで，2 つの提案手法による推薦の最大 *MRR* がともにベースラインを上回ったことが分かる．OpenSSL では特に差が大きく，*minconf*=0.9 の時の *MRR* はベースラインの 0.446 に対し，提案手法 1 が 0.595 と約 1.3 倍になった．*Recall* の観点では，全プロジェクトで提案手法における推薦の最大 *Recall* がベースラインを下回った．

型付きの共変更ルールの使用が，正解を上位に推薦することに大きく貢献した実例として，OpenSSH のハッシュ値 63ddc89 のコミットを対象とした推薦をあげる．gss-serv-krb5.c の関数 ssh_gssapi_krb5_storecreds を変更漏れとして扱った場合に，ベースラインでは正解を 3 位で推薦していたのに対し，提案手法では 1 位で推薦できた．提案手法において正解の推薦に貢献した共変更ルールは以下のルールである．

正解の推薦に貢献したルール

auth-krb5.c#auth_krb5_password[HEIMDAL]
$\Rightarrow$ gss-serv-krb5.c#ssh_gssapi_krb5_storecreds[HEIMDAL]

このコミットでは実際に auth-krb5.c の関数 auth_krb5_password と gss-serv-krb5.c の関数 ssh_gssapi_krb5_storecreds において，HEIMDAL というマクロ変数を条件部にもつ前処理命令の範囲内が変更されていた．ベースラインでは auth-krb5.c の関数 auth_krb5 と auth-krb5.c の関数 auth_krb5_tgt が 1 位，2 位に推薦されていた．これらはルールの左辺の関数と同じファイル内の関数であり，妥当な推薦だと言える．一方，提案手法では可変点コードに対する変更を考慮することで，真に必要な変更を 1 位で推薦できた．

図 4 にファイルレベルの改版履歴を用いた場合の推薦結果を示す．提案手法 2 による推薦では全てのプロジェクトに共通して，最大 *MRR* がベースラインを上回った．一方，提案手法 1 による推薦は全プロジェクトにおいて，*MRR*，*Recall* 双方の評価尺度でベースラインを下回る結果となった．

提案手法 1 がベースラインの結果を下回ってしまった原因を考察する．実際の推薦結果を確認したところ，既存手法で正解の推薦に貢献していた共変更ルールが，型付きの改版履歴をマイニングした場合には抽出することができないケースが多数確認できた．これは，変更に型をつけることで変更内容が細分化され，共変更ルールの抽出時に *minsup* や *minconf* を超えなかったことが原因である．一方，提案手法 2 では右辺に型のつかないルールも使用するため，提案手法 1 で推薦が行えなかったケースに対して推薦可能な機会が増加する．ルールの左辺の型情報まで一致している場合に推薦が行われるため，既存手法で有益であったルールを更に厳密な条件で推薦に利用したことが，最大 *MRR* が向上した原因であると考える．

今回の実験設定では，過去に実際に行われた変更を 1 つ欠損させ，仮想的な変更漏れとして扱っているため，頻繁に変更されるファイルや関数ほど推薦を成功させやすい．頻度の高い変更を右辺にもつ共変更ルールはリフト値が低くなるため，この実験ではリフト値の低いルールほど正解の推薦に貢献しやすくなる．一方で，リフト値の高いルールは質の観点では有益であるが，右辺の変更の頻度が低いためこの実験では正解の推薦に貢献しない場合が多い．しかし，実際に変更推薦を行う状況を想定すると，頻繁に変更されるファイルや関数を開発者が変更し忘れる可能性は低く，逆に稀にしか変更しないファイルや関数ほど開発者が変更し忘れることが想定され，リフト値の高いルールは変更漏れに対する推薦において有効に働くと考える．開発者の見落としやすさを表す尺度を定義するか，実際に発生した変更漏れから正解セットを作成して評価実験に用いることで，リフト値の高い型付き共変更ルールが変更漏れに対する推薦に有効に働くことを検証する必要があると考える．

RQ4 への回答：*MRR* の最大値が向上し，可変点コードに対する変更を加味した共変更ルールが変更推薦に有効であることを示せた．

4.5 妥当性への脅威

RQ4 に対応する実験の際には，共変更ルールの抽出時に必要なパラメータである *minsup* を，4.4.1 節で説明した値に設定した．今回行った実験では，得られる共変更ルールの数が少ないと変更漏れに対して推薦がほとんど発生せず，推薦性能の適切な比較が行えない．現在のところ適切な *minsup* の値を一意に定める手法がないため，*minsup* の決定方法が一意ではないが，個々のプロジェクトにおいて手法を比較する際の *minsup* を統一しているため，各プロジェクトごとの *minsup* が異なっていても実験結果の妥当性に影響を与えないと考える．また，本稿では実験の対象として 3 つの異なるプロジェクトを用いた．前処理命令をソースコード中に多く含む OSS プロジェクトという基準で実験対象を選定したが，どれも通信プロトコルの実装であり，評価対象の選定に偏りがあったと考える．今後，他ドメインのプロジェクトを対象とした実験を行い，同様の結果が得られるか更なる調査が必要である．

5. おわりに

CS に対して改版履歴に基づく変更支援手法を適用する場合には，CS の重要な要素である前処理命令の情報を加味できていないという問題が存在する．本稿では，可変点コードに対する変更の情報を加味した共変更ルールを抽出する手法を提案した．手法を 3 つの OSS に適用し，確信度及びリフト値の観点から質の高い有益な共変更ルールが得られることを確認した．推薦性能の評価実験では，可変点コードに対する変更を考慮した共変更ルールが変更推薦の性能向上に寄与することを示した．

今後の課題として，前処理命令の用途の考慮が挙げられる．関数の定義内で前処理命令が用いられている場合はその関数の振る舞いを変更することが目的である．一方，関数呼び出しが前処理命令の範囲内にある場合は，その関数の呼び出しそのものが前処理命令によって制御されることを意味する．このような違いも考慮することで，より正確な情報を扱えると考える．また，初期分析の結果 Samba，GCC，Linux などの OSS は前処理命令を多く含むことが明らかになっている．これらの規模の大きいプロジェクトに手法を適用する際に，現在の実装では非常に時間がかかるという問題があるため，スケーラビリティの向上が必要である．

謝辞　本研究の一部は JSPS 科研費 (#24300006, #26280021, #15H02683，#15K15970，#15K15973) の助成を受けた．

文　献

[1] J. Van Gurp, J. Bosch, and M. Svahnberg, “On the notion of variability in software product lines,” Proc. WICSA, pp.45–54, 2001.

[2] R. Srikant, Q. Vu, and R. Agrawal, “Mining association rules with item constraints,” Proc. SIGKDD, pp.67–73, 1997.

[3] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, “Mining version histories to guide software changes,” IEEE TSE, vol.31, no.6, pp.429–445, 2005.

[4] T. Mori, A. Hagward, and T. Kobayashi, “Effects of recency and commits aggregation on change guide method based on change history analysis,” Proc. ICSEA, pp.96–101, 2015.

[5] T. Rolfsnes, S.D. Alesio, R. Behjati, L. Moonen, and D.W. Binkley, “Generalizing the analysis of evolutionary coupling for software change impact analysis,” Proc. SANER, pp.201–212, 2016.

[6] 森達也，A. Hagward，小林隆志，“改版履歴の分析に基づく変更支援手法における時間的近接性の考慮と同一作業コミットの統合による影響，” 情報処理学会論文誌，vol.58，no.4，2017.

[7] N. Dintzner, A. van Deursen, and M. Pinzger, “FEVER: extracting feature-oriented changes from commits,” Proc. MSR, pp.85–96, 2016.

[8] 今西洋二，渥美紀寿，森崎修司，山本修一郎，阿草清滋，“前処理命令の制御構造とその構造内のコード改変に関する調査，” 情報処理学会研究報告，vol.2016，no.6，pp.1–8，2016.

[9] L. Braz, R. Gheyi, M. Mongiovi, M. Ribeiro, F. Medeiros, and L. Teixeira, “A change-centric approach to compile configurable systems with #ifdefs,” Proc. GPCE, pp.109–119, 2016.

[10] M.L. Collard, M.J. Decker, and J.I. Maletic, “srcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration,” Proc. ICSM, pp.516–519, 2013.

[11] J.I. Maletic and M.L. Collard, “Exploration, analysis, and manipulation of source code using srcML,” Proc. ICSE, pp.951–952, 2015.

[12] A. Hindle, D.M. German, and R. Holt, “What do large commits tell us?: a taxonomical study of large commits,” Proc. MSR, pp.99–108, 2008.