Vol.2020-SE-204 No.4
2020/3/2

探索に基づくローカルリファクタリングの検出

筒井 湧暉[1,a)]　セーリム ナッタウット[1,b)]　林 晋平[1,c)]　佐伯 元司[1,d)]

概要：版間で実施されたリファクタリングを自動で検出する既存の手法は，関数の内部に閉じて行われるローカルリファクタリングの大部分を検出できない．本論文では，まず 123 のオープンソースソフトウェアリポジトリからローカルリファクタリングの実例を抽出し，ローカルリファクタリングを構成する原始的な操作を特定した．また，版間のローカルリファクタリングの検出を原始的な操作列を探索する問題と捉え，版間の差分を最小化する操作列を導出する手法を提案する．探索空間中の状態には，ソースコードの構造を表した抽象構文木を用いる．評価関数には，探索中の操作列適用後の版と最終状態の版との間の編集距離を推定して用いる．具体的な探索手法として A 探索とビームサーチを用いて提案手法の評価を行った．

1. はじめに

版間で実施されたリファクタリングの自動検出に関する様々な研究が行われている [1–5]．リファクタリング [6] は開発チームによって頻繁に適用される [7] ため，版間に実施されたリファクタリングの内容は，ソフトウェアの進化を理解するための貴重な情報となる．また，検出によって得られるリファクタリングの事例は様々な実証的研究に役立つ [7, 8]．

リファクタリングはメソッドの内部に閉じて実施されることがある．本論文では，これをローカルリファクタリングと呼ぶ．ローカルリファクタリングも開発者によるリファクタリング活動の中で重要な役割を担っていると考える一方で，既存手法は，ローカルリファクタリングの検出に十分に注目していない．例えば，著名な検出器のひとつである RefactoringMiner [2] を用いても，その大部分を検出できない．

本研究は，2 つの目的をもつ．1 つは，ローカルリファクタリングの種類や特徴を明らかにし，その自動検出の必要性を明らかにすることである．もう 1 つは，ローカルリファクタリングの自動検出を可能とすることである．

本論文の主要な貢献を以下に示す．

- 123 の OSS リポジトリから，ローカルリファクタリングの実施事例を調査し，その特徴をまとめるとともに，ローカルリファクタリングの種類を定義しカタログ化した（3 章）．
- 探索に基づいたローカルリファクタリングの自動検出手法を提案，実装した（4 章）．
- 収集したローカルリファクタリングの事例を用いて，提案手法を評価した（5 章）．

本論文の構成を以下に示す．2 章では，ローカルリファクタリングの定義と例を示す．3 章では，ローカルリファクタリング事例の調査について述べる．4 章では，本論文で提案するローカルリファクタリング検出法について述べる．5 章では提案手法を評価する．6 章では関連研究を紹介する．最後に 7 章で本論文をまとめ，今後の課題について述べる．

2. ローカルリファクタリング

本論文では，変更が 1 つのメソッドの内部に閉じているリファクタリングをローカルリファクタリングと定義する．ここで，変更が 1 つのメソッドの内部に閉じているとは，以下の 2 条件が満たされていることをいう．

(1) 変更前後の対応関係が同一メソッド内で完結する．
(2) 変更において対象メソッドの外部の情報（クラス，メソッド，フィールド）を参照しない．

条件 1 を満たさない例に Extract Method [6] がある．このリファクタリングは，変更前のステートメント群を抽出して新しいメソッドを作成するため，2 つ以上のメソッドに変更を行う．また，条件 2 を満たさない例に Change Value to Reference [9] がある．このリファクタリングによる変更は単一のメソッドに閉じるものの，同一のオブジェクトを単一の参照オブジェクトに変換するため，変

[1] 東京工業大学情報理工学院
[a)] tsutsui_y@se.cs.titech.ac.jp
[b)] natthawute@se.cs.titech.ac.jp
[c)] hayashi@c.titech.ac.jp
[d)] saeki@se.cs.titech.ac.jp

図 1　ローカルリファクタリングの例

換対象の他のオブジェクトの情報が必要になる．

図 1 にローカルリファクタリングの例を示す．この版間では，メソッド `calc` の呼出しの実引数として利用されている変数 `val` をインライン化する Inline Variable が実施されている．このリファクタリングは，代入元と代入先ともに 1 つのメソッドの内部で，かつ外部情報を必要としないため，ローカルリファクタリングの条件を満たす．

既存研究でも類似の概念が示されている．たとえば，Weißgerber ら [4] の定義するローカルリファクタリングは，その変更範囲をクラス内部と広くとっているものの，リファクタリングによる変更の局所性について扱っている．また，Soares ら [3] による Low Level リファクタリングは我々の定義に近い．また，厳密な定義はないものの，既存のリファクタリング手法やプログラム変換操作には，ローカルリファクタリングの定義を満たすものを扱っている [10,11]．

3. ローカルリファクタリング事例の調査

3.1 ローカルリファクタリング事例の抽出

我々の知る限り，ローカルリファクタリング事例のデータセットは存在しない．例えば，Tsantalis らが公開しているリファクタリング実施事例のデータセット [2] は検出ツール RefactoringMiner を用いて抽出したものであり，ローカルリファクタリングの事例はほとんど含まれていない．我々は，研究動機をより明確なものとするため，オープンソースソフトウェア（OSS）のリポジトリからローカルリファクタリングの実例を抽出する．様々な種類のローカルリファクタリングを検出可能なツールは存在しないため，我々は，より広い観点でリファクタリングを含む可能性があるコミットを取得し，そのうちローカルリファクタリングの条件を満たしているものを候補として選定した．選定した候補を目視で確認することにより，ローカルリファクタリングの実例を抽出した．

手順を図 2 に示す．まず，123 の OSS リポジトリ[*1]を GitHub から取得し，それらから 1,518,384 コミットを得た．次にこれらに対して下記のフィルタリングを行った．

(1) より粒度の小さいコミットを分析対象として扱うべく，マージコミットでないものに限定した．
(2) Ratzinger らの手法 [12] に倣い，コミットメッセージが語 “refactor” や “restruct” を含むコミットを，リファクタリングに関連するコミットとして抽出した．
(3) 行われた変更が特定のメソッドに閉じているものに限定すべく，変更対象が単一の Java ファイルであり，変更箇所が単一のメソッドに限定されるものに限定した．変更箇所の解析には FinerGit [13] を用いた．

[*1] 2018 年 10–11 月に取得．対象リポジトリは Tsantalis ら [2] が用いたものと同様である．ただし，intellij-community はコミット数が多かったため除外した．

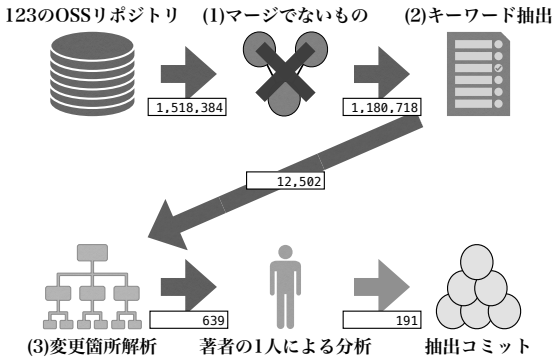

図 2　ローカルリファクタリングに関連するコミットの抽出

これらにより 639 コミットを得た．著者のうち 1 名が，各コミットの変更がローカルリファクタリングを含むかを目視で調べた．リファクタリング実施の判別が困難だったものや，リファクタリングの実施にメソッド外部の情報が必要と判断されたものは除外した．以上により，191 コミットから 487 のローカルリファクタリング事例を得た．

得た事例の分類により，45 種類のローカルリファクタリングパターンを特定した．特定したローカルリファクタリングの一覧を表 1 に示す．例えば，Convert Conditional Statement to Expression は，if 文を用いた代入などの一連の処理を，三項演算子を用いた式に書き換えるリファクタリングで，9 回観測された．これらの全パターンを，リファクタリング名と説明，実例を簡略化して作成した適用例の形でパターンカタログとしてまとめた．

得たローカルリファクタリング実例が，既存のリファクタリング検出器 RefactoringMiner [2][*2]でいくつ検出できるかを調べた．その結果，487 事例のうち，Extract Variable を 20，Rename Variable を 33，Rename Parameter を 7，計 60 事例のみを検出した．すなわち，RefactoringMiner は，抽出したローカルリファクタリングのうち 3 種類，12%（=60/487）しか検出できなかった．

図 3 にコミットが含むローカルリファクタリング数の内訳を示す．対象コミット 191 のうち，ローカルリファクタリングを 1 つだけ含むコミットは 90 あり，それらを除く 101 のコミットでは複数のローカルリファクタリングが観測された．このことは，多くのコミットにおいて，複数の

[*2] 2018 年 10 月時点のものを利用した．https://github.com/tsantalis/RefactoringMiner/commit/efb3040

表 1　ローカルリファクタリング

リファクタリングパターン	観測数
Replace Expression With Variable	106
Extract Variable	66
Move Variable Declaration	55
Rename Variable	46
Inline Variable	35
Reverse Conditional	14
Trim-INVERSE	14
Trim	13
Inline Assignment	11
Import Stop Statement	10
Postpone Assignment	10
Convert Conditional Statement to Expression	9
Introduce Block	9
Rename Parameter	8
Unify Conditionals	7
Aggregate Code Clone	6
Change Assignment To Declaration	6
Swap Operands	6
Convert Conditional to Switch	5
Intersect Conditionals	5
Convert Conditional To Boolean Return	4
Expand Block	4
Introduce Void Stop Statement	3
Move Assignment	3
Change Declaration Description	2
Concat	2
Convert Conditional Expression to Statement	2
Expand Augmented Assignment	2
Export Stop Statement	2
Introduce Try-Catch	2
Introduce Try-Finally	2
Merge Nested Blocks	2
Replace Variable With Expression	2
Restructure Conditional Expression	2
Swap Conditionals	2
Change Declaration To Assignment	1
Contract Augmented Assignment	1
Convert Boolean Return To Conditional	1
Delete Conditional	1
Introduce Redundant Conditional	1
Merge Try-Catch	1
Parallel Conditionals	1
Remove Redundant Conditional	1
Remove Redundant Parentheses	1
Remove Void Stop Statement	1
総数	487

ローカルリファクタリングが混在していることを示唆している．

3.2　ローカルリファクタリングの特徴

リファクタリング以外の変更との混在． Görg らは，他のリファクタリングやリファクタリング以外の変更と混在した変更を impure リファクタリングと呼んでいる [14]．実際に impure なローカルリファクタリングコミットが観測された．図 4 にその例を示す[*3]．このコミットでは，Extract Variable を実施しており，式 `pomMS.getFM().getPP(p)` を変数 `phy` として抽出している．それと同時に，リファクタリング以外の他の変更として変数の null チェックを行う Assert 文が追加されている．

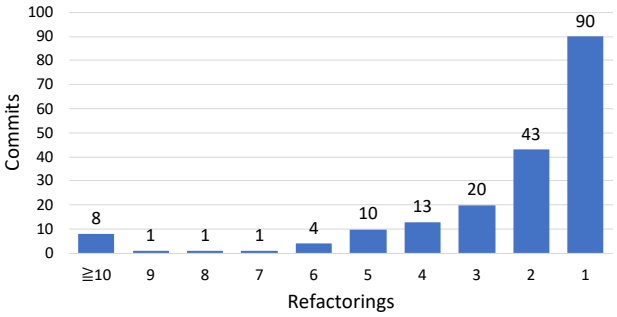

図 3　コミットが含むローカルリファクタリング数

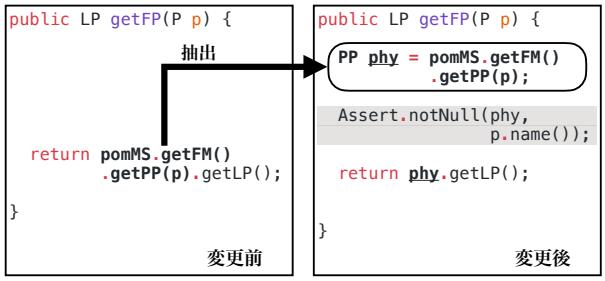

図 4　リファクタリング以外の変更との混在

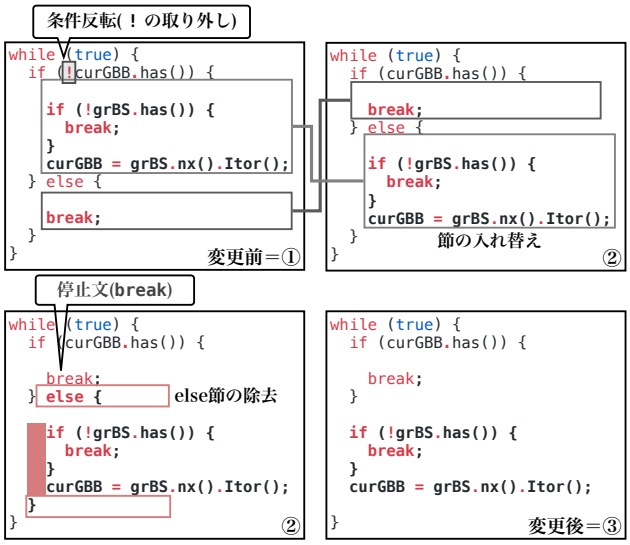

図 5　リファクタリングの連鎖

リファクタリングの連鎖． 図 3 に示した通り，ローカルリファクタリングは他のリファクタリングと混在して

[*3] https://github.com/spring-projects/spring-roo/commit/a24b516．以降も含め，コード中の識別子は適宜短縮している．

いる．特定のリファクタリングを適用しなければ，適用可能にならないようなリファクタリングの連続的な実施が観測された．以降，これをリファクタリングの連鎖と呼ぶ．**図 5** に連鎖の例を示す[*4]．このコミットでは，コード①がコード③に修正されている．中では，2 つの異なるリファクタリングが同じ箇所に連続で実施されている．まず，Reverse Conditional により，条件`!curGBB.has()` が否定されると同時に，then 節と else 節の中身が交換されている（① → ②）．次に，if 節末尾に break ステートメントがあることを利用し，Trim を実施して else 節内の文を節外に移動し，else 節を消去している（② → ③）．

3.3 妥当性の脅威

Soares ら [15] が指摘するように，多くのリファクタリングコミットのメッセージはリファクタリングに関するキーワードを含んでいない．また，我々は Ratzinger らの手法で提案されているキーワードのうち，一部のみを利用している．これらは，多くのリファクタリングに関するコミットを取り漏らしている可能性につながる．また，我々は，メソッドの変更範囲が単一であるコミットのみを抽出している．そのため，複数のメソッドおよび複数のファイルに対して同時に独立したローカルリファクタリングが実施されている場合を取り漏らしてしまう．

4. 提案手法

崔らの調査 [1] によると，連鎖するリファクタリングの検出には，探索に基づく手法が有効であると述べられている．本論文でも，最良優先探索の 1 つである A 探索に基づいてリファクタリングの検出を行う既存手法 [16] を応用する形でローカルリファクタリングの検出を試みる．以降では，まず既存手法に従ったナイーブな手法を紹介した上で，我々が加えた 3 種類の改善について述べる．

4.1 最良優先探索に基づくローカルリファクタリング検出

既存手法の枠組みに従った，ローカルリファクタリング検出のための探索は以下のように定義できる．

- 状態 $s \in S$：メソッド実装を表す抽象構文木（AST）
- 初期状態 $s_0 \in S$：変更前のメソッド
- 目標状態 $s_g \in S$：変更後のメソッド
- 状態展開 $expand(s) \subset S$：s にローカルリファクタリング候補群をそれぞれ適用して得た状態群
- 終了判定 $goal(s) \in \{\mathrm{T}, \mathrm{F}\}$：目標状態 s_g との一致
- 状態評価 $eval(s) \in \mathbb{N}$：s と目標状態 s_g との間の AST 差分に基づく評価を与える

ローカルリファクタリング検出では，変更がメソッドの内部に閉じたリファクタリングを対象に扱うため，既存手法とは異なり，状態を単一のメソッドに限定できる．本手法にも AST の構造に基づく処理が含まれており，状態として AST によるソースコードの中間表現を用いる．

最良優先探索は以下のように実行される．

(1) $s \leftarrow s_0$.
(2) $goal(s)$ が成立すれば，s が得られるまでに適用したリファクタリング操作列を出力して終了する．
(3) 状態展開 $expand(s)$ により得た状態群を優先待ち行列 Q に挿入する．Q の優先度は $eval(s)$ により定める．
(4) Q 先頭の状態へ遷移する：$s \leftarrow pop(Q)$．2 に戻る．

A 探索における状態評価． 既存手法 [16] では最良優先探索として A 探索を利用している．A 探索では，状態評価 $eval(s)$ に，初期状態 s_0 から s までの状態遷移の長さを表す経路コスト $g(s)$ と，s から目標状態 s_g への経路コストを見積もるヒューリスティック距離 $h(s)$ の和 $eval(s) = g(s) + h(s)$ を利用する [17]．

編集スクリプトの計算． 既存の木構造差分検出アルゴリズムを用いることにより，2 つの AST s_1, s_2 の差分を，一方から他方への編集操作列を含んだ編集スクリプト $E = diff(s_1, s_2)$ として得る．編集スクリプト中の編集操作には，挿入 Insert$(target, to)$，削除 Delete$(target)$，移動 Move$(target, to)$，更新 Update$(target, old, new)$ がある．ここで $target$ は操作対象のノード，to は挿入先のノード，old, new は更新前後のノードの値を表す．

編集距離． 本手法では，状態間のヒューリスティック距離を，異なる 2 つの AST 間の編集距離を用いて表現する．編集距離 $d(s_1, s_2)$ は，編集スクリプト中の編集操作のコスト和として計算される．

$$d(s_1, s_2) = size(diff(s_1, s_2)), \quad size(E) = \sum_{e \in E} cost(e)$$

コスト $cost(e)$ は，Insert，Delete では 1.0，Update では 0.5，Move では移動させる部分木のノード数に重み w を乗じたものを用いる．例えば，重み w が 2.0 だった場合，これは部分木全体を Delete したのちに Insert した場合と同等のコストとなる．

リファクタリング候補生成． 状態 s の次状態を生成するためのリファクタリング候補は，s 中で変更された全ノード $n \in N_s$ を n を行きがけ順で走査し，それに適用可能なものをすべて導出する．ノード n が変更されているとは，n あるいは n の子孫のいずれかを対象とするような編集操作 e が，s と s_g との間の編集スクリプト E 内に存在することを指す．ここで，編集操作 e がノード n を対象とするとは，e の種類が Detele，Move，Update のときはその $target$，Insert の場合はその to パラメータが n と一致することを指す．

[*4] https://github.com/prestodb/presto/commit/4a876e6

4.2 改善 1：無更新期間の導入

問題点． 前述した探索では，終了判定に目標状態との一致のみを調べていた．しかし，この判定では，対象とする変更にリファクタリング以外の変更が混在している場合に探索が終了しない．これは，リファクタリングの列ではリファクタリング以外の変更を表現できないため，目標状態に一致する状態を生成できないからである．

解決策． 無更新期間の概念を導入する．無更新期間は，最適状態の更新が観測されない期間を指すもので，最後に目標状態に最も近い状態に遷移したときからの遷移回数で表される．無更新期間 t があらかじめ定めた閾値を超えたとき，評価値の更新が収束したとして探索を終了する．以上を組み込んだ探索手順を以下に示す．

(1) $t \leftarrow 0$; $s_* \leftarrow s_0$; $Q \leftarrow \{s_0\}$.
(2) 優先待ち行列 Q が空になるまで以下を繰り返す．
  (a) 状態遷移：$s \leftarrow pop(Q)$; $t \leftarrow t+1$.
  (b) 最良状態更新：$best(s)$ が真のとき，$s_* \leftarrow s; t \leftarrow 0$.
  (c) 終了判定：$goal(s)$ が真のとき，s_* が得られるまでに適用したリファクタリング列を出力して終了．
  (d) 状態展開：$Q \leftarrow expand(s)$.

ここで，$best(s)$ は s が s_* よりも目標状態 s_g に編集距離が近い状態である場合に真となる．また，終了判定 $goal(s)$ は，目標状態との一致のみならず，探索回数 t が無更新期間に到達する場合にも真とする．

4.3 改善 2：編集距離の見積もりによる差分抽出の省略

問題点． 差分計算は重い処理である．差分は，リファクタリング候補の生成と，目標状態との間の距離の見積もりの際に用いられる．前者は状態遷移時に行われるが，生成された状態の多くは実際には遷移されない．遷移されない状態に対して，後者の見積もり時に差分計算を省略できれば，探索の時間コストを削減できる．

解決策． Hayashi らは編集スクリプトの情報からリファクタリング候補を生成しており，これはリファクタリングの構成要素となり得る編集操作がリファクタリングの適用によって打ち消されるという考えに基づいている [16]．この考えを状態評価に応用し，状態評価におけるリファクタリング r 適用後の状態 s' と目標状態との間の編集距離 $d(s', s_g)$ を，適用前の状態 s における編集スクリプト $E = diff(s, s_g)$ とリファクタリング r の適用によって生じる編集操作 E_r を用いて

$$d'(s', s_g) = d(s, s_g) - size(E \cap E_r) + size(E_r \setminus E \setminus E_M)$$

と見積もる．すなわち，リファクタリングを表す編集スクリプト E_r に相当する編集があれば，編集後に該当する差分は現れないとしてそれらに関するコストを差し引き，そうでなければ，それらに関するコストを追加する．ただし，Move 操作が部分的に一致していた場合，特別な補正を行う．例えば，ノード n が n_1 へ移される Move 操作が差分中に観測され，n を n_2 へ移す Move 操作を行うリファクタリングを考えた場合，リファクタリング適用後の状態と目標状態との間の差分には，もとの差分に含まれていた Move 操作に代わり，n_1 を n_2 へ移す Move 操作が抽出されることが期待される．この場合，リファクタリングが導入する Move 操作は編集距離の増減に影響を与えないため，このような Move 操作 $E_M \subseteq E$ に対してはコストを与えない．

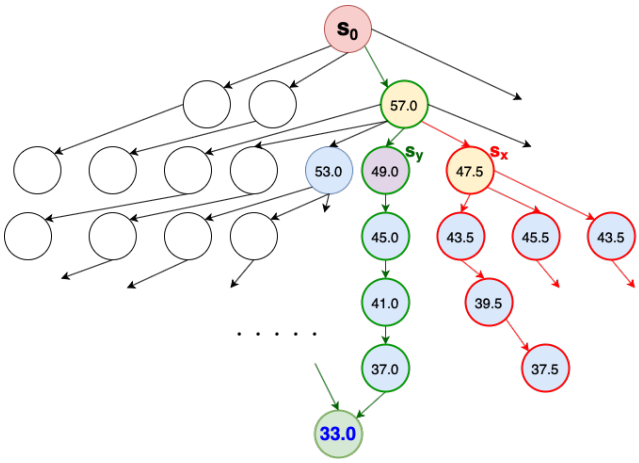

図 6 A 探索における探索木

4.4 改善 3：ビームサーチの導入

問題点． 評価値を部分的に良くするような，誤ったリファクタリング操作が行われた状態が最良状態として選ばれてしまうと，その先の展開，遷移が繰り返され，局所解に陥ってしまう．

局所解への収束例を示す探索木を**図 6** に示す[*5]．正解の解経路は，緑色の枠と矢印で記されている．紫色の状態と黄色の状態を 1 回ずつ，青色の状態を 4 回経由することで，緑色の正解の状態に辿り着く．状態に記された数字は，その時点での編集距離を表している．A 探索を適用した場合，最適ではないものの小さい編集距離（47.5）をもつ状態 s_x へ遷移してしまうと，その下の状態を誤って長時間探索してしまう．正解の解系列では，値が十分に小さくなるまでに遷移を要し，途中の状態が有力なものとして選ばれないためである．

解決策． A 探索に代わりビームサーチの利用を検討する．ビームサーチでは，同じ探索深さにおける評価値の良い上位 k 個に対して，遷移を限定した幅優先探索を行う [18]．この k をビーム幅とよぶ．ビームサーチを用いることにより，評価値で状態空間を圧縮したうえで，局所解への収束の回避を図る．

図 7 に，図 6 に示した例においてビームサーチを適用した場合の探索木を示す．図 6 で示した，最適ではないものの編集距離の小さい状態 s_x だけでなく，やや編集距離が

[*5] https://github.com/siacs/Conversations/commit/aaeba69

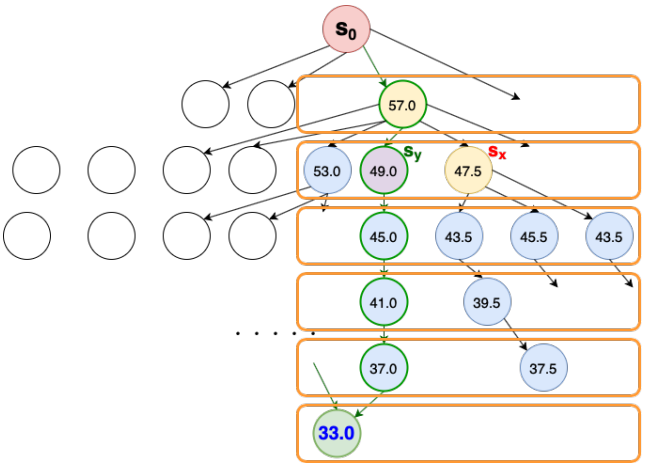

図 7　ビームサーチにおける探索木

大きいものの正解に至る状態 s_y も含めて並列に状態遷移を行うことで，最終的に最適状態に至る解系列を得る．

4.5　実装

Java 言語を対象に検出手法を実装した．構文解析には javaparser[*6]，編集スクリプトの計算には GumTree [19]，探索アルゴリズムの実装には hipster4j [20] を用いた．また本ツールでは 3 章で得た 45 種類のローカルリファクタリングのうち 11 の検出を実装した．これらは，調査において観測数が多かったもの，特徴的な連鎖がみられた例において出現していたものを中心に選ばれた．

5.　評価実験

以下の 3 観点から提案手法を評価した．

- **Q1: A 探索とビームサーチはどれだけの精度でローカルリファクタリングを検出するか?** ローカルリファクタリングの検出がどの程度達成できるかを確認する．同時に，提案手法で検討したビームサーチの効果を，A 探索における結果と比較して確認する．
- **Q2: A 探索とビームサーチはどれだけ早く最良状態に到達するか?** 実利用において探索空間を制限する際の戦略を検討するため，それぞれの探索アルゴリズムがどれだけの探索空間を必要とするかを確認する．
- **Q3: 編集距離の見積もりは，差分計算量をどれだけ削減するか?** 4.2 節で提案した編集距離の見積もりが時間コストをどれだけ削減し得るかを確認する．

5.1　実験設定

3 章で得たリファクタリング事例のうち，検出ツールで候補を生成可能なリファクタリング事例を 1 つ以上含む 85 コミット（182 リファクタリング事例）を用いた．ここで，候補生成不能なリファクタリング操作からの連鎖を要求するリファクタリングは，生成不能とみなした．種類ごとのリファクタリング操作数を**表 2** に示す．評価には，正解と比較しての適合率，再現率，F 値を用いた．

[*6] https://javaparser.org/

表 2　実験に用いた正解ローカルリファクタリング事例

リファクタリングパターン	個数
Replace Expression With Variable	68
Extract Variable	52
Inline Variable	22
Reverse Conditional	9
Trim	9
Import Stop Statement	8
Introduce Block（then 節）	5
Introduce Block（else 節）	2
Intersect Conditionals	3
Introduce Void Stop Statement	2
Expand Block（then 節）	0
Expand Block（else 節）	1
Unify Conditional	1
合計	182

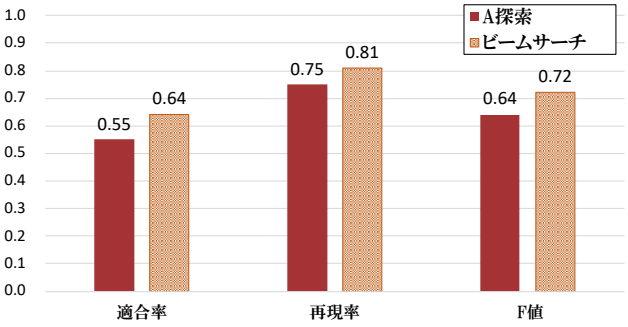

図 8　検出結果（精度）

提案手法のパラメータは，無更新期間として 5000，Move 重み w として 0.5，1 リファクタリングあたりの経路コストとして 5，ビーム幅として 1000（無更新期間の 1/5）を用いた．これらのうち無更新期間以外は，3 章の調査で得たうちの 10 コミットに対して手法を試し，最適なものとして実験的に得た．

本実験では，A 探索とビームサーチ，異なる探索法の比較を行っている．一方の探索結果における状態遷移数がもう一方の状態遷移数よりも小さかった場合，公平性のため，無更新期間に基づく終了判定を行わず，同一の状態遷移数となるまで探索を継続させた．

5.2　Q1：精度

結果を**図 8** に示す．A 探索における適合率を除いて，適合率，再現率，F 値いずれの観点でも 0.60 以上を示した．この結果は，提案手法によりローカルリファクタリングが一定数検出可能であることを示唆している．また，いずれの評価指標においても，A 探索に比べビームサーチが優れた結果を示した．

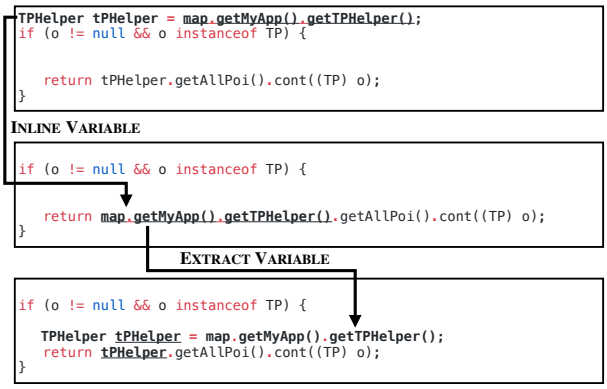

図 9　不正解の操作列が他のリファクタリング操作を表現した例

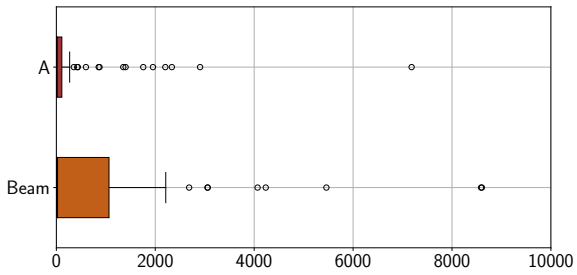

図 10　収束状態までの状態遷移数

低い適合率を示した例を図 9 に示す[*7]．この例では，いずれの手法においても，同一の 3 つのリファクタリング操作を出力した一方，そのうち 1 つのみが正解とみなされた．図 9 は，不正解とみなされた INLINE VARIABLE と EXTRACT VARIABLE の適用の様子を表している．この 2 操作の連鎖は，変数 tPHelper を移動させる効果をもつ．実際に，これらの実施により，変数 tPHelper の宣言に対する Move 操作が表現されており，編集距離は縮んでいる．すなわち，定義した正解では表現されていないリファクタリングの検出がなされたと見なすことができる[*8]．この例を 37 コミットで，同様に正解の解系列により得た状態よりも最良状態が目標状態に近づいた．

5.3　Q2：収束速度

A 探索，ビームサーチのいずれにおいても同一のリファクタリング列が得られた 70 コミットで，最良の状態が何回目の状態遷移で得られたかを計測した．計測結果の状態遷移数の分布を図 10 に示す．図における A は A 探索，Beam はビームサーチの結果を指している．得られた箱ひげ図より，A 探索を用いた方が，より早い段階で出力状態に到達していることが見て取れる．なお，ビームサーチ側の第三四分位数が 1000 付近となったことは，ビーム幅を 1000 に設定したことに起因していると考える．

表 3　検出結果（探索空間）

		A 探索	ビームサーチ
状態生成数	平均	729,135	711,366
	中央	196,319	156,245
	最大	5,114,740	4,474,472
状態遷移数	平均	5,921	5,862
	中央	5,215	5,215
	最大	13,604	13,604

5.4　Q3：差分計算回数

探索の終了までに要した状態生成数および状態遷移数の統計量を表 3 に示す．ここで，実験において状態遷移数の調整を行っているにも関わらずその平均値が異なるのは，目標状態に到達する場合があるためである． 編集距離の見積もりは，状態生成時の差分計算を省略し，状態遷移時にのみ差分計算を行うための改善であるため，生成されるも遷移しない状態数ぶん，差分計算を省略できる．表より，差分計算回数は A 探索において 729,135 から 5,921 に，ビームサーチにおいて 711,366 から 5,862 に減っており，いずれにおいても 99.2% の差分計算が省略できたことがわかる．

5.5　妥当性の脅威

内的妥当性．各データに対して，ビーム幅が余分であったかなどの調査は，本評価ではパラメータに対して結論を導出できなかったことなどから，行えていない．したがって，同じ結果に辿り着く最小のビーム幅を設定できたりした場合は，結論が覆る可能性がある．

外的妥当性．実験に用いたデータはコミットごとにリファクタリングの構成に関する偏りが大きい．他のコミットを用いた場合には，異なる結果を導く可能性がある．

6.　関連研究

版間からリファクタリングを自動検出するべく様々な手法が研究されている．崔らはリファクタリング自動検出技術を大まかに 6 種類に分類した [1]．

Soares らは，ローカルリファクタリングに相当するリファクタリングを Low Level リファクタリングと呼び，その検出に取り組んでいる [3]．しかし，具体的なリファクタリングの実施内容はわからず，リファクタリング以外の変更が混在している場合の検出が不可能という欠点をもつ．

Hayashi ら [16] は，リファクタリング検出を探索問題として定義し，グラフ探索アルゴリズムを適用する検出手法を提案した．この手法では，AST 差分の内容からリファクタリング候補を導出し，各差分に設けた得点から次に遷移する状態を決定する．目標状態に遷移したときに，探索を終了する．

*7　https://github.com/osmandapp/Osmand/commit/4a130c3
*8　これは実際には 3 章で MOVE VARIABLE DECLARATION として特定済のものである．検出ツールでは未実装なため，正解集合にはこの操作を含めていない．

Mahouachi ら [5] は遺伝的アルゴリズムを用いてリファクタリングの検出を行っている．入力からソースコードメトリクスを抽出し，その差分を用いた評価関数が最も良くなるリファクタリング操作の組合せを導出する．グラフ探索アルゴリズムとの違いは，単一のリファクタリング操作をその都度評価するのではなく，複数のリファクタリング操作の組み合わせを用いて評価を行う点にある．

Tsutsumi らは Hayashi らの手法 [16] に動的解析を組み込んだ手法を提案した [21]．この手法では，探索後のコードと変更前のコードとの実行結果の同一性を確認する．同一性が得られれば，探索後コードと変更後コードを比較して得た差分をリファクタリング以外の変更とする．

7. おわりに

本論文では，ローカルリファクタリングの実施例とその内容を明らかにするべく，123 の OSS リポジトリから事例を抽出し調査を行った．その結果，その実施事例に含まれるローカルリファクタリングの原子的な操作列を特定し，検出の必要性を示した．また，特定したローカルリファクタリングの操作列を検出するための，探索に基づいた自動検出手法を提案した．実験により，ローカルリファクタリングを十分な検出精度で検出できることと，A 探索は探索空間面において，ビームサーチは精度面において有利となる傾向があることを示した．

今後の課題を以下に示す．

- **ツールが検出可能なリファクタリング種類の拡充．**現在は，カタログ化した 45 種類のうち 11 種類のみを実装している．
- **状態評価の改善．**一部の不適切な探索結果は，GumTree により得られた編集スクリプトの不適切さや，編集距離見積もりの誤りから生じている．
- **異なる探索方法の検討．**コミットが複数のリファクタリング操作を含んでいても，それらすべてが連鎖しているわけではない．それらを独立に検出したり，特定の局所解の結果の再利用が有効と考える．

謝辞　リファクタリング検出についてご議論頂いた星野友宏氏，梅田侑氏に感謝する．本研究の一部は，日本学術振興会科学研究費補助金（JP18K11238）の助成を受けた．

参考文献

[1] 崔　恩瀞，藤原賢二，吉田則裕，林　晋平：変更履歴解析に基づくリファクタリング検出技術の調査，コンピュータソフトウェア，Vol. 32, No. 1, pp. 47–59 (2015).
[2] Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D. and Dig, D.: Accurate and Efficient Refactoring Detection in Commit History, *Proc. ICSE*, pp. 483–494 (2018).
[3] Soares, G., Catao, B., Varjao, C., Aguiar, S., Gheyi, R. and Massoni, T.: Analyzing Refactorings on Software Repositories, *Proc. SBES*, pp. 164–173 (2011).
[4] Weißgerber, P. and Diehl, S.: Identifying Refactorings from Source-Code Changes, *Proc. ASE*, pp. 231–240 (2006).
[5] Mahouachi, R., Kessentini, M. and Cinnéide, M. Ó.: Search-based Refactoring Detection Using Software Metrics Variation, *Proc. SSBSE*, pp. 126–140 (2013).
[6] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional (2018).
[7] Silva, D., Tsantalis, N. and Valente, M. T.: Why We Refactor? Confessions of GitHub Contributors, *Proc. FSE*, pp. 858–870 (2016).
[8] Mahmoudi, M., Nadi, S. and Tsantalis, N.: Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts, *Proc. SANER*, pp. 151–162 (2019).
[9] Fowler, M.: Refactoring. https://refactoring.com.
[10] Gil, Y. and Orrù, M.: The Spartanizer: Massive Automatic Refactoring, *Proc. SANER*, pp. 477–481 (2017).
[11] Tsai, H.-Y., Huang, Y.-L. and Wagner, D.: A Graph Approach to Quantitative Analysis of Control-flow Obfuscating Transformations, *IEEE Trans. Information Forensics and Security*, Vol. 4, No. 2, pp. 257–267 (2009).
[12] Ratzinger, J., Sigmund, T. and Gall, H. C.: On the Relation of Refactorings and Software Defect Prediction, *Proc. MSR*, pp. 35–38 (2008).
[13] 肥後芳樹，楠本真二：Java メソッドの高精度な追跡のための細粒度版管理システムの提案，信学技報，Vol. 118, No. 384, pp. 133–138 (2019).
[14] Görg, C. and Weißgerber, P.: Detecting and Visualizing Refactorings from Software Archives, *Proc. IWPC*, pp. 205–214 (2005).
[15] Soares, G., Gheyi, R., Murphy-Hill, E. and Johnson, B.: Comparing Approaches to Analyze Refactoring Activity on Software Repositories, *Journal of Systems and Software*, Vol. 86, No. 4, pp. 1006–1022 (2013).
[16] Hayashi, S., Tsuda, Y. and Saeki, M.: Search-Based Refactoring Detection from Source Code Revisions, *IEICE Trans.*, Vol. E93-D, No. 4, pp. 754–762 (2010).
[17] Russell, S.，Norvig, P.，古川康一：エージェントアプローチ人工知能第 2 版，共立出版 (2008).
[18] Reddy, D. R.: *Speech Understanding Systems: A Summary of Results of the Five-Year Research Effort*, Department of Computer Science (1977).
[19] Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M. and Monperrus, M.: Fine-Grained and Accurate Source Code Differencing, *Proc. ASE*, pp. 313–324 (2014).
[20] Rodriguez-Mier, P., Gonzalez-Sieira, A., Mucientes, M., Lama, M. and Bugarin, A.: Hipster: An Open Source Java Library for Heuristic Search, *Proc. CISTI*, pp. 481–486 (2014).
[21] Tsutsumi, S., Choi, E., Yoshida, N. and Inoue, K.: Graph-Based Approach for Detecting Impure Refactoring from Version Commits, *Proc. IWoR*, pp. 13–16 (2016).