コードクローン管理手法に基づく
大規模ソフトウェア保守支援に関する研究

提出先　大阪大学大学院情報科学研究科

提出年月　2025 年 7 月

徳井　翔梧

論文一覧

主要論文

1. 徳井 翔梧, 吉田 則裕, 崔 恩瀞, 井上 克郎, 肥後 芳樹, “コードクローン集約によるファジングの実行効率調査,” コンピュータソフトウェア, Vol.42, No.2, pp.135-141, 2025 年 5 月, （学術論文）
2. 徳井 翔梧, 吉田 則裕, 崔 恩瀞, 井上 克郎, “Cross-Polytope LSH を用いた コードクローン検出のためのパラメータ決定手法,” コンピュータソフトウェア, Vol.38, No.4, pp.60-82, 2021 年 10 月, （学術論文）
3. Shogo Tokui, Norihiro Yoshida, Eunjong Choi, Katsuro Inoue, “Clone notifier: developing and improving the system to notify changes of code clones,” Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER 2020), pp. 642-646, February 2020. (国際学会)

関連論文

1. Hirotaka Honda, Shogo Tokui, Kazuki Yokoi, Eunjong Choi, Norihiro Yoshida, Katsuro Inoue, “CCEvovis: A clone evolution visualization system for software maintenance”, 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), pp.122-125, May 2019

内容梗概

近年，社会におけるソフトウェアの重要性が高まっており，社会基盤を支える大規模なソフトウェアを保守し，長期間にわたって品質を保つことが求められている．ソフトウェア保守において，潜在的な障害を未然に検出・対処することは重要な課題の 1 つである．しかし，限られた時間と工数の中でソフトウェアの潜在的な障害をすべて取り除くことは困難である．そのため，潜在的な障害を未然に防ぐための支援開発技術が必要とされている．

リファクタリングやバグ修正を行う際には，修正箇所に類似するコード片にも同様の問題が残っている可能性が高く，同様に修正を適用する必要がある．既存コードのコピーアンドペーストによる再利用等で発生する，一致または類似した部分を持つコード片であるコードクローンはソフトウェア保守を困難にさせる要因の 1 つであると指摘されており，コードクローンを管理する手法が盛んに研究されている．

コードクローンを保守するために，ソースコード中からコードクローンを識別して管理する必要がある．しかし，ソースコードの規模が大きくなるとソースコード中に含まれるコードクローンも膨大な量となり，手作業でコードクローンを管理することが困難となる．そこで，ソフトウェアの保守作業を効率化させるために，ソースコードから自動的に検出し，コードクローンの存在を把握する手法が研究されている．さらに，ソフトウェアの保守性を向上させるために，コードクローンを 1 つの関数やクラスに集約して除去する手法が研究されている．しかし，一部の処理が変更されたコードクローンや継承関係にあるクラスなど，集約が困難なコードクローンは潜在的な障害としてソースコード中に残存するため，コードクローンを追跡して管理する手法が提案されている．

本論文では，ソフトウェア保守の品質を低下させる要因の 1 つであるコードクローンに着目し，その検出・追跡・集約という管理手法に関する 3 つの研究を実施した．

1. Cross-Polytope LSH を用いたコードクローン検出のためのパラメータ決定手法
2. コードクローン変更管理システムの開発と改善
3. コードクローン集約によるファジングの実行効率調査

1 については，局所性鋭敏型ハッシュ（Locality-Sensitive Hashing，以降 LSH）を

用いたコードクローン検出器のパラメータ値を自動的に決定する手法を提案した．Cross-Polytope LSH は，高速かつ低メモリ消費を実現する LSH アルゴリズムの 1 つである．ただし，検出精度や実行時間に大きな影響を与えるパラメータの数が多く，適切なパラメータ選択にはアルゴリズムに対する深い知見が必要となる問題がある．本手法では，類似度を用いたコードクローン検出が，利用者が与えた再現率の目標値を満たしつつ可能な限り短時間で実行できることを目的として，プロジェクトの規模から適切なパラメータ値を求める線形回帰モデルを構築し，コードクローン検出対象に適した Cross-Polytope LSH に与えるパラメータ値の組を決定する．これにより，CCVolti の利用者は，高速なパラメータを選択することで，大規模なプロジェクトに対して頻繁にコードクローン検出し，修正の即時対応やコードクローンの早期発見，追跡手法への応用を可能とする．

2 については，コードクローンの変更情報を開発者に通知するツールであるコードクローン変更管理システムに，一貫性のない変更を識別する機能を追加した．コードクローンに対する一貫性のない変更は数多く確認されており，その中にはバグ修正の変更が含まれることが指摘されている．既存システムでは，開発者が一貫性のない変更を手作業で識別する必要があり，特に大規模なソフトウェアの検出結果を人手で確認するのは困難である．本手法では，この課題を解決するため，意味的に一致するコードクローン検出器の導入，追跡方法の改善，一貫性のない変更の分類の追加など，4 点の改善を施した．これにより，一貫性のない変更を含むクローンセットの検出と通知を実現し，大規模ソフトウェアの検出結果の確認コストを軽減した．

3 については，テスト対象のソースコード内のコードクローンを集約することによる，ファジングのパス探索の実行効率について調査した．ファジングとは，実行パスに基づいて大量の入力を自動生成し，対象プログラムの潜在的な障害を検出する，ソフトウェア保守における手法の 1 つである．大規模なソフトウェアでは，長時間ファジングを実行してもより深い階層にある未知のパスや潜在的な障害箇所へ到達することが難しい．この課題に対処するため，基本ブロックを含むコードクローンを集約することで，そのコードクローンを含むパスが集約され，未発見のパスに到達しやすくなるという仮説のもと，コードクローン集約前後のプログラムに対する AFL の比較評価を実施した．結果として，コードクローン集約はファジングの挙動を変化させ，大規模ソフトウェアの潜在的な障害の早期発見に寄与する可能性があることを示した．

これらの研究により，大規模ソフトウェア保守における，潜在的な障害の検出を効率よく行うことができる．

謝辞

本研究を行うにあたり，日頃から様々なご指導を賜りました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 肥後芳樹教授に，心から感謝申し上げます．

本論文を執筆するにあたり，様々なご指導ご助言を頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 楠本真二教授，ならびに伊野文彦教授に感謝申し上げます．

本研究を行うにあたり，様々なご指導ご助言を賜りました，立命館大学情報理工学部 井上克郎教授に，心から感謝申し上げます．

本研究を行うにあたり，研究方針など様々なご指導ご助言を頂きました，立命館大学情報理工学部 吉田則裕教授，ならびに京都工芸繊維大学情報工学・人間科学系 崔恩瀞准教授に心から感謝申し上げます．

本研究を行うにあたり，様々なご指導ご助言を頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下誠准教授，ノートルダム清心女子大学情報デザイン学部情報デザイン学科 神田哲也准教授，奈良先端科学技術大学院大学先端科学技術研究科情報科学領域 嶋利一真助教に感謝申し上げます．

日々の研究活動の中で，活発な議論を交わし研究を深める一助となりました，株式会社 NTT データグループ 横井一樹氏，ならびに富士通株式会社 石津卓也氏に感謝します．

また，本研究の実施にあたり，富士通株式会社の皆様には研究の機会や実践の場を与えていただきましたことを感謝いたします．研究についてのご支援やご助言を頂いた，富士通株式会社 徳本晋氏，ならびに中川尊雄氏にこの場で御礼申し上げます．

最後に，日々の研究生活の中でご助言，ご協力を頂いた，大阪大学大学院情報科学研究科コンピュータサイエンス専攻ソフトウェア工学講座の皆様に厚く御礼申し上げます．

目次

図目次

表目次

第 1 章

はじめに

1.1　ソフトウェア保守とその課題

ソフトウェアシステムの社会依存度が年々高まる中，開発後のソフトウェア保守の重要度が高まっている．ソフトウェア保守は，欠陥の修正にとどまらず，環境変化への適応，機能拡張，性能向上，信頼性や安全性の確保といった多様な側面を含み，継続的かつ戦略的な活動である [2–4]．ソフトウェアを長く保守するためには，一定の人的コストが必要であり，特に，障害発生時には追加コストがかかることがあり，緊急対応が遅れた際には甚大なビジネス的損失を発生させるおそれがある．ソフトウェアを低コストで高品質に維持するために，ソフトウェア工学分野ではソフトウェア保守の支援が重要となっている．ソフトウェア保守は以下の 4 つに分類される [5–7]．

是正保守　ソフトウェア製品の引渡し後に発見された問題を訂正するために行う受身の修正

予防保守　引渡し後のソフトウェア製品の潜在的な障害が運用障害になる前に発見し，是正を行うための修正

適応保守　引渡し後，変化した又は変化している環境において，ソフトウェア製品を使用できるように保ち続けるために実施するソフトウェア製品の修正

完全化保守　引渡し後のソフトウェア製品の潜在的な障害が，故障として現れる前に，検出し訂正するための修正

リリース後に発見された欠陥の修正作業にあたる是正保守と予防保守，環境変化に合わせたソフトウェアの改良作業にあたる適応保守と完全化保守に分けられる．是正保守や適応保守は，問題や環境変化が発生した際の対応作業であるが，予防保守や完全化保守は潜在的な障害を未然に防ぐための活動が定義されている．

予防保守や完全化保守において潜在的な障害を検出することは非常に困難である．障害の原因となる要素は，要求・要件の考慮不足，設計不足やミス，ソースコードのバグや記述ミス，ライブラリに含まれる脆弱性の見落とし，ハードウェアの障害，外

部システムとの連携など無数に挙げられる．しかし，時間と工数が制限されている中で，すべての障害の原因となる要素についてテストし修正することは現実的に困難である [8–11]．そこで，ソフトウェア工学分野では，各開発工程の実情の調査研究，潜在的な障害を未然に防ぐための開発者支援ツールについて盛んに研究されている．

1.2 コードクローン管理手法

ソフトウェアの保守を困難にする大きな要因の 1 つとして，コードクローンが指摘されている [12]．コードクローンとは，ソースコード中に存在する互いに一致または類似した部分を持つコード片のことであり，既存コードのコピーアンドペーストによる再利用等が原因で生じる．コードクローンを保守するために，ソースコード中からコードクローンを識別して管理する必要がある．しかし，ソースコードの規模が大きくなるとソースコード中に含まれるコードクローンも膨大な量となり，手作業でコードクローンを管理することが困難となる．

この問題を解決するために，コードクローンをソースコードから自動的に検出する手法 [12–14]，コードクローンをリファクタリングして除去する手法 [15, 16]，コードクローンを追跡して管理する手法 [17] が提案されている．コードクローンに対するリファクタリングでは，コードクローンの集約が実施されるが，一部の処理が変更されたコードクローンやクラスの継承関係にあるクラスなど，リファクタリングできないコードクローンが存在する．そのようなコードクローンは潜在的な障害になる可能性があるため，追跡して管理することが求められる．

1.2.1 コードクローン検出

ソースコードの規模が大きくなるにつれ，手作業でコードクローンを管理することが困難となる．Roy らは，コードクローン間の違いの度合いに基づき，コードクローンを以下の 4 つの定義に分類している [12]．

タイプ 1 空白やタブの有無，コーディングスタイル，コメントの有無などの違いを除き完全に一致するクローン

タイプ 2 タイプ 1 の違いに加えて，変数名などのユーザー定義名，変数の型などが異なるコードクローン

タイプ 3 タイプ 2 の違いに加えて，文の挿入や削除，変更などが行われているコードクローン

タイプ 4 類似した処理を実行するが，構文上の実装が異なるコードクローン

これらのコードクローンをソースコードから自動的に検出するための手法が多く提案されている [13]．コードクローン検出手法は，その手法で用いる検出単位によって，

行単位の検出，字句単位の検出，抽象構文木を用いた検出，プログラム依存グラフを用いた検出，メトリクスなどその他の技術を用いた検出に分類することができる [18].

行単位の検出　行単位の検出では，言語に依存せず検出できるが，タイプ 1 のコードクローンのみ検出可能である．Baker らは，プログラミング言語に依存せず線形時間でコードクローンを検出できる手法を提案した [19].

字句単位の検出　字句単位の検出では，比較的高速に，タイプ 1 からタイプ 2 のコードクローンを検出可能である．神谷らが提案した字句単位の検出手法は，ユーザ定義名を特殊文字に置き換えるという言語依存の処理をするにもかかわらず，C/C++，Java，COBOL など広く用いられている複数のプログラミング言語に対応している [20].

抽象構文木を用いた検出　抽象構文木を用いた検出では，検出の前処理としてソースコードに対して構文解析を行うことで抽象構文木を構築し，抽象構文木上の同形あるいは類似した部分木をコードクローンとして検出する．また，各部分木を特徴ベクトルに変換し，特徴ベクトル間の類似度を求めることによって，ある程度特徴ベクトルに違いがあっても検出可能であり，タイプ 1 からタイプ 3 までのコードクローンを検出できる．横井らは，情報検索技術を利用することにより，コードブロック単位のコードクローンを検出する手法を提案した [21].

プログラム依存グラフを用いた検出　プログラム依存グラフを用いた検出では, プログラムの意味的な処理の類似性に着目しているため, 文の並び替えが発生したコードクローンなど，タイプ 4 のコードクローンを検出可能である．Komondoor らは，ソースコード中の文をプログラム依存グラフのノードとすることで，同一のグラフ構造となるコード片をコードクローンとして検出する手法を提案した [22].

メトリクスなどその他の技術を用いた検出　メトリクスなどその他の技術を用いた検出では，プログラムのモジュールに対してメトリクスを計測し，それらの類似度を計算することによって，タイプ 1 からタイプ 3 のコードクローンを検出できる．Mayrand らは，関数に対して 21 種類のメトリクスを計測することによってコードクローンを検出する手法を提案した [23].

横井らが提案したブロッククローン（コードブロック単位のコードクローン）検出ツール CCVolti [21] は情報検索技術の 1 つであるベクトル化手法 TF-IDF (Term Frequency-Inverse Document Frequency) 法 [24] を利用して，コサイン類似度を計算してコードクローンを検出する．これにより，従来の手法では困難であった意味的に処理が類似したコードクローンを検出可能になり，従来のコードクローン検出手法より精度を向上することに成功した．また，局所性鋭敏型ハッシュ (Locality-Sensitive Hashing，以降 LSH) の 1 つである Cross-Polytope LSH [25] を利用することによって，CCVolti は大規模ソフトウェアに対しても現実的な計算時間でのコードクローン

検出を可能にした．実際，15MLOC の Linux Kernel に対して CCVolti を用いてコードクローン検出した場合，20 分程度で検出が完了し，100MLOC においても 4 時間程度でコードクローンを検出した．

局所性鋭敏型ハッシュ（LSH）

CCVolti [21] や Deckard [26] など一部のコードクローン検出手法は，コード片をベクトル化し，類似度が閾値以上のすべてのベクトル対を求めてコードクローン対を検出する．しかし，すべてのコード片の組の類似度を計算するためには，コード片の数の 2 乗のオーダーの計算量で類似度計算の時間が増加し，大規模ソフトウェアのコードクローン検出では類似度計算に非常に長い時間を要する．例えば，15MLOC の Linux Kernel に対して CCVolti は約 36 万個のベクトルを生成し，ベクトル対の組合せ総数 6.5×10^{10} 回の類似度計算を必要とする．さらに，ベクトルの次元数はソースコード中の語彙数に比例して増加するため，1 回あたりの類似度計算に要する時間やメモリ量も増加し，大規模ソフトウェアのコードクローン検出が現実的に不可能という問題が発生する．そこで，ベクトル化手法を用いたコードクローン検出手法では，LSH を利用して類似度計算の高速化を図っている [27, 28]．

LSH とは，ハッシュを用いた確率的な処理により，高次元ベクトルデータをクラスタリングする手法であり，近似最近傍探索問題を解くアルゴリズムである．近似最近傍探索問題とは，入力ベクトルに対してベクトル集合であるデータセットの中から一定の類似度以上のベクトルを近似的に高速に見つける問題である．LSH は，類似度が高いベクトルが高確率で同じハッシュ値になるように設計されたハッシュ関数を利用し，ハッシュ値が衝突したベクトルを近傍ベクトル候補とする．LSH を用いたコードクローン検出手法では，膨大なベクトル対の類似度を計算する前に，LSH を用いて近傍に存在するベクトル対をフィルタリングする．そして，近傍に存在するベクトル対のみ類似度計算を行うことで，大規模ソフトウェアのコードクローン検出を現実的な計算時間で可能とした．

CCVolti は，ベクトル化手法として TF-IDF 法を用いており，疎な多次元ベクトルを生成しやすいベクトル化手法と相性が良い Cross-Polytope LSH を採用した [25]．Cross-Polytope LSH は，単位球面上に正規化されたベクトルに対してユークリッド距離またはコサイン類似度に基づいた探索精度の有効性が数学的に保証された，高次元空間上における近似最近傍探索を高速かつ高精度に実現するための LSH アルゴリズムである．Cross-Polytope LSH のアルゴリズムにおけるハッシュ関数では，事前に正規化した入力ベクトルに対して，ランダム回転を施し，最も類似度が高い基底ベクトルを識別し，その基底ベクトルに対応するハッシュ値を割り当てる．ベクトルにランダム行列を乗算することにより，類似度が高いベクトル同士が一定の確率で衝突を起こすようになる．

Cross-Polytope LSH ライブラリ FALCONN では，前処理にて次元圧縮をしたり，

ランダム回転の処理に高速アダマール変換を用いたりするなど，メモリ削減や高速化を行っている [29, 30]．それにより，FALCONN は従来の LSH が抱えるメモリ使用量の問題点を改善した．しかし，FALCONN には 10 種類のパラメータが存在し，次元圧縮後の次元数などの検出結果に影響を与えるパラメータと，メモリ上のデータ保持方法などメモリや計算速度に影響を与えるパラメータがある [31]．しかし，これらのパラメータは手法に対する深い知見がなければ適切な値を設定できず，本来の高速性や精度が発揮されないことが課題である．

1.2.2 コードクローンに対するリファクタリング

ソースコードの品質を向上させるために，ソフトウェアのふるまいを保ちつつ内部構造を改善することをリファクタリングと呼ぶ [32]．リファクタリングすべきコードとして，巨大なクラスや長大メソッド，コードクローンなどが挙げられる．コードクローンに対するリファクタリングとして以下のパターンがある [15]．

メソッドの抽出 既存メソッドの一部のコードクローンを新たなメソッドとして抽出する．

クラスの抽出 既存クラスのコードクローンを新たなクラスとして抽出すること．抽出したクラスを親クラスとして抽出することも可能である．ただし，既存クラスがすでに親クラスを所有している場合，複数の親クラスを継承できない言語では親クラスへの抽出はできない．その場合，新規クラスに抽出する機能を既存クラスから委譲するようにリファクタリングを実施する．

メソッドの引き上げ 共通の親クラスを持つ複数の子クラスに含まれる既存メソッドを親クラスに引き上げる．

テンプレートメソッドの形成 詳細な処理が異なるためにメソッドの引き上げができない場合は，親クラスにテンプレートメソッドを形成して共通処理を定義する．

メソッドの移動 メソッドを他のクラスに移動すること．親クラスへの移動ではなく，機能ごと別のクラスに移動する．

メソッドのパラメータ化 メソッド内で定義された変数やリテラルをメソッドの引数に変更する．

コードクローンリファクタリングの研究分野では，自動または半自動でリファクタリングする手法が多く提案されている [16]．Mazinanian らは，コードクローン検出の結果を解析し，リファクタリング可能なコードクローンを特定して自動でリファクタリングする手法 JDeodorant を提案した [33]．Jdeodorant は以下の 7 つの条件のうちすべてを満たすとき，リファクタリング可能と判断する [34]．

- 変数のパラメータ化の際に制御依存，データ依存，反復操作や出力のふるまいに変更がないこと．

- それぞれ異なる子クラス型を持つ変数は，共通の親クラスで宣言されているか，あるいはオーバライドされたメソッド内でのみ呼び出されていること．
- フィールド変数のパラメータ化は，そのフィールド変数が変更不可であるときのみ可能．
- メソッド呼び出しのパラメータ化は，void 型を返さないときのみ可能．
- 抽出されたメソッドが 2 つ以上の返り値を必要としないこと．
- 条件付き return 文がコードクローンのコード片に含まれないこと．
- 分岐処理を意味する命令文（BREAK，CONTINUE）があれば，それに対応する反復命令文がコードクローンのコード片に含まれないこと．

コードクローンを自動でリファクタリングするためには強い制約が必要であり，多くのコードクローンはリファクタリングできずに残存する．さらに，タイプ 3 やタイプ 4 のコードクローンは，詳細な処理内容が異なることが多く，本節で説明したリファクタリング手法を適用することが難しい．そこで，リファクタリングが困難なコードクローンについては，継続的な管理が必要であり，コードクローンへの変更や進化を追跡する手法が研究されている．

1.2.3 コードクローン追跡

ソフトウェア進化に伴い，コピーアンドペーストによるコードクローンの増加や，リファクタリングやバグ修正などのコード変更に起因するコードクローンの変更・削除は頻繁に発生する．しかし，コードクローンが他にも存在することに気付かずに，一部のコードクローンに対してのみ変更を加えた場合，本来同様に変更すべきコード片が潜在的な障害として残存する可能性がある．実際に，コードクローンに対する一貫性のない変更は数多く確認されており，その中にはバグ修正の変更が含まれることが指摘されている [35–37]．このような課題に対処するために，ソフトウェアの進化過程において，コードクローンを追跡し，コードクローンに対する変更の検知および同期を行うことで，コードクローンに対する一貫した変更を支援する手法が提案されている [16]．

コードクローン同期　主にタイプ 1 のコードクローンを対象に，エディタ上で同時編集を支援ツール [38, 39] や，履歴マイニングを用いたクローン同期を支援するツールが提案されている [40, 41]．AST ベースでコードクローンを検出して同期を支援する手法 [42, 43] が提案されている．

コードクローン追跡　ソフトウェアの進化におけるクローンの系譜を追跡することを目的として，既存のコードクローン検出器の検出結果を利用してクローン系譜を抽出する手法 [44, 45] が提案されている．また，ソースコード中のクローン領域をテキストや位置ではなく構文や構造といった構成情報として管理するこ

とで追跡する手法 [46] が提案されている．

山中らは，コードクローンを追跡して，コードクローンの変更情報を開発者に通知するシステムを提案した [47, 48]．開発者は，変更内容によって分類されたコードクローン情報を確認して変更されたコードクローンが含まれていた場合など，詳細な情報を確認すべきときは 1 つずつ詳細にコードクローンを確認する作業を実施する．しかし，検出されるコードクローンが膨大で，すべてのコードクローンを手動で確認するのは手間がかかることが課題である．

1.3 ファジング

実際の障害や攻撃が発生する前に潜在的な障害を検出し，未然に防ぐための保守作業が予防保守である．その 1 つの手法であるファジングは，未発見の障害や脆弱性を自動的に検出する技術として，近年注目されている．ファジングとは，大量の入力を自動生成し，それらを対象プログラムに対して実行することで，異常な動作やクラッシュなどを検出することを目的とした研究分野である [49]．

当初，ファジングは主にセキュリティにおける脆弱性を検査する目的で研究されてきた．しかし近年では，セキュリティ分野にとどまらず，ソフトウェアテスト自動生成手法の 1 つとしても注目されている．通常のソフトウェアテストは，手動でテストデータが作成され，時間的コストがかけられているにも関わらず，作成したテストケースでは検出困難な障害が存在する可能性がある．ファジングに用いられるランダム生成やミューテーション生成によるテストデータ大量生成は，ソフトウェアの堅牢性や信頼性を向上させ，ソフトウェアの品質保証に効果的である．

ソフトウェアが入力として受け取り得るすべてのデータをテストすることが理論的な理想であるが，実際には入力空間が極めて広大であるため現実的なコストや時間の制約の下で全探索を行うことは不可能である．そこで，限られた時間内にできるだけ多くの潜在的な障害を検出するために，高いコードカバレッジを達成するような大量の入力データを戦略的に生成するファジング手法が盛んに研究されている．ファジングは，入力生成の方法に基づいて，以下の 3 つに分類される [50, 51]．

ホワイトボックスファジング　対象プログラムの内部構造および実行時に収集された情報を解析してファジングする．ホワイトボックスファジングはコンコリックテストを指すことが多い．コンコリックテストは具体的な値の実行とシンボリック実行を組み合わせた手法である [52, 53]．シンボリック実行とは，シンボリックを入力として実行パスを解析し，入力データの制約を特定する手法である．ホワイトボックスファジングは，対象プログラムの静的解析が必要であるため制約が多く適用対象が限られる場合があり，実行時間が長いことが課題である．

ブラックボックスファジング　対象プログラムの内部構造を静的解析せず，プログラムの入出力の挙動のみを観察してファジングする．ブラックボックスファジングはランダム生成やミューテーションを用いて大量の入力データを生成する．効率的な障害発見のために，特定のプロトコルに特化したファザーや，ランダム生成を改善する手法が提案されている [54–56]．ブラックボックスファジングは簡易性や汎用性に優れる一方で，コードカバレッジに偏りが出やすく，深いバグに到達する入力を生成することが難しい．

グレイボックスファジング　対象プログラムの内部構造を静的解析せず，実行中の内部情報を動的に観測してファジングする．グレイボックスファジングは，プログラム解析を用いて静的にテストケース生成するホワイトボックスファジングに比べ，プログラミング言語依存などの制約が少なく，入出力のみでテストケースを生成するブラックボックスファジングよりも効率的にテストケースを探索する．

ホワイトボックスファジングは，ブランチカバレッジを網羅するようなテストケース生成に有効である．一方で，静的解析やソースコードの取得が必要となるため，適用には多くの制約を伴う．ブラックボックスファジングは，プログラムの逆解析ができないなどプログラムの内部情報が読み取れない場合に有効であり，簡易性や汎用性に優れる．ただし，内部構造を活用しないため，深いバグを発見することが難しい．グレイボックスファジングは，実行中に内部情報を動的に観測してファジングを行う．ホワイトボックスファジングと異なり静的解析を行わないため，対象ソフトウェアの規模に対する制限は小さく，ブラックボックスファジングより効率的にテストケースの探索が可能である．

グレイボックスファジングの代表的なツールとして American Fuzzy Lop (AFL) が挙げられる．AFL は，プログラム実行中の内部情報を利用して実行パスを探索し，可能な限り高いコードカバレッジを達成するテストを生成する [1]．AFL は，多くの未知の障害を発見した実績があり，AFL に基づいた派生ツールや特定の用途に特化した派生ツールなどが数多く提案されている [51]．

都築らは，AFL の派生ツール群を比較評価するためのベンチマークを作成し，評価を行った [57]．その結果，後発のファジングツールほどファジング結果が向上し，障害を多く検出することを確認した．しかし，ブランチカバレッジの向上への貢献は小さく，ソフトウェア品質保証への寄与が小さいことを確認した．さらに，AFL では，初期入力データの選択がパス探索の向上に貢献することを明らかにした．有効なテスト対象プログラムに対応する有効な形式の入力データと，無効な入力データの両方を初期入力データとして与えることが，ファジングのパス探索の効率を向上させることに貢献する．このように，ファジングは，アルゴリズム，評価指標，初期入力データ選定，ユーザビリティなど，さまざまな観点から研究されている [57, 58]．

1.4 本研究の概要

本研究では，潜在的な障害に対処するために実施される予防保守や完全化保守の支援を目的として，ソフトウェア保守を困難にする要因の 1 つであるコードクローンに着目し，以下の研究を実施した．

- Cross-Polytope LSH を用いたコードクローン検出のためのパラメータ決定手法
- コードクローン変更管理システムの開発と改善
- コードクローン集約によるファジングの実行効率調査

■Cross-Polytope LSH を用いたコードクローン検出のためのパラメータ決定手法 2 章では，Cross-Polytope LSH を用いたコードクローン検出ツール CCVolti に対し，検出精度を維持しつつ高速化するパラメータの値を決定する手法を提案する．CCVolti は，従来の手法では困難であった意味的に類似するコードクローンを高速に検出可能とした．しかし，CCVolti は検出時間が Cross-Polytope LSH に大きく依存し，Cross-Polytope LSH によるコードクローンの検出漏れが発生するという問題点がある．これは，Cross-Polytope LSH のパラメータ設定により精度と実行時間が大きく変化することが原因である．そこで，本手法では，クローン検出の利用者が与えた再現率の目標値を満たしつつ，できるだけ時間を短縮することを目的として，プロジェクトの規模から適切なパラメータ値を求める線形回帰モデルを構築し，コードクローン検出対象に適した Cross-Polytope LSH に与えるパラメータ値の組を決定する．これにより，CCVolti の利用者は，目標再現率を下げて高速なパラメータを選択することで，大規模なプロジェクトに対して頻繁にコードクローン検出し，修正の即時対応やコードクローンの早期発見，追跡手法への応用を可能とする．評価実験では，20 個のプロジェクトに対して本手法で決定されたパラメータ値を CCVolti に適用し，本手法の有効性を示す．

■コードクローン変更管理システムの開発と改善 3 章では，コードクローンの変更情報を開発者に通知するツールであるコードクローン変更管理システムを，一貫性のない変更を識別し，開発者に提示できるように改善する．既存システム [48] では，構文的に一致するコードクローンの集合（クローンセット）を追跡し，クローンセットの変更情報を開発者に通知する．既存システムは，検出された多くのコードクローン変更情報から，開発者が手作業で一貫性のない変更を識別する必要があり，特に大規模なソフトウェアでの結果を人手で確認するのは困難である．そこで，この課題を解決するため，(1) 意味的に一致するコードクローン検出器の導入，(2) クローンセットの再定義，(3) クローン追跡方法の見直し，(4) クローンセットの詳細分類の追加，の

4 点を改善する．これにより，一貫性のない変更を含むクローンセットの検出と通知が可能となり，大規模ソフトウェアに対する検出結果の確認コストを軽減する．最後に，PostgreSQL のリポジトリの 1 年分のコミット履歴を分析し，実際の一貫性のない変更が行われたクローンセットを示す．

■コードクローン集約によるファジングの実行効率調査　4 章では，ファジング対象のソースコードに含まれるコードクローンが，AFL のパス探索効率に与える影響について調査する．AFL は，プログラムの基本ブロックレベルの実行パスを観測し，それに基づいて未発見のパスを通過するような入力データを効率的に探索するグレイボックスファジングツールである．大規模なソフトウェアに対しては，探索範囲を十分に広げることは難しく，長時間ファジングを実行してもより深い階層にある未知のパスや潜在的な障害箇所へ到達できない可能性がある．その原因として，コードクローンがプログラム中に複数存在する場合，実行時の挙動はほぼ同一であるにもかかわらず，それぞれのコードクローンに対して別々の実行パスとして探索を繰り返すことになるからだと考えた．このような冗長な探索が続くと，より深い階層にある未知のパスや潜在的な障害箇所への到達が遅れ，ファジングの効率が低下する可能性がある．そこで，基本ブロックを含むコードクローンを集約することで，AFL が観測するパスの総数を削減し，未発見のパスに到達しやすくなるという仮説のもと，コードクローン集約前後のプログラムに対する AFL の比較評価を実施する．

1.5　各章の構成

以降，第 2 章では Cross-Polytope LSH を用いたコードクローン検出のためのパラメータ決定手法について述べる．第 3 章ではコードクローン変更管理システムの開発と改善について述べる．第 4 章ではコードクローン集約によるファジングの実行効率調査について述べる．最後に，第 5 章では本論文のまとめと将来の研究方針について述べる．

第 2 章

Cross-Polytope LSH を用いたコードクローン検出のためのパラメータ決定手法

2.1 まえがき

ソフトウェア開発において，コピーアンドペーストによる再利用等が原因で，コードクローンが頻繁に発生する．コードクローンを保守するために，ソースコード中からコードクローンを識別して管理する必要がある．しかし，ソースコードの規模が大きくなるとソースコード中に含まれるコードクローンも膨大な量となる．手作業でコードクローンを管理することが困難となるため，コードクローンをソースコードから自動的に検出するための手法が提案されている [13, 18].

横井らが提案したブロッククローン（コードブロック単位のコードクローン）検出ツール CCVolti[*1]は情報検索技術 [24] と局所性鋭敏型ハッシュ (LSH) [25] を利用することによって，従来の手法では困難であった意味的に処理が類似したコードクローンを検出できる [21]．CCVolti におけるコードブロックは，関数と，関数内部の if, for 文等の波括弧で囲まれた部分である．CCVolti は，入力ソースコードに対して構文解析を行い，コードブロックの抽出を行う．その後，抽出した各コードブロックを情報検索技術の 1 つである TF-IDF (Term Frequency-Inverse Document Frequency) 法 [24] に基づいて特徴ベクトルに変換する．最後に，特徴ベクトル間の類似度が閾値以上の対をコードクローン対として検出する．CCVolti は高速にコードクローン対を検出するために，近似最近傍探索アルゴリズム Cross-Polytope LSH [25] を用いている．Cross-Polytope LSH とは，高次元なベクトル集合を確率的にハッシュ化して最近点を求める近似最近傍探索アルゴリズムである局所性鋭敏型ハッシュ (LSH) の一種で

[*1] https://github.com/k-yokoi/CCVolti

ある．Cross-Polytope LSH はパラメータ値の与え方によって，精度や実行速度を変化させることができる．CCVolti は，Cross-Polytope LSH を利用するために，LSH ライブラリ FALCONN[*2]を使用した．

CCVolti は，既存のコードクローン検出法と比べて高い精度でコードクローンを検出できる [21]．また，大規模なプロジェクトに対して現実的な計算時間でコードクローン検出可能である．実際，15MLOC の Linux Kernel に対して CCVolti を用いてコードクローン検出した場合，20 分程度で検出が完了し，100MLOC においても 4 時間程度でコードクローンを検出した．

一方，CCVolti には次の 2 つの問題点が挙げられる [31]．1 つ目は，Cross-Polytope LSH を用いて類似度が閾値以上のベクトル対を探索する処理で，約 10% の検出漏れが発生する場合がある点である．2 つ目は，大規模ソフトウェアの検出に多大な計算時間を必要とし，頻繁なコードクローン検出ができない点であり，検出時間の約 90% を類似度が閾値以上のベクトル対を探索する時間が占めている．頻繁なコードクローン検出は，修正の即時対応やコードクローンの早期発見のために必要である．これらの問題は，CCVolti が Cross-Polytope LSH のパラメータをクローン検出対象プロジェクトに対して調整していないことが原因である．特徴ベクトルの数や次元の大きさはプロジェクトごとに異なるため，ベクトル対の探索の精度と実行速度のトレードオフとなるパラメータはプロジェクトごとに異なる．しかし，FALCONN には 10 を超えるパラメータが存在するため，CCVolti の利用者が Cross-Polytope LSH のアルゴリズムを理解し，クローン検出するたびにクローン検出対象プロジェクトに適したパラメータを調整することは困難である．

この問題を解決するために，本研究では，類似探索の検出漏れを最小限に抑えつつ，処理時間の削減を両立するため，Cross-Polytope LSH に与えるパラメータを自動的に決定する手法を提案する．本手法では，類似度が閾値以上のベクトル対のうち，検出されたベクトル対の割合を再現率と定義し，クローン検出の利用者が与える再現率の目標値を目標再現率と定義する．本手法では，学習用プロジェクトに対して複数のパラメータ値の組で実行して計測したデータを学習データとして線形回帰モデルを作成し，コードクローン検出対象プロジェクトの規模を表すメトリクスを入力とし，目標再現率を満たしつつ実行時間が最小となるパラメータ値の組を決定する．これにより，CCVolti の利用者は，目標再現率を下げて高速なパラメータを選択することで，大規模なプロジェクトに対して頻繁なコードクローン検出が可能となる．

評価実験では，10 個の C 言語プロジェクトと 10 個の Java プロジェクトを学習用プロジェクトとして，本手法で決定されたパラメータを使用する CCVolti を用いてコードクローン検出を実施した．実験結果では，多くの場合で再現率が目標再現率を上回ることを確認できた．また，本手法で決定されたパラメータは FALCONN のデ

[*2] https://falconn-lib.org/

フォルトのパラメータ値より多くの場合で高速であることを確認した．デフォルトのパラメータ値での再現率と同等の再現率が得られるように目標再現率 0.99 としたときの探索時間と比較して，目標再現率 0.8 での探索時間はおよそ半減できていた．

以降，2.2 節では，コードクローン検出法 CCVolti，Cross-Polytope LSH とその関連技術について述べる．2.3 節では，CCVolti の利用者が与えた再現率の目標値を満たしつつ，高速な Cross-Polytope LSH のパラメータ決定手法を示す．2.4 節では，本手法の有効性の評価を行う．2.5 節では，実験結果をふまえた本手法の有効性の範囲と妥当性について考察する．最後に，2.6 節では，まとめと今後の課題について述べる．

2.2 コードクローン検出とその関連技術

本節では，局所性鋭敏型ハッシュ (LSH)，近似最近傍探索アルゴリズム Cross-Polytope LSH，コードクローン検出ツール CCVolti について述べる．

2.2.1 局所性鋭敏型ハッシュ (LSH)

LSH とは，近似最近傍探索問題をハッシュを用いて解くアルゴリズムである [25]．近似最近傍探索問題とは，入力ベクトルに対してベクトル集合であるデータセットの中から近傍ベクトルを近似的に高速に見つける問題であり，最近点問題の一種である．近傍ベクトルとは，入力ベクトルに対して一定の類似度以上のベクトルのことである．LSH のアルゴリズムは，類似度の閾値 θ と近似因数 $c < 1$ に対して，ベクトル集合の中に入力ベクトルとの類似度が θ 以上のベクトルが存在するとき，類似度が $c\theta$ 以上のすべてのベクトルを返すことが数学的に保証されている [25]．

入力ベクトルに最も近いベクトルを求める最も基本的な手法は，データセット内のすべてのベクトルと類似度を計算する方法であり，計算量は $O(n^2d)$ となる．この手法は，ベクトル数 n やベクトルの次元数 d が大きい場合，計算時間が非常に長くなるという問題がある．例えば，15MLOC の Linux Kernel に対して CCVolti は約 36 万個のベクトルを生成し，ベクトル対の組合せ総数 6.5×10^{10} 回の類似度計算を必要とし，ベクトルの類似度計算のみで数日を要する．一方，LSH は，ハッシュを用いて入力ベクトルの近傍ベクトルを求め，近傍ベクトルに対してのみ類似度計算を行う．この手法の計算量は $O(n^{1+\rho}d), (0 \leq \rho \leq 1)$ であり，全探索に比べて高速に最も近いベクトルを求めることが可能となる．実際，15MLOC の Linux Kernel に対して 20 分で類似度を計算できる．

ベクトルをハッシュ値に変換するハッシュ関数に対して，2 つのベクトル x, y が同じハッシュ値を取ることをハッシュの衝突という．LSH のハッシュ関数は，類似度が高いベクトル同士がハッシュの衝突を起こしやすくなるように定義される．ベクトルのハッシュ値が入力ベクトルのハッシュ値と衝突するとき，そのベクトルは入力ベク

トルの近傍ベクトルとなる．2 つのベクトル x, y に対して類似度 $S(x, y)$ が定義された d 次元空間上において，x, y のハッシュ値が衝突する確率を衝突確率と呼ぶ．

データセット中から，ある入力ベクトルに対する近傍ベクトルを LSH を用いて探索する処理の時間計算量は $O(dn^{\rho})$ となる [25]．ここで，d はベクトルの次元数，n はベクトル集合のベクトル数を表し，ρ は式 2.1 のように表される．

$$\rho = \frac{\log(1/p)}{\log(1/q)} \tag{2.1}$$

ここで，p は類似度 $S(x, y)$ が θ 以上となる 2 つのベクトルの衝突確率を表し，q は類似度 $S(x, y)$ が $c\theta$ 以下となる 2 つのベクトルの衝突確率を表す．ρ が小さいほど実行時間の計算量のオーダーが小さくなるため，ρ は LSH のアルゴリズムの評価基準として用いられる．

2.2.2 近似最近傍探索アルゴリズム Cross-Polytope LSH

LSH の一種である Cross-Polytope LSH は，d 次元単位球上のベクトル集合に対して有効性が保証されており，効率的な実装も可能である [25]．コードクローン検出ツール CCVolti が用いる LSH ライブラリ FALCONN は，大規模なベクトル集合の近似最近傍探索問題を解くための実装として Andoni らにより開発された．本節では，Cross-Polytope LSH のアルゴリズム，および Cross-Polytope LSH を用いた類似探索について説明する．

Cross-Polytope LSH のアルゴリズム

Cross-Polytope LSH は，2 つのベクトル x, y に対するユークリッド距離 $d(x, y) = \|x - y\|$ に基づいて近傍ベクトルを探索するアルゴリズムである．コサイン類似度 $C(x, y) = \frac{x \cdot y}{\|x\|\|y\|}$ は式 $C(x, y) = 1 - \frac{d(x,y)^2}{2}$ によってユークリッド距離と 1 対 1 対応する．本節では，CCVolti が用いるコサイン類似度に基づいて議論する．

Cross-Polytope LSH のハッシュ関数を用いた，d 次元ベクトル x に対するハッシュ値の計算方法について説明する．まず，ベクトル x を正規化し，ランダム行列 $A \in \mathbb{R}^{d \times d}$ を乗算してランダム回転を行い，ベクトル $y = Ax/\|Ax\|$ に変換する．次に，ランダム回転後のベクトル y に対して，正規直交基底の基底ベクトルとそれらの逆ベクトル $\{\pm e_i\}_{1 \le i \le d}$ の中から最も類似度が高いベクトルを求める．最も類似度が高いベクトルの符号と添え字 i によって，x のハッシュ値が $\pm i$ に決定される．すなわち，Cross-Polytope LSH のハッシュ関数は，行列 A を用いて入力ベクトルをランダムに回転させ，回転後のベクトルが d 個に分割された単位球のどの区画に含まれるかを，入力ベクトルのハッシュ値とする．

ベクトルにランダム行列を掛けることにより，ベクトルがランダムに回転し，類似度が高いベクトル対が一定の確率で衝突を起こすようになる．CCVolti が用いる

Cross-Polytope LSH ライブラリ FALCONN では，前処理で次元圧縮をしたり，ランダム回転の処理に高速アダマール変換を用いたりするなど，メモリ削減や高速化を行っている [29, 30]．FALCONN には 10 種類のパラメータが存在し，次元圧縮後の次元数などの検出結果に影響を与えるパラメータと，メモリ上のデータ保持方法などメモリや計算速度に影響を与えるパラメータがある [31]．

ある類似度 δ に対して 2 つのベクトル x, y が $C(x, y) = \delta$ をみたすとき，Cross-Polytope LSH の衝突確率 P_T は式 2.2 のように表される [25]．

$$\ln \frac{1}{P_T} = \frac{1-\delta}{1+\delta} \cdot \ln T + O_\delta(\ln \ln T) \tag{2.2}$$

T は区画の分割数を表す．$O_\delta(\ln \ln T)$ は δ に依存する誤差項であり，$\ln \ln T$ に比例する．誤差項 $O_\delta(\ln \ln T)$ は，区画の分割数 T が大きくなるほど 0 に近づく [25]．また，式 2.2 と同様に，$C(x, y) = c\delta$ のときの衝突確率 Q_T を式 2.3 とする．

$$\ln \frac{1}{Q_T} = \frac{1-c\delta}{1+c\delta} \cdot \ln T + O_{c\delta}(\ln \ln T) \tag{2.3}$$

Cross-Polytope LSH による近傍ベクトルの検出時間の時間計算量 $O(dn^\rho)$ の ρ は，式 2.2 と式 2.3 を用いて，2.2.1 節の式 2.1 から式 2.4 に変形でき，T に依存して決まることが分かる．

$$\rho = \frac{\frac{1-\delta}{1+\delta} \cdot \ln T + O_\delta(\ln \ln T)}{\frac{1-c\delta}{1+c\delta} \cdot \ln T + O_{c\delta}(\ln \ln T)} \tag{2.4}$$

Cross-Polytope LSH を用いた類似探索

類似探索とは，ベクトル集合から類似度が閾値 θ 以上のベクトル対を探索することである．Cross-Polytope LSH を用いた類似探索のアルゴリズムは，以下の 3 つのステップで構成される [25]．

STEP A　ベクトルの集合から L 個のハッシュテーブルを作成
STEP B　いずれかのハッシュテーブルで，ハッシュ値が衝突するベクトル対を抽出
STEP C　STEP B で抽出したすべてのベクトル対の類似度を計算し，類似度が閾値以上であるベクトル対をクローンペアとして検出する

Cross-Polytope LSH のランダム性から，異なるハッシュ関数を複数用意できる．

ハッシュテーブル 1 つ当たり K 個のハッシュ関数を使用することで，最も近いベクトルの候補を減らし，より高速に最も近いベクトルを検出することができる．K 個のハッシュ関数の衝突確率をそれぞれ，$P_{T_1}, ..., P_{T_K}$ とすると，ハッシュテーブル 1 つ当たりの衝突確率は式 2.5 のように表される [59]．

$$P_{T,K} = \prod_{i=1}^{K} P_{T_i} \tag{2.5}$$

ただし，ライブラリ FALCONN では，T の上限 d に対して $P_{T_1}, ..., P_{T_K-1} = P_d$ となるように実装されているため，FALCONN でのハッシュテーブル 1 つ当たりの衝突確率は式 2.6 のように表される．

$$P_{T,K} = {P_d}^{K-1} \cdot P_T \tag{2.6}$$

L 個のハッシュテーブルを用意し，いずれかのハッシュテーブルで衝突したベクトル対をクローンペアの候補として抽出する．ハッシュテーブルを増やすことで，探索時間は増加するが，衝突確率を上げて検出漏れを減らすことができる．このとき，式 2.5 で表したハッシュテーブル 1 つ当たりの衝突確率 $P_{T,K}$ に対して，L 個のハッシュテーブルでの衝突確率 $P_{T,K,L}$ は式 2.7 のように表される．

$$P_{T,K,L} = 1 - (1 - P_{T,K})^L \tag{2.7}$$

2.2.3 コードクローン検出ツール CCVolti のアルゴリズムと問題点

本節では，コードクローン検出ツール CCVolti のアルゴリズムと，その問題点について述べる．

大規模なソースコード中のコードクローンを手作業で管理することが困難であるため，1.2.1 節に述べたようにコードクローンを自動で検出する手法が多数提案されている．コードクローン検出手法は，その手法で用いる検出単位によって，行単位の検出，字句単位の検出，抽象構文木を用いた検出，プログラム依存グラフを用いた検出，メトリクスなどその他の技術を用いた検出に分類することができる [18]．

横井らが提案した抽象構文木を用いた検出ツール CCVolti は，情報検索技術を利用することにより，タイプ 1 からタイプ 4 までのブロッククローン（コードブロック単位のコードクローン）を検出できる [21]．コードブロックとは，if 文や for 文や関数などの波括弧で囲まれたコード片を指す．CCVolti は，既存のコードクローン検出法と比べて高い精度でコードクローンが検出でき，大規模なプロジェクトに対して現実的な計算時間でコードクローン検出可能である．実際，CCVolti を用いて 15MLOC の Linux Kernel のコードクローン検出を行うと，20 分程度で検出が完了し，100MLOC においても 4 時間程度で検出可能であった．

CCVolti は以下のステップで入力ソースコードからコードクローンを検出する．

STEP 1 ソースコードの構文解析を行い，抽象構文木を生成
STEP 2 抽象構文木からワードとコードブロックを抽出
STEP 3 TF-IDF 法 [24] により，コードブロック単位の特徴ベクトルを計算
STEP 4 Cross-Polytope LSH を用いた類似探索を行い，コサイン類似度が閾値 0.9 以上のクローンペアを検出

STEP 1 では，ソースコードの構文解析を行い，抽象構文木を生成する．STEP 2 では，STEP 1 で生成した抽象構文木からワードとコードブロックを抽出する．ワードとは，予約語と識別子名を構成する言語とする．STEP 3 では，TF-IDF 法によりコードブロック単位の特徴ベクトルを計算する．TF-IDF 法とは，ワードの出現頻度によって重み付けを行うベクトル化手法である [24]．STEP 4 では，2.2.2 節で述べた Cross-Polytope LSH を用いた類似探索を行い，コサイン類似度が閾値 0.9 以上のクローンペアを検出する．

徳井らは先行研究において，LSH を用いるコードクローン検出法に対して，以下の 2 つの問題点を指摘した [31]．

- Cross-Polytope LSH を用いた類似探索において，検出漏れが発生する場合があることを指摘している．同時修正箇所の検出などの目的で CCVolti を利用する場合，高い精度が求められるにも関わらず，類似探索において検出漏れが多く発生することは問題である．
- CCVolti の Cross-Polytope LSH を用いた類似探索の処理時間が CCVolti のコードクローン検出時間の約 90% を占めており，クローン検出時間が類似探索の処理時間に大きく依存していることを指摘している．

これらの問題は，CCVolti が Cross-Polytope LSH のパラメータをクローン検出対象プロジェクトに対して調整していないことが原因である．類似度が閾値以上のベクトル対の探索の精度と実行速度は Cross-Polytope LSH に与えるパラメータによって調整できる．プロジェクトごとに特徴ベクトルの数や次元の大きさは異なるため，類似度が閾値以上のベクトル対の探索の精度と実行速度のトレードオフとなるパラメータはプロジェクトごとに異なる．しかし，CCVolti の利用者がクローン検出するたびに対象プロジェクトに適した Cross-Polytope LSH のパラメータを調整することは困難である．

2.3 Cross-Polytope LSH に与えるパラメータ決定手法

本節では，2.2.3 節で述べたコードクローン検出ツール CCVolti の問題点を解決するために，CCVolti の利用者が与えた目標再現率を超える再現率となり，かつ高速であるための Cross-Polytope LSH に与えるパラメータ決定手法を提案する．本手法は，学習用プロジェクトの規模に対する適切なパラメータ値の組を学習データとして，プロジェクトの規模に対するパラメータ値の組を決定する線形回帰モデルを作成し，コードクローン検出対象プロジェクトを入力として線形回帰モデルを適用することで，目標再現率を超える再現率となりつつ，高速なパラメータ値の組を求める．本研究における再現率は，類似度が閾値以上のベクトル対の数に対して Cross-Polytope LSH が検出したベクトル対の数の割合を指す．

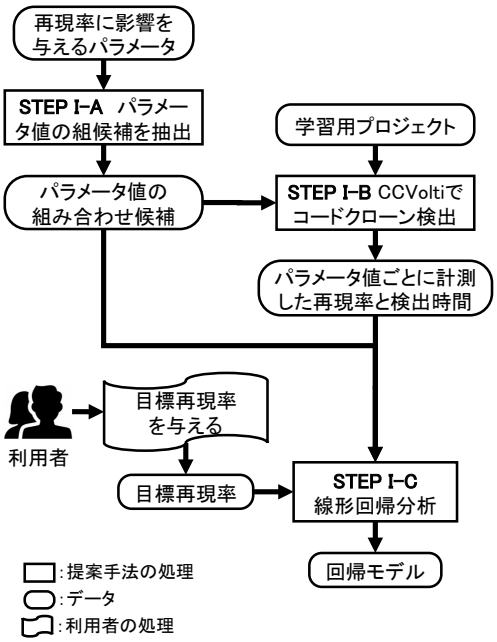


図 2.1　提案手法 STEP I の学習プロセス

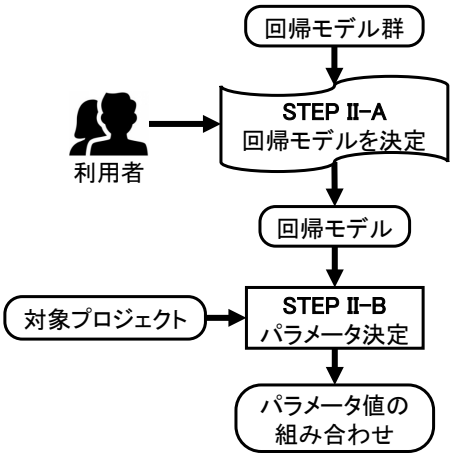


図 2.2　提案手法 STEP II の適用プロセス

本手法の目的は CCVolti の利用者が指定した目標再現率を超えつつできるだけ高速なパラメータを決定することである．コードクローン検出の目的に応じて目標再現率を設定し，速度を優先することや精度を優先することができる．例えば，再現率を落とすことで比較的高速にコードクローン検出すると，コードクローンがどの程度存在するかの予備分析を低コストで実行できる．また，精度を優先したパラメータでコードクローン検出すると，より正確に同時修正すべきクローンペアを検出し，保守作業としてコードクローンをリファクタリングできる．しかし，リファクタリングはコスト削減を目的としているため，リファクタリングに投入できる時間に限界がある．そのため，本手法は精度を優先しつつできるだけ高速にコードクローン検出を行うパラメータを決定する．

本手法は，利用者が決定した目標再現率に対して学習用プロジェクトを用いて線形回帰モデルを作成する学習プロセスと，クローン検出対象プロジェクトの規模に対して線形回帰モデルを用いてパラメータ値を決定する適用プロセスからなる．学習プロセスでは，学習用プロジェクトの規模に対する適切なパラメータ値の組を学習データとして，クローン検出対象プロジェクトの規模に対するパラメータ値の組を決定する線形回帰モデルを作成する．適用プロセスでは，線形回帰モデルを用いて，クローン検出対象プロジェクトを入力として，目標再現率を超える再現率となりつつ，高速なパラメータ値を決定する．

本手法で作成する線形回帰モデルはプロジェクトの規模とパラメータ値の関係を推論するモデルである．利用者は学習プロセスで作成した線形回帰モデルを適用プロセ

スで再利用可能であり，さらに，未学習のプロジェクトに対して適用可能である．しかし，Java 言語と C 言語のプロジェクトで学習した線形回帰モデルを，スクリプト言語や関数型言語などの異なるドメインのプロジェクトに適用する場合，プロジェクトの規模とパラメータ値の関係が異なると考えられる．したがって，利用者はコードクローン検出対象プロジェクトと同じドメインのプロジェクトを用意して学習プロセスを実施する必要がある．

学習プロセスでは，学習データ作成と線形回帰分析に時間を要するため，利用者が求める目標再現率に対応する線形回帰モデルを事前に作成する必要がある．一方で，作成した線形回帰モデルは文法が類似する言語のプロジェクトに対して汎用的に利用可能である．したがって，コードクローン検出利用者だけではなく開発現場の環境整備の担当者や CCVolti の開発者が学習プロセスをあらかじめ実施し，利用者に線形回帰モデルを提供することが可能である．

図 2.1 は STEP I の学習プロセスを示す．学習プロセスでは，学習用プロジェクトを用いて線形回帰モデルを以下の 3 つのステップで作成する．

STEP I-A　Cross-Polytope LSH に与えるパラメータから，再現率に影響を与えるパラメータを抽出する．抽出したパラメータが再現率と探索時間に与える影響を分析し，パラメータ値の組み合わせの候補を抽出する．

STEP I-B　抽出したパラメータ値の組み合わせの候補を与えた Cross-Polytope LSH を用いて CCVolti で学習用プロジェクトに対してコードクローンを検出し，パラメータごとの再現率と類似探索の探索時間を計測する．類似探索の探索時間とは 2.2.3 節で述べた STEP 4 の類似探索にかかる時間である．

STEP I-C　学習用プロジェクトの実験結果を用いて，プロジェクトの規模の対して目標再現率を超えつつ高速なパラメータ値の組み合わせを決定する線形回帰モデルを作成する．

図 2.2 は STEP Ⅱ の適用プロセスを示す．適用プロセスでは，学習プロセスで生成した 1 つ以上の線形回帰モデルとクローン検出対象プロジェクトを入力として以下の 2 つのステップでパラメータを決定する．

STEP Ⅱ-A　STEP I で作成した線形回帰モデル群から利用者が目標再現率に対応する線形回帰モデルを選択する．

STEP Ⅱ-B　クローン検出対象プロジェクトを入力として，選択した線形回帰モデルを用いてパラメータ値を決定する．

ただし，CCVolti が用いる LSH ライブラリ FALCONN は複数のパラメータを持ち，高速化のためのパラメータなどのパラメータが存在する．そこで，Cross-Polytope LSH を用いた類似探索の再現率に影響するパラメータを 2.3.1 節で示す．

2.3.2 節以降，提案手法の各ステップの詳細を示す．

2.3.1 類似探索の再現率に関係するパラメータ

本節では，類似探索の再現率に影響を与えるパラメータを過不足なく抽出するために，本手法における再現率の定義と算出方法を示し，定義した再現率の期待値が Cross-Polytope LSH の衝突確率 $P_{T,K,L}$ に一致することを示す．本節では，類似探索の再現率を目標再現率以上の値とし，できるだけ高速となるために，提案手法では，再現率に影響を与える 3 つのパラメータ T, K, L を決定することを示す．

最初に，本研究における再現率 r を定義する．ベクトル集合に対して，U_{all} を閾値 θ 以上の類似度であるすべてのベクトル対の集合，U_{lsh} を LSH を用いた類似探索により検出したベクトル対の集合として，再現率 r は式 2.8 のように表される．ここで $|\cdot|$ は集合の要素数を表す．

$$r = \frac{|U_{\mathrm{lsh}}|}{|U_{\mathrm{all}}|} \tag{2.8}$$

LSH を用いた類似探索は，U_{all} に含まれる可能性があるベクトル対を LSH を用いて探索し，LSH で取得した全てのベクトル対の類似度を計算し，閾値 θ 以上の類似度であるベクトル対を得る．そのため，U_{lsh} は包含関係 $U_{\mathrm{lsh}} \subseteq U_{\mathrm{all}}$ を常にみたす．類似探索の適合率を，LSH を用いて検出したベクトル対集合 U_{lsh} に対して閾値 θ 以上であるベクトル対の割合とすると，包含関係 $U_{\mathrm{lsh}} \subseteq U_{\mathrm{all}}$ だから，類似探索の適合率は常に 1 となる．

あるプロジェクトに対して閾値を θ として類似探索を用いたコードクローン検出を実行したとき，類似探索の再現率は以下の手順で算出される．

STEP i LSH を利用せずすべてのベクトル対の類似度を計算して，類似度が閾値 0.9 以上のベクトル対を求め，ベクトル対の数 $|U_{\mathrm{all}}|$ を計測する．

STEP ii LSH を用いた類似探索を実行し，類似度が閾値 0.9 以上のベクトル対の数 $|U_{\mathrm{lsh}}|$ を計測する．

STEP iii 2 つの値を式 2.8 に代入して再現率 r を算出する．

次に，Cross-Polytope LSH の衝突確率と CCVolti の類似探索の再現率の期待値が一致することを示す．2.2.2 節で述べた Cross-Polytope LSH を用いた類似探索において，衝突確率は式 2.7 のように表される．本研究における再現率の定義と，Cross-Polytope LSH の衝突確率の定義から以下の定理を導くことができる．

定理 1. 閾値 θ に対して類似探索を行うとき， $S(x, y) \geq \theta$ である 2 つのベクトル x, y に対する *LSH* の衝突確率 $P_{T,K,L}$ と，再現率の期待値 E は 一致する．

Proof. 2.2.2 節より，確率 $P_{T,K,L}$ は L 個のハッシュテーブルに対する衝突確率を表し，ある 1 つの類似ペアが検出できる確率といえる．すべてのベクトル対集合 U_{all} の

ベクトルは，それぞれ確率 $P_{T,K,L}$ で衝突するから，衝突するベクトル対の数は二項分布に従う．したがって，検出できるベクトル対の数の期待値 E_{lsh} は，ベクトル対の数 $|U_{\mathrm{all}}|$ と 1 つのベクトル対が衝突する確率を用いて $E_{\mathrm{lsh}} = |U_{\mathrm{all}}| \times P_{T,K,L}$ ように計算される．また，再現率の期待値 E は，検出できるベクトル対の数の期待値 E_{lsh} とベクトル対の数 $|U_{\mathrm{all}}|$ を用いて $E = \frac{E_{\mathrm{lsh}}}{|U_{\mathrm{all}}|} = P_{T,K,L}$ と表される．

よって，再現率の期待値と衝突確率は一致する．実際に，いくつかのパラメータ値の組み合わせにおいて 20 プロジェクトに対して実験を行い再現率を計測したところ，再現率の信頼区間に式 2.7 から算出した衝突確率が含まれていた． □

定理 1 より，再現率の期待値は衝突確率と一致する．Cross-Polytope LSH の衝突確率 $P_{T,K,L}$ は，Cross-Polytope LSH のパラメータである，区画の分割数 T，ハッシュ関数の数 K，ハッシュテーブル数 L に依存して増減する．したがって，本手法は，再現率が目標再現率を超えつつ，できるだけ高速となるために，類似探索の再現率に影響を与える 3 つのパラメータ T, K, L の値を決定する．

2.3.2 STEP I-A パラメータ値の組候補を抽出

3 つのパラメータ T, K, L はそれぞれが独立に再現率と探索時間に影響を与えるため，再現率が目標再現率を超えつつ，できるだけ高速となるための T, K, L のパラメータ値を同時に決定する必要がある．3 つのパラメータ T, K, L の値を同時に決定するために，本節では，線形回帰モデルの目標変数とするラベルを示す．ラベルとは，T の上限の値を d とするとき，区画の分割数 T とハッシュ関数 K の値の組 (T, K) と 1 対 1 対応する式 $(K-1)\ln d + \ln T$ の値とする．

区画の分割数 T とハッシュ関数 K の値の組 (T, K) は，1 対 1 対応する式をハッシュテーブル 1 個当たりの衝突確率の式 2.5 から導出できる．式 2.6 の両辺の逆数に対して底 2 の対数を取り，右辺の各項に式 2.2 を誤差項を無視して代入し，式 2.9 を導く．

$$\ln \frac{1}{P_{T,K}} = \frac{1-\delta}{1+\delta}((K-1)\ln d + \ln T) \tag{2.9}$$

右辺に現れた式 $(K-1)\ln d + \ln T$ について，T, K が自然数であり d は T の上限であることから，任意の整数 $K_1 > K_2$ に対して $(K_1-1)\ln d + \ln T > (K_2-1)\ln d + \ln T$ である．したがって，T と K の値の組合わせと $(K-1)\ln d + \ln T$ の値は 1 対 1 に対応している．これ以降，ハッシュ関数の数 K と区画の分割数 T の組を，ラベル $(K-1)\ln d + \ln T$ として表す．

ラベルと L の性質を調査するため，Linux Kernel を対象に予備実験をした．予備実験の内容と結果を付録 1 に示す．実験結果から，K と T の組を表すラベルとハッシュテーブル数 L について，以下の性質を確認した．

- ラベルを増加すると，再現率は減少し，探索時間は指数的に減少する．
- ハッシュテーブルの数を増加すると，再現率は増加し，探索時間は線形に増加する．
- さらに，探索時間に与える影響は，ラベルの方がハッシュテーブル数より大きい．

1, 2 番目の性質から，ラベルが小さいと再現率は高くなり，ハッシュテーブルの数が大きいと再現率は高くなるといえる．3 番目の性質から，ラベルのほうが探索時間に大きな影響を与えるため，ラベルの値をハッシュテーブルより先に決定することで，探索時間をより短くできると考えられる．ラベルの値に対して，ハッシュテーブル数 L は再現率を目標再現率以上になるために十分な値を決定する必要がある．できるだけ高速なラベルを決定するための線形回帰モデルと，ラベルから L を一意に決定する方法を STEP I-C に示す．したがって，本手法はラベルの値を決定することで，再現率が目標再現率を超えつつ，できるだけ高速となるための 3 つのパラメータ T, K, L を自動的に決定する．

パラメータ T, K, L が取りうる値の組の候補として，ラベルとハッシュテーブル数に取りうる値の範囲の総当たりが考えられる．ラベル $label = (K-1)\ln d + \ln T$ に 1 以上の整数値を 1 刻みで与えるような値を区画の分割数 T とハッシュ関数の数 K に与える．ハッシュテーブル数 L には 1 以上の整数値を 1 刻みで与える．これらの総当たりのパラメータ値の組の候補に対して学習データを生成する．

2.3.3 STEP I-B 学習データの生成

本節では，学習用プロジェクトを用いて，線形回帰モデルを作成するための学習データを作成する．学習データは，学習用プロジェクトのコードブロック数，Cross-Polytope LSH に与えるパラメータ値の組に対する類似探索の再現率と探索時間とする．

STEP I で抽出したパラメータ T, K, L の組の候補を Cross-Polytope LSH に与え，CCVolti を用いて学習用プロジェクトに対するコードクローン検出を実行し，コードブロック数，類似探索の再現率，探索時間を計測する．計測結果を含めた 4 つのデータ，Cross-Polytope LSH に与えたパラメータ値の組，コードブロック数，類似探索の再現率，探索時間，の組を学習データとする．学習プロセスで作成した線形回帰モデルを適用プロセスで，再利用可能，あるいは未学習のプロジェクトに対して適用可能とするために，コードクローン検出対象プロジェクトと同じドメインの学習用プロジェクトを用意する必要がある．また，本手法が生成する線形回帰モデルはコードブロック数とパラメータ値の関係を示すため，コードブロック数が異なるプロジェクトを複数用意することが必要である．本評価実験では，コードブロック数が 2,000 以上ある OSS プロジェクトを収集し学習用データとして利用した．

2.3.4 STEP I-C 線形回帰分析

本節では，STEP I-B で作成した学習データを用いて，コードクローン検出対象プロジェクトのコードブロック数に対して，目標再現率を超えつつできるだけ探索時間を短縮するためのパラメータ値を推論する線形回帰モデルを作成する．STEP Ⅱ では STEP I で作成した線形回帰モデルからパラメータ決定に利用するモデルを選択する．そのため，STEP I の実施者である利用者あるいは開発現場の環境整備の担当者や我々は，CCVolti の利用者が求める目標再現率に対応する線形回帰モデル群を STEP I で作成する．

STEP I-C では，STEP I-B で取得した学習用プロジェクトに対する再現率の計測結果から，すべてのプロジェクトの再現率が目標再現率を超えるパラメータ組 $(label, L)$ をすべて抽出する．抽出されたパラメータ組 $(label, L)$ は，すべての学習用プロジェクトで再現率が目標再現率を超えるため，任意のプロジェクトで再現率の期待値が目標再現率を超える．そのため，線形回帰モデルによりラベルの値を決定すると一意にハッシュテーブル数を決定する．

再現率に基づいて抽出したパラメータ組に対して，STEP I-B で取得した学習用プロジェクトに対する探索時間の計測結果から，プロジェクトごとに探索時間が最も短いパラメータ組を抽出する．学習用プロジェクトごとに学習データから抽出したパラメータ組を用いて，コードブロック数を説明変数とし，ラベルを目標変数とする線形回帰モデルを作成する．線形回帰モデルの説明変数としてプロジェクトの規模を表すメトリクスにコードブロック数を用いた理由は，CCVolti がコードブロック単位のコードクローン検出手法だからである．一方，プロジェクトの規模を表すメトリクスは複数あるのに対し，コードブロック数のみを説明変数としたのは，行数やメソッド数などのメトリクス同士の相関が強く，多重共線性により結果を偏らせると判断したからである．

線形回帰モデルにコードクローン検出対象プロジェクトのコードブロック数を入力すると，ラベルの値が決定される．ラベルの値が決定されると，ラベルを表す式 $label = (K-1)\ln d + \ln T$ から，ラベルの値に対応する 2 つのパラメータ T, K を求めることができる．また，すべての学習用プロジェクトに対して抽出した目標再現率を超える再現率であるパラメータ組を用いて，ラベルの値に対して探索時間が最短の L を決定する．L は小さくなるほど探索時間が線形に短くなるため，目標再現率を超える再現率であるパラメータ組に対して線形回帰モデルで決定されたラベルを含むパラメータ組の内，最も小さい L に決定する．したがって，作成した線形回帰モデルは，コードブロック数を説明変数として，Cross-Polytope LSH に与える 3 つのパラメータ T, K, L を決定することができる．

2.3.5 STEP Ⅱ 線形回帰モデルを用いたパラメータ決定

本節では，学習プロセスで作成した線形回帰モデルを用いて，本手法の利用者がパラメータを決定する手順 STEP Ⅱ-A, Ⅱ-B を示す,

STEP Ⅱ-A では，STEP I で作成された線形回帰モデル群から，利用者が目標再現率に対応する線形回帰モデルを選択する．線形回帰モデル群は，利用者あるいは開発現場の環境整備の担当者や我々によって，目標再現率ごとに STEP I にしたがって事前に作成される．利用者は，作成された線形回帰モデル群から，速度を優先する目標再現率に対応する線形回帰モデルや，精度を優先する目標再現率に対応する線形回帰モデルを選択する．

STEP Ⅱ-B では，STEP Ⅱ-A で選択した線形回帰モデルを用いて，コードクローン検出対象プロジェクトを入力として，区画の分割数 T，ハッシュ関数の数 K，ハッシュテーブル数 L の値を決定する．STEP I-A で示した通り，他のパラメータは再現率に影響を与えないため，任意のパラメータ値を用いてよい．選択した線形回帰モデルにプロジェクトのコードブロック数を入力すると，ラベルの値が決定され，自動的にパラメータ組 (T, K, L) が決定される．ラベルと K, T に関する式 $label = (K-1)\ln d + \ln T$ から，ラベルの値に対してパラメータ T, K が一意に決まる．また，STEP I-C で示したようにラベルの値に対して探索時間が最短の L を一意に決定できる．線形回帰モデルを用いて，コードクローン検出対象プロジェクトのコードブロック数から，目標再現率を超えつつ，できるだけ高速なパラメータ値の組 (T, K, L) を決定する．

2.4 評価実験

本実験では，本手法の有効性を評価するために，LSH ライブラリ FALCONN のデフォルトのパラメータ値に対して，本手法に基づいて決定されたパラメータ値を比較する実験を行った．LSH ライブラリ FALCONN は，CCVolti が用いる Cross-Polytope LSH の 1 つの実装である．Cross-Polytope LSH の振舞いをパラメータ値により変更できるという利点があるため，評価実験においてもライブラリ FALCONN を用いる．本節で実施する評価実験は，類似探索の探索時間と再現率という 2 つの観点で行う．

本手法の目的は，クローン検出の利用者が与えた目標再現率を満たし，かつ高速であるための Cross-Polytope LSH に与えるパラメータ値を決定することである．そこで本実験では，本手法で決定したパラメータ値と，表 2.3 に示した FALCONN のデフォルトのパラメータ値との比較実験を行い，類似探索の探索時間と再現率という 2 つの観点で有効性の評価を行う．そして，本手法の有効性を示すために，以下の 2 つの RQ に対して実験結果に基づいて考察を行う．

表 2.1　学習用プロジェクト

プロジェクト	言語	コード ブロック数	類似度 0.9 以上の クローンペア数	行数
Antlr 4.7.1	Java	2,787	3,566	92,976
SNNS 4.2	C	3,113	2,640	133,968
Maven 3.5.4	Java	3,468	3,448	133,238
Ant 1.10.5	Java	5,619	1,785	273,631
zfs-linux 2.19.1	C	6,806	1,119	259,771
HTTPD 2.4.35	C	7,626	1,501	255,468
ArgoUML 0.34	Java	8,696	5,038	391,837
Python 3.7.1	C	9,685	2,223	400,916
heimdal 2.19.1	C	12,083	2,335	549,880
Pig 0.17.0	Java	12,259	16,462	398,130
Tomcat 9.0.12	Java	13,488	9,043	562,549
Jackrabbit 2.16.3	Java	15,591	7,930	617,459
WildFly 14.0.1	Java	19,026	11,394	906,776
PostgreSQL 10.1	C	25,596	12,108	1,314,890
Camel 2.22.0	Java	50,515	508,298	1,953,433
gcc 8.2.0	C	93,104	847,841	4,079,924
OpenJDK 11.28	Java	110,364	53,347	4,766,529
FireFox 59.0.3	C	182,233	92,757	7,046,826
Linux Kernel 4.19	C	363,935	108,932	15,000,647
FreeBSD 11.2	C	379,014	196,714	15,694,482

RQ1　本手法で決定したパラメータ値での再現率は目標再現率を超えているか？

RQ2　FALCONN のデフォルトのパラメータ値での探索時間に対して，本手法で決定したパラメータ値での探索時間は減少しているか？

以降，評価実験の詳細と結果，そこから得られる考察について述べる．

2.4.1　実験対象

本節では，本実験の実験対象プロジェクトと，比較対象とする FALCONN のデフォルトのパラメータ値について述べる．

本実験では，実験対象のプロジェクトとして，過去にコードクローン検出器の評価の実験対象にされたことがある 20 個のプロジェクトを用意した．表 2.1 は学習用プロ

表 2.2　再現率に影響を与えるパラメータ値の入力可能範囲

パラメータ名	値の範囲
区画の分割数 T	$1 \leq T \leq 1024$
ハッシュ関数の数 K	$K = 1, 2, 3$
ハッシュテーブル数 L	$1 \leq L$

表 2.3　FALCONN のデフォルトのパラメータ値

パラメータ名	値
区画分割数 T	$2^{(r-1)}$ ($r = d \bmod \log_2 n$)
ハッシュ関数の数 K	$(\log_2 n - 1)/d$
ハッシュテーブル数 L	10

ジェクトの言語，コードブロック数，類似度が 0.9 以上のクローンペア数，および行数を示す．プロジェクトの順はコードブロック数によって並びかえた．クローン検出の対象とするプロジェクトは，C 言語で記述されたプロジェクトと Java で記述されたプロジェクトがそれぞれ 10 個ずつある．これらは，コードクローンに関する論文の評価実験等で用いられたプロジェクトから収集し，類似度が 0.9 以上のベクトル対集合 U_{all} が 1000 以上あるプロジェクトを選択した [13, 21, 26, 60–65].

これらの学習用プロジェクトに対して，CCVolti を用いてプロジェクトごとにコードクローンを検出し，Cross-Polytope LSH に与えるパラメータごとに再現率と類似探索の探索時間を計測する．クローンペアの基準として類似度の閾値 θ を CCVolti がデフォルトとする 0.9 を用いる．CCVolti が用いる Cross-Polytope LSH のパラメータの内，STEP I で抽出したパラメータ以外のパラメータ値は，ライブラリ FALCONN のデフォルトのパラメータ値に統一した．STEP I で抽出したパラメータには，ラベルとハッシュテーブル数に取りうる値の範囲の組み合わせを総当たりを与える．ライブラリ FALCONN の 3 つのパラメータ T, K, L に入力可能な値は，表 2.2 示す範囲の整数値である．ラベル $label = (K-1)\ln d + \ln T$ に $1 \leq label \leq 20$ の範囲で 1 刻みで値を与えるように，区画の分割数 T とハッシュ関数の数 K に値を与える．ハッシュテーブル数 L には，$1 \leq L \leq 30$ の範囲で 1 刻みで値を与える．

FALCONN のデフォルトのパラメータ値を表 2.3 に示す．FALCONN のデフォルトのパラメータ値は，最近点を高確率で検出できる値を規模に応じて与えられる．d はベクトル集合の各ベクトルの次元数を表し，n はベクトル集合に含まれるベクトルの数を示す．つまり，区画の分割数 T とハッシュ関数の数 K のデフォルトのパラメー

タ値は，ベクトル集合のベクトル数とベクトルの次元数から算出している．この算出方法の理由は開発者によって明示されていないが，ベクトル集合の密度に対して区画の大きさを調整することによって，クエリベクトルに対して最も近いベクトルを高確率で検出するためだと考えられる．また，ハッシュテーブル数 L は 10 に固定されている．ハッシュテーブルが 10 個あれば，限りなく 1 に近い確率で最も近いベクトルを探索できると考えられる．

2.4.2 線形回帰モデル作成

本節では，表 2.1 の実験対象プロジェクトを 20 個の学習用プロジェクトとして，目標再現率 0.9 に対して生成される線形回帰モデルを示す．また，生成した線形回帰モデルを各実験対象プロジェクトに適用して得られるパラメータの値を示す．

表 2.1 の実験対象プロジェクトを 20 個の学習用プロジェクトとして，目標再現率 0.9 に対する線形回帰モデルを作成する．STEP I-B にしたがって，各学習用プロジェクトに対してコードクローン検出を行い，パラメータごとの再現率と類似探索の探索時間を計測し，学習データを作成する．さらに，STEP I-C にしたがって，各学習用プロジェクトに対して目標再現率を超えつつ探索時間が短いパラメータ組を 1 つずつ抽出し，各プロジェクトのコードブロック数を説明変数とし，抽出したラベルを目標変数として線形回帰分析を行い，線形回帰モデルを作成する．ラベルを目標変数，プロジェクトのコードブロック数を説明変数として線形回帰分析すると，回帰係数は 1% 水準で統計的に有意であった．

目標再現率 0.9 に対して生成された線形回帰モデルの回帰係数は 1.87×10^{-7} であり，切片は 9.79 である．作成した線形回帰モデルは，学習データとして 2 つの言語の異なる規模のプロジェクトを用いており，C 言語や Java 言語などの手続き型言語に対して汎用的に利用可能な線形回帰モデルであると考えられる．本手法の評価実験を踏まえた汎用性に関する考察を 2.5.2 節で述べる．

生成された線形回帰モデルに対象プロジェクトのコードブロック数を入力すると，ラベルの値が決定される．ラベルを表す式は $label = (K-1)\ln d + \ln T$ であり，表 2.2 から $d = 1024$,$1 \leq T \leq 1024$ だから，決定されたラベルの値に対して 2 つのパラメータ T, K を算出できる．例えば，$label = 14$ の場合，$K = 2$,$T = 16$ となる．ラベルの値に対して STEP I-C で取り出されるパラメータ組の中で最も小さい L を決定し，探索時間が最短の L を得る．したがって，作成した線形回帰モデルを用いて，対象プロジェクトのための Cross-Polytope LSH に与える 3 つのパラメータ T, K, L を決定できる．

表 2.4　目標再現率 0.9 に対する線形回帰モデルで決定される各プロジェクトのパラメータ値

プロジェクト	T	K	L
Antlr 4.7.1	1024	1	3
SNNS 4.2	1024	1	3
Maven 3.5.4	1024	1	3
Ant 1.10.5	512	1	3
zfs-linux 2.19.1	1024	1	3
HTTPD 2.4.35	1024	1	3
ArgoUML 0.34	1024	1	3
Python 3.7.1	1024	1	3
heimdal 2.19.1	1024	1	3
Pig 0.17.0	1024	1	3
Tomcat 9.0.12	1024	1	3
Jackrabbit 2.16.3	1024	1	3
WildFly 14.0.1	1024	1	3
PostgreSQL 10.1	1024	1	3
Camel 2.22.0	1024	1	3
gcc 8.2.0	128	1	3
OpenJDK 11.28	1024	1	3
FireFox 59.0.3	1024	1	3
Linux Kernel 4.19	16	2	5
FreeBSD 11.2	16	2	5

2.4.3　実験方法

本実験では，本手法の有効性を示すために，0.8 以上 1 未満の 20 個の目標再現率に対して表 2.1 の 20 個のプロジェクトを用いて 10 分割交差検証を実施する．20 個のプロジェクトを 18 個の学習用プロジェクトと 2 個のコードクローン検出対象プロジェクトに分割する．18 個の学習用プロジェクトを用いて，0.8 以上 1 未満の 0.01 刻みの 20 個の目標再現率の各値に対して STEP I を実行し，各目標再現率に対する線形回帰モデルを作成する．作成した線形回帰モデル群と 2 個のコードクローン検出対象プロジェクトに対して STEP II を実行しパラメータ値の組を決定する．決定したパラメータを与えた Cross-Polytope LSH を用いて，CCVolti によってコードクローン検出を行い，再現率と探索時間を計測する．類似探索の再現率の計測は，2.3.1 節で述べた方

法と同様の手順で行う．よって，適合率は常に 1 となる．10 分割交差検証のために，学習用プロジェクトとコードクローン検出対象プロジェクトを入れ替え，各目標再現率に対してすべての実験対象プロジェクトのためのパラメータ値を決定し，CCVolti によるコードクローン検出の再現率と探索時間を計測する．

実験環境は，CPU Intel Xeon 2.80GHz，メモリ 32.0GB，OS Windows 10 64bit. Java 仮想マシンのヒープ領域 15GB とした．クローンペアとするコサイン類似度の閾値 θ は，コードクローン検出法 CCVolti がデフォルトとする 0.9 とした．また，CCVolti が用いる Cross-Polytope LSH ライブラリ FALCONN に与えるパラメータの内，本手法の STEP I-A で抽出したパラメータ以外のパラメータ値は，FALCONN のデフォルトのパラメータ値に統一した．

2.4.4 実験結果

実験結果を表 2.5, 図 2.3, 図 2.4, 図 2.5, 図 2.6, 付録 2 に示す．表 2.5 は, FALCONN のデフォルトのパラメータ値での再現率と探索時間を計測した結果を示す．図 2.3 は，目標再現率毎に本手法で決定したパラメータ値と，FALCONN のデフォルトのパラメータ値に対して，再現率の比較を箱ひげ図で示した．このグラフの横軸は目標再現率の値を表し，縦軸はその目標再現率のときの本手法で決定したパラメータ値での実験を行ったときの再現率を表す．ただし，最も右の列は FALCONN のデフォルトのパラメータ値での結果を表している．図 2.4 は，本手法で決定したパラメータ値での探索時間に関して，FALCONN のデフォルトのパラメータ値での探索時間と比較した増減率を目標再現率毎に箱ひげ図で示した．この箱ひげ図の横軸は目標再現率の値を表し，縦軸はその目標再現率に対して本手法で決定したパラメータ値で実験を行ったときの類似探索の時間に関して FALCONN のデフォルトのパラメータ値での探索時間と比較した増減率を表している．図 2.5, 図 2.6 は，本手法で決定したパラメータ値での探索時間に関して，目標再現率の変化に伴うプロジェクト毎の探索時間の増減率をパラメータ毎に調査した結果である．このグラフの横軸は目標再現率の値を表し，縦軸はその目標再現率のときの本手法で決定したパラメータ値でそのプロジェクトの探索時間の増減率を表しており，折れ線グラフは各プロジェクトの増減率の変化を表している．付録 2 は，プロジェクトごとに各目標再現率と，FALCONN のデフォルトのパラメータ値に対して，CCVolti の探索時間を計測した結果を示す．

RQ1

図 2.3 から，75% 以上のプロジェクトにおいてどの目標再現率についても再現率が目標再現率を超えていることが分かる．これにより，あるプロジェクトに対して CCVolti によってコードクローン検出する際，本手法は多くの場合で再現率の目標値を満たすことができるといえる．

しかし，いくつかのプロジェクトで再現率が目標再現率を下回る場合を確認し，ま

表 2.5　FALCONN のデフォルトのパラメータ値での実験結果

プロジェクト名	再現率	探索時間 [ms]
Antlr	1.000	1,411
SNNS	0.998	1,566
Maven	0.999	1,696
Ant	0.999	2,737
zfs-linux	0.990	3,363
HTTPD	1.000	3,806
ArgoUML	0.997	6,634
Python	0.996	7,171
heimdal	0.997	9,197
Pig	0.999	11,010
Tomcat	0.997	10,514
Jackrabbit	0.994	12,167
WildFly	0.991	15,457
PostgreSQL	0.995	20,011
Camel	0.986	2,158,552
gcc	0.996	5,755,016
OpenJDK	0.983	93,318
FireFox	0.976	183,147
Linux Kernel	0.974	480,423
FreeBSD	0.980	1,054,177

た，目標再現率 0.9 以下を与えているにも関わらず，再現率が 0.95 付近になる場合を確認した．再現率にばらつきが起こる原因は，クローンセット (互いにクローンペアとなるコードクローンの集合) 内のコード片の数の平均に差があることや，閾値に近い類似度であるクローンペアの数が多いことだと考える．また，FALCONN のデフォルトのパラメータ値での再現率はすべて 0.97 以上である．本手法で決定したパラメータ値が FALCONN のデフォルトのパラメータ値と同程度の再現率を得るためには，目標再現率を 0.98 以上にする必要がある．

RQ1 の答え

多くの場合で本手法は再現率の目標値を満たしている．ただし，再現率が目標再現率を下回るプロジェクトがいくつかあることを確認した．

RQ2

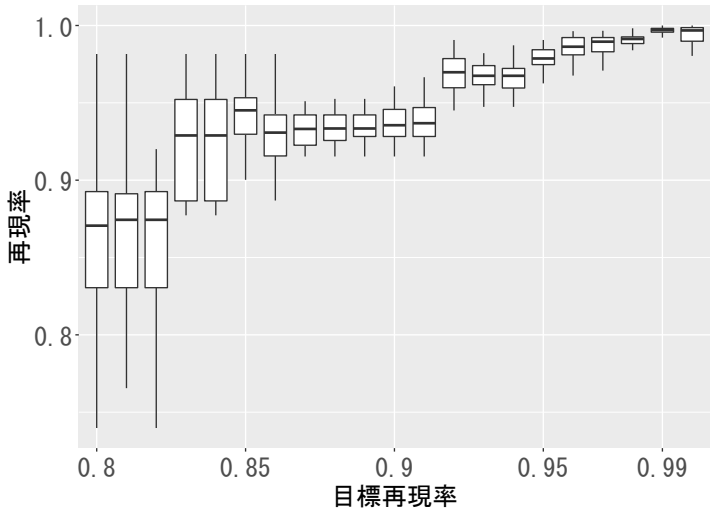


図 2.3　再現率比較

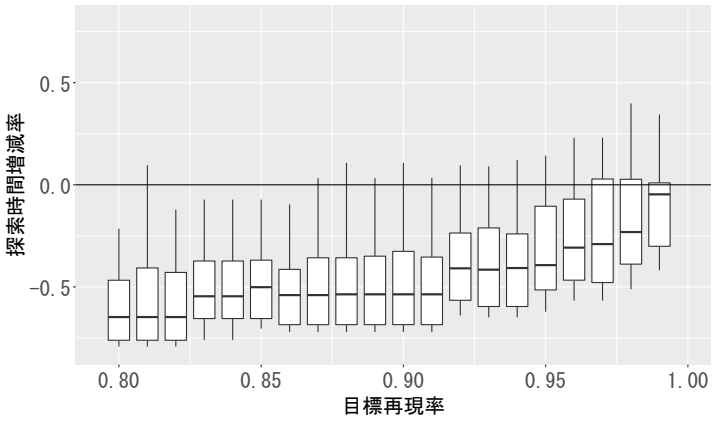


図 2.4　目標再現率毎の検出時間増減率

図 2.4 から，目標再現率が 0.96 以下では，16 プロジェクトが FALCONN のデフォルトのパラメータ値より高速であり，目標再現率が 0.98 以上の場合でも，14 プロジェクトで FALCONN のデフォルトのパラメータ値より高速であることが分かる．さらに，目標再現率が 0.91 以下の場合，18 プロジェクトに関して，デフォルトの探索時間からの増減率が-0.3 を下回っている．このように，多くの場合で FALCONN のデフォルトのパラメータ値より高速である．これにより，あるプロジェクトに対して CCVolti によりコードクローン検出する際，他の OSS を学習して生成した回帰モデルを用いて高速なパラメータを選択できるといえる．

また，図 2.5 と図 2.6 から，多くのプロジェクトに関して，目標再現率を下げることによって探索時間削減できていること分かる．特に，gcc と Camel を除く 18 プロジェクトにおいて，目標再現率 0.8 での探索時間は，目標再現率 0.99 のときの探索時

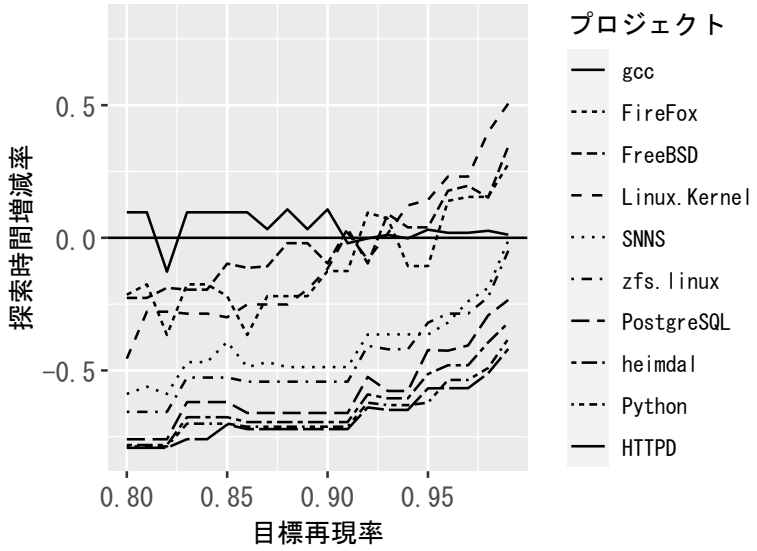


図 2.5　プロジェクト毎の検出時間（C プロジェクト）

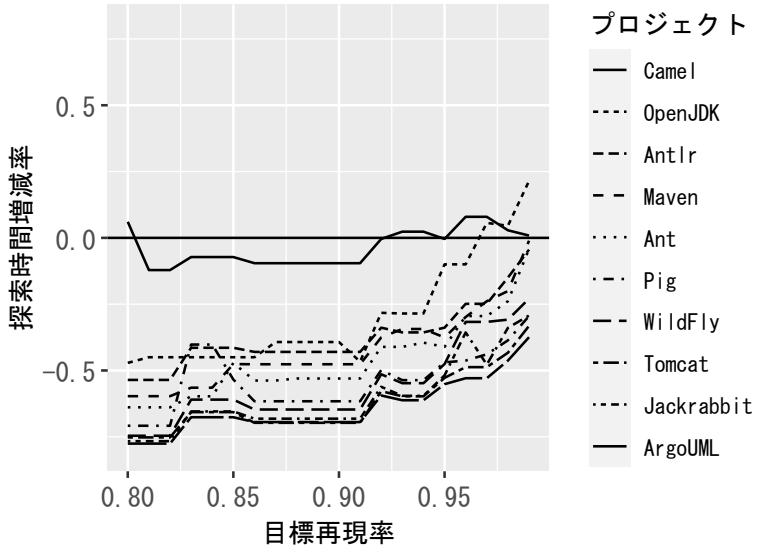


図 2.6　プロジェクト毎の検出時間（Java プロジェクト）

間からおよそ半減できている．これにより，CCVolti の利用者が速度を優先したい場合，低めの目標再現率を設定することで，CCVolti の利用者が許容する再現率を満たしかつ高速化できる．

図 2.5 と図 2.6，2.3.4 節の表 2.1 から，クローンペア数が多いプロジェクトであるほど，FALCONN のデフォルトのパラメータ値との探索時間の増減率が変化しないこ

とがわかる．特に，gcc と Camel の 2 つのプロジェクトに関しては，目標再現率 0.80 の場合に FALCONN のデフォルトのパラメータ値と比べて探索時間が増加している．この原因について考察するために，2.2.3 節で述べた CCVolti の検出の STEP 4 におけるクローンペアの検出の手順を，クローンペアの類似探索とフィルタリングに分割する．クローンペアのフィルタリングでは，クローンペアの重複を排除したり，被覆関係のクローンペアを排除する．FALCONN のデフォルトのパラメータ値での，クローンペアの類似探索とフィルタリングに分割してそれぞれの時間を計測した結果を表 2.6 に示す．FALCONN のデフォルトのパラメータ値の場合，クローンペア数が最も少ない zfs-linux では，フィルタリング時間が類似探索の約 0.04 倍であるのに対し，gcc では 81 倍，Camel では 56 倍もの時間をフィルタリングにかけている．クローンペアが多い場合，フィルタリングの時間が支配的になり，類似探索の高速化だけでは検出時間全体の時間短縮ができない．今後は，クローンペアのフィルタリングの改善が必要だと考えられる．

RQ2 の答え

多くの場合で本手法は FALCONN のデフォルトのパラメータ値より高速である．特に，gcc と Camel を除く 18 プロジェクトにおいて，目標再現率 0.8 での探索時間は，目標再現率 0.99 での探索時間からおよそ半減している．

2.5 考察

本節では，本手法を適用した CCVolti の有用性と，本手法における妥当性への脅威について考察する．

2.5.1 本手法を適用した CCVolti の有用性

本手法を適用した CCVolti の有用性について考察する．本実験では，CCVolti の再現率，適合率，F 値を計測していないが，類似探索の再現率を低く設定したときの CCVolti の精度について CCVolti の評価実験の結果に基づいて，類似探索の再現率が 0.8 のときの CCVolti の検出結果を推定し議論する．CCVolti の再現率とは，正解集合とするコードクローンに対して実際に検出された割合を指す．適合率とは，検出結果に対して正しかったコードクローンの割合を指す．F 値とは，再現率と適合率の調和平均によって表される値である．

類似探索の再現率に対してクローン検出の適合率が一定であると仮定するとき，クローン検出の再現率と類似探索の再現率の関係を図 2.7 に示す．類似探索の再現率が r のとき，CCVolti が検出したクローンペアの数は r 倍となる．さらに適合率が一定である仮定から，CCVolti が検出した正解クローンの数も r 倍となるため，クローン

表 2.6　類似探索時間とクローンペアのフィルタリング時間の比較

プロジェクト名	類似探索 [ms]	フィルタリング [ms]	フィルタリング / 類似探索
Antlr	1,188	223	0.188
SNNS	1,386	180	0.13
Maven	1,484	212	0.143
Ant	2,567	169	0.066
zfs-linux	3,223	141	0.044
HTTPD	3,634	172	0.047
ArgoUML	6,302	332	0.053
Python	6,992	180	0.026
heimdal	9,025	172	0.019
Pig	9,049	1,961	0.217
Tomcat	9,690	824	0.085
Jackrabbit	11,444	723	0.063
WildFly	13,836	1,622	0.117
PostgreSQL	18,604	1,407	0.076
Camel	37,529	2,121,023	56.517
gcc	69,973	5,685,043	81.246
OpenJDK	79,453	13,865	0.175
FireFox	133,360	49,787	0.373
Linux Kernel	264,757	215,666	0.815
FreeBSD	277,871	776,306	2.794

検出の再現率も r 倍となる．例えば，類似探索の再現率が 1 のときクローン検出の再現率が 0.8 とすると，類似探索の再現率が 0.8 のときの CCVolti の再現率は 0.56 となる．したがって，類似探索の再現率が r のとき CCVolti の再現率は，類似探索の再現率が 1 の場合での再現率の r 倍となる．

表 2.7 は，CCVolti の評価実験の結果と，推定した類似探索の再現率が 0.8 のときの CCVolti の検出結果を示す [21]．表 2.7 から，クローン検出の再現率と F 値に関して，類似探索の再現率が 0.8 のときの推定した結果が，関数クローン検出法，CCFinder，粗粒度クローン検出法のいずれよりも優れていることが分かる．って，類似探索の再現率を 0.8 まで下げたとしても，既存手法である関数クローン検出法と CCFinder と同程度の精度になる．

実際には，類似探索の再現率が低下した場合，類似度の低いクローンペアが検出さ

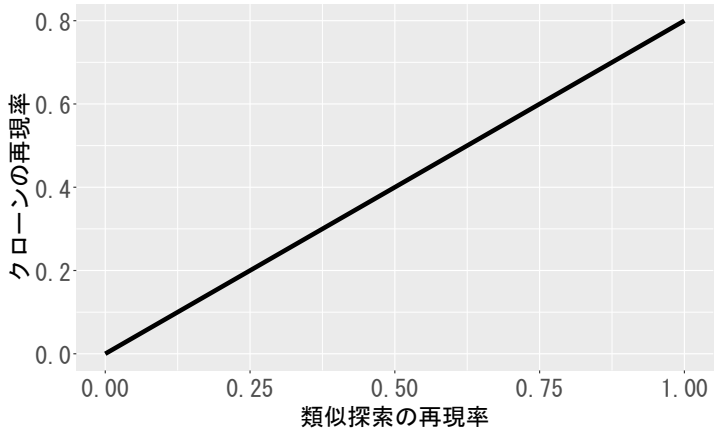


図 2.7　クローン検出の適合率が一定であるときの類似探索の再現率とクローン検出の再現率の関係

れなくなる可能性が高い．これらのクローンペアは，真のクローンペアでない場合が多く，適合率を下げる要因である可能性が高い．そのため，類似探索の再現率の目標値を意図的に下げることで，検出されるコードクローンの再現率は減少するものの，適合率は向上し，F 値は大きく変化しない可能性がある．

本手法の実験結果と，本手法を適用した CCVolti の精度の推定から，目標再現率を決定するときに以下のような指標が得られる．目標再現率を 0.98 以上とすると，FALCONN のデフォルトのパラメータ値と同程度の再現率を得られるパラメータ値の組が決定されることが考えられる．目標再現率 0.98 以上での実験結果では，70% のプロジェクトでデフォルトのパラメータより類似探索の時間を短縮できた．目標再現率を 0.8 とすると，既存手法と同程度の再現率や F 値となり，デフォルトのパラメータ値の半分の探索時間で検出できるパラメータ値の組が決定されることが考えられる．

本手法による高速化は，コードクローン検出の適用頻度の向上を可能にする．たとえば，長時間の実行時間を要するために夜間に限定されていたコードクローン検出を，目標再現率を下げて高速に実行することで，日中の作業時間中にも実行可能となる．また，開発フローにおける日々の CI/CD パイプラインにコードクローン検出を組み込む場合に，1 分かかっていた処理を 30 秒以内に短縮できる可能性があり，継続的な開発環境との統合が現実的となる．これにより，開発者はより迅速にフィードバックを得ることができ，修正の即時対応やコードクローンの早期発見を可能にする．

2.5.2　妥当性の脅威

本手法によって作成される線形回帰モデルの信頼性について考察する．本手法の評価実験では，10 分割交差検証を行い，目標再現率ごとに 10 個の線形回帰モデルを作成した．10 個の線形回帰モデルの差異について調査した．目標再現率ごとのモデルご

表 2.7　検出精度の比較

検出手法	クローン検出の適合率	クローン検出の再現率	F 値
CCVolti	0.68	0.70	0.69
関数クローン検出法	0.67	0.47	0.55
CCFinder	0.57	0.52	0.54
粗粒度クローン検出法	0.91	0.24	0.38
CCVolti (類似探索の再現率 0.8 の場合の推定値)	0.68	0.56	0.63

とに，回帰係数の平均と分散，切片の平均と分散を計算した．目標再現率毎に，10 分割交差検証に使用した 10 個の線形回帰モデルの回帰係数と切片の平均を，20 個のプロジェクトを学習して作成した線形回帰モデルの回帰係数と切片との差分を計算した．すべての目標再現率に対して，回帰係数の平均の差分は 10^{-9} 以下で，分散は 10^{-14} 以下だった．また切片の平均の差分は 10^{-2} 以下，分散は 10^{-1} 以下だった．どの目標再現率に対しても，10 分割交差検証で作成した線形回帰モデルは，20 個のプロジェクトを学習して作成した線形回帰モデルと大きな差異がないと言える．

次に，本手法の汎用性について考察する．本手法は C 言語と Java 言語のプロジェクトを用いて線形回帰モデルを作成する．評価実験では，10 分割交差検証で C 言語や Java 言語のプロジェクトに適したパラメータを決定し，多くの場合で目標再現率を超え，多くの場合でデフォルトのパラメータ値より高速なパラメータ値を決定した．しかし，HTML や Cobol など，C 言語や Java 言語と特徴が大きく異なる言語で記述されたプロジェクトに対して，本手法で作成した線形回帰モデルを適用して，コードブロック数から正しくラベルの値を決定できないと考えられる．したがって，本手法の利用者は対象言語と似たプロジェクトを学習データとして本手法の STEP I に基づいて線形回帰モデルを作成する必要がある．

最後に，本手法の実行時間について考察する．本手法は，学習プロセスと適用プロセスの 2 段階から構成される．学習プロセスでは，複数のプロジェクトに対して異なるパラメータ値で類似探索を実行する必要があるため，多くの時間を要する．実際に，本実験における学習データの作成には約 2 週間を要した．しかし，作成された回帰モデル群は汎用性を有しており，CCVolti の利用者は学習プロセスを再実行する必要はなく，適用プロセスのみでパラメータ決定が可能である．適用プロセスは，回帰モデルの選択とパラメータ決定を機械的に行うだけであり，処理時間は非常に小さい．さらに，利用者が与える目標再現率とソースコードに基づいてパラメータを自動的に決定できることから，CCVolti の処理フローに本手法の適用プロセスを組み込むことで，

利用者はLSHのパラメータを意識することなく，類似探索の目標再現率をハイパーパラメータとして指定するだけで，効率的なコードクローン検出が可能となる．

2.6 まとめと今後の課題

本章では，コードクローン検出ツールCCVoltiが用いるCross-PolytopeLSHに与えるパラメータ値を，クローン検出の利用者が与えた目標再現率を満たし，かつ高速になる値に決定する方法を提案した．そして，本手法をCCVoltiが用いるCross-PolytopeLSHに適用し，異なる規模の20のプロジェクトに対してコードクローン検出を行った．その結果，本手法で決定されたパラメータ値は多くの場合でデフォルトのパラメータ値より高速であり，多くの場合で再現率が目標値を超えることを確認した．また，特に目標再現率が0.91以下の場合，75%のプロジェクトに関して本手法で決定されたパラメータ値はデフォルトのパラメータ値と比べて探索時間が30%以上下回っており，目標再現率を下げることによる高速化を確認した．これにより，CCVoltiの利用者は目標再現率を下げることで高速なパラメータを選択することで，大規模なプロジェクトに対して頻繁にコードクローン検出し，修正の即時対応やコードクローンの早期発見，追跡手法への応用を可能とする．

今後の課題として，gccやcamelなどのクローンペアが多いプロジェクトの場合はクローンペアのフィルタリング時間が支配的となり，LSHの高速化だけでは検出時間全体の時間短縮ができないため，クローンペアのフィルタリングの速度改善が挙げられる．また，本手法による検出速度を活かした応用として，本手法を適用したCross-PolytopeLSHを用いたCCVoltiと他のコードクローン検出法に対して検出精度や検出時間を比較することが挙げられる．さらに，限られた時間の中でCCVoltiを用いたクローン検出を行うCCVolti利用者のために，本手法と同様に目標検出時間を与え，検出時間が目標値を上回りつつ，できるだけ再現率が高く維持できるようなパラメータ値を決定する方法への利用を考えることが挙げられる．最後に，CCVoltiを用いて検出可能なタイプ3のコードクローンに対するリファクタリングは，一部の行をコードクローンの範囲外へ移動させる必要があるなど，処理が複雑な場合が多く，タイプ3のコードクローンに対する自動リファクタリング手法の検討が今後の課題であると考えられる．

第 3 章

コードクローン変更管理システムの開発と改善

3.1 まえがき

ソフトウェア開発において，リファクタリングやバグ修正などのコード変更に伴うコードクローンに対する変更・削除は頻繁に発生する．しかし，開発者が他のコードクローンの存在に気付かず，コードクローンに対して一貫性のない変更を加えた場合，同様に変更すべきコード片に潜在的な障害が残存する可能性がある．実際に，コードクローンに対する一貫性のない変更は多く確認されており，その中にはバグ修正が含まれる場合があると指摘されている [35–37].

開発者は欠陥を修正する際には，修正対象のコード片だけでなく，そのコード片を含むコードクローンの集合（クローンセット）内のコードクローンに一貫性のある修正を同時に施す必要がある [32]．クローンセットの一貫した修正を支援するために，1.2.3 節に述べたように，コードクローン追跡やコードクローン同期に関する手法が多数提案されている [16].

山中らはコードクローンを追跡し，コードクローンの変更情報を開発者に通知するシステム，コードクローン変更管理システムを開発した [47,48]．山中らが提案した従来システムは，字句ベースのコードクローン検出器 CCFinderX [20] を用いて，コードクローンを検出する．次に，2 つのリビジョン間のクローンセット内のコードクローンの変更有無に基づいて，クローンセットを（Changed, Deleted, New, Stable）の 4 つに分類する．従来システムは，コードクローンの変更情報について定期的に検出し，変更に関する情報を電子メールで開発者に送信する．

山中らは，一部のコードクローンが変更されたクローンセット（Changed Clone Set）は一貫性のない変更を含む可能性があり，旧リビジョンに存在せず，新リビジョンにのみ存在するクローンセット（New Clone Set）はリファクタリング対象となる可能性があるとしている．しかし，適用実験にて検出された Changed Clone Set のう

ち修正されたクローンセットはなく，119 個の New Clone Set のうち，11 個のみ集約された [47]．開発者は，出力された多くの Changed Clone Set や New Clone Set の中から，一貫性のない変更が含まれるクローンセットやリファクタリング対象であるクローンセットを手作業で識別しなければならない．ソフトウェアの規模が大きくなるにつれて，検出されるクローンセットの数は膨大になるため，人手で結果を確認して保守作業を行うのは困難である．

この問題を軽減するために，我々は従来のコードクローン変更管理システムを改善した Clone Notifier を提案する．一貫性のない変更など特に確認すべき箇所を開発者に示すために，以下の 4 点を改善した．

- コードクローン検出器 CCVolti [66] と SourcererCC [67] の追加
- クローンセットの定義の変更
- コードクローンの追跡方法の改善 [17]
- クローンセットの詳細な分類

一部変更されたコードクローンを検出可能な CCVolti および SourcererCC を導入することで，一貫性のない変更が行われたクローンセットの検出を可能とした．また，従来のコードクローン変更管理システムとは異なり，構文的に一致しないコードクローンを検出するため，類似度の低いコード片がクローンセットに含まれることを防ぐ目的で，クローンセットの定義を再検討し，追跡方法の改善を行った．さらに，変更されたクローンセットに対して，一貫性のある変更の有無などの詳細な分類情報を付加する機能を導入した．これらの改善により，開発者が特に注意すべき一貫性のない変更を効果的に通知することが可能となり，大規模ソフトウェアに対する検出結果の確認コストを大幅に削減した．

本章では，開発者がクローンセットの不具合を一貫して修正できるように，Clone Notifier が開発者をどのようにサポートするかを示す．オープンソースソフトウェア PostgreSQL[*1] を用いて Clone Notifier の利用シナリオを示す．PostgreSQL の 2018 年 6 月 23 日から 2019 年 6 月 22 日の 1 年間のコミット履歴に対し，コミットごとに Clone Notifier を適用しコードクローンの変更を検出した．Clone Notifier は，全 2,152 コミットのうち 160 件で，クローンセットに対する一貫性のない変更を検出した．また，30,382 個の変更されたクローンセットのうち，299 個のクローンセットが一貫性のない変更として分類された．これにより，従来のコードクローン変更管理システムと比べて，開発者の結果確認コストを大幅に削減した．一貫性のない変更のクローンセットの検出結果から，目視確認にて発見した客観的に明らかな変更忘れであると考えられる 3 つの事例を示す．

以降，3.2 節では，提案システム Clone Notifier のクローンセット追跡・分類手法

[*1] `https://github.com/postgres/postgres`

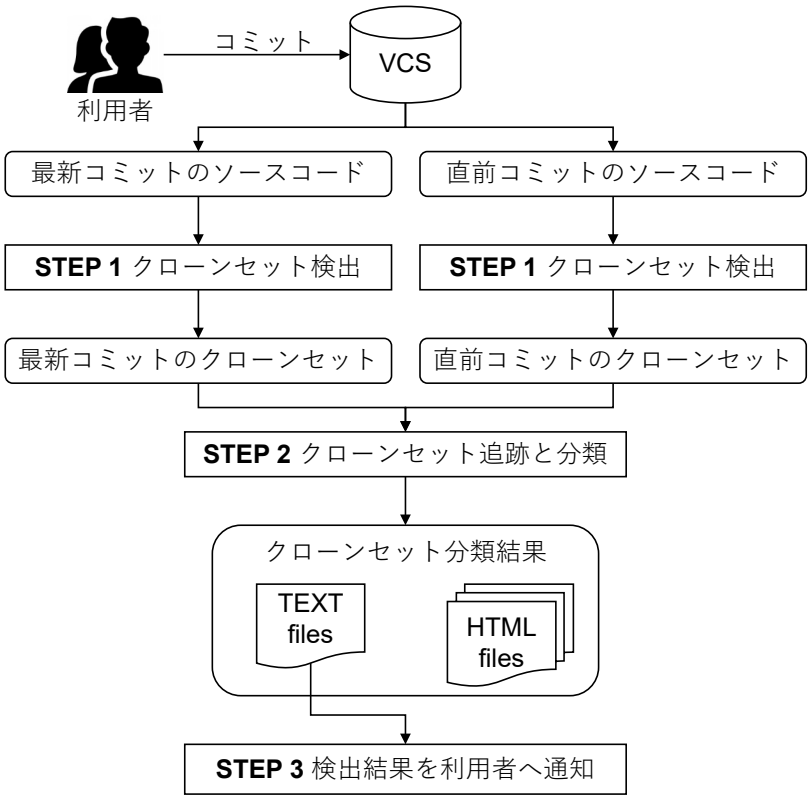


図 3.1　Clone Notifier の概要

について述べる．3.3 節では，Clone Notifier の利用シナリオと，一貫性のない変更のクローンセット検出事例を示す．3.4 節では，検出された一貫性のない変更の事例をふまえた，本手法の有効性の範囲と妥当性について考察する．最後に 3.5 節では，まとめと今後の課題について述べる．

3.2　提案システム Clone Notifier

Clone Notifier は，2 つのリビジョン間のソースコードに対してコードクローンの変更情報を分析し，分析結果の概要を開発者に通知するシステムである．Clone Notifier は 3 ステップで実行される．

1. 各リビジョンでクローンセットを検出.
2. 2 つのリビジョン間のクローン追跡と分類.
3. クローンの変更情報を HTML 形式で生成し，実行結果を開発者に通知.

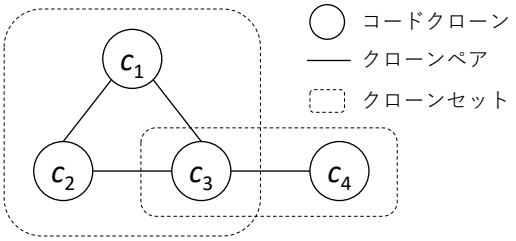


図 3.2　意味的なコードクローン検出器におけるクローンセットの例

3.2.1　クローンセット検出

クローンセットとは，すべてのコードクローンの対が互いに一致または類似しているコードクローンの集合である．Clone Notifier は，3 種類のコードクローン検出器 “SourcererCC [67]”, “CCFinderX [20]”, “CCVolti [66]” を用いてソースコード内のクローンペアを検出し，クローンペアからクローンセットを構築する．Clone Notifier がコードクローン検出可能な言語はコードクローン検出器に依存する．

Clone Notifier におけるクローンセットの定義は，すべてのコードクローンのペアがクローンペアである集合とする．構文的コードクローン検出器である CCFinderX が 2 つのクローンペア (c_1, c_2), (c_2, c_3) を検出した場合，その関係は推移的特性を保持するため，クローン c_1 と c_3 は構文的に一致する．CCFinderX を用いて検出されたクローンペアは，すべて連結してクローンセットを構築可能である．一方で，意味的なコードクローン検出器である SourcererCC と CCVolti は 2 つのクローンペア (c_1, c_2), (c_2, c_3) を検出した場合，コード片のペア (c_1, c_3) をクローンペアとして検出しないことがある．コード片のペア (c_1, c_3) は，クローン検出器が検出できなかったクローンペアであり本来はコードクローンとして管理すべき対象である可能性や，その時点では類似性が低くクローンペアではない可能性など考えられる．類似性が低いコード片の組は一貫した修正が不必要である可能性が高いと考えたため，Clone Notifier はすべてのコードクローンのペアがクローンペアである集合をクローンセットと定義した．

Clone Notifier は，利用者が意味的なコードクローン検出器を選択する場合，クローンセット内のすべてのコード片の組がクローンペアとなるように，クローンセットを構築する．すべてのコードクローンのペアがクローンペアであるようなコードクローンの集合を効率よく探索するため，これをグラフ理論における極大クリーク問題と捉え，極大クリーク問題を解くためのライブラリ JGraphT[*2] を使用した．クリークとは，無向グラフの頂点の部分集合であり，クリーク内の 2 つの異なる頂点間に辺が存在するような部分完全グラフである．極大クリークとは，他のクリークの部分集合で

[*2] `https://jgrapht.org/`

表 3.1　クローンセットの分類

分類	説明
Stable Clone Set	すべてのコード片が変更されていないクローンセット
New Clone Set	新リビジョンにのみ存在するクローンセット
Deleted Clone Set	旧リビジョンにのみ存在するクローンセット
Changed clone sets	編集または追加または削除されたコード片を含むクローンセット

表 3.2　変更されたクローンセット (**Changed Clone Set**) のラベル

ラベル	説明
Add	追加されたコード片を含む
Subtract	削除されたコード片を含む
Shift	別のクローンセットから移動してきたコードクローンを含む
Consistent	全てのコード片が編集されている
Inconsistent	編集されたコード片と編集されていないコード片が同時に存在する

はないクリークである．

Clone Notifier におけるクローンセットの定義の例を図 3.2 に示す．コードクローン c_1, c_2, c_3, c_4 が存在して，クローンペア (c_1, c_2), (c_2, c_3), (c_3, c_1), (c_3, c_4) が検出された場合，Clone Notifier は 2 つのクローンセット (c_1, c_2, c_3), (c_3, c_4) を構築する．JGrapht を使ってクローンペア集合の極大クリークを構築すると，一貫した修正が必要な可能性があるコードクローンを過不足なくクローンセットに含めることができる．

3.2.2　クローンセットの追跡と分類

Clone Notifier はクローンセットを検出した後，2 つのリビジョンで取得したクローンセットの位置情報を用いて，旧リビジョンのクローンセットから新リビジョンのクローンセットを追跡する．2 つのリビジョン間のコード片を追跡するために，Miryung Kim らの Location Overlapping 関数を参考に，コードクローンにおける行の重複度を利用した [17]．行の重複度 $LO(l1, l2)$ は，位置情報 $l1$ が位置情報 $l2$ に対する重複行の割合を示す．各リビジョンの同名ファイルの行単位の差分を取得し，追加または削除された行を除いて各コードクローンの開始行と終了行を取得し，式 3.1 を利用してコードクローンの行の重複度 $LO(l1, l2)$ を計測する．

$$LO(l1, l2) = \frac{min(n_e, o_e) - max(n_s, o_s)}{n_e - n_s}, (0 \leq LO(l1, l2) \leq 1) \tag{3.1}$$

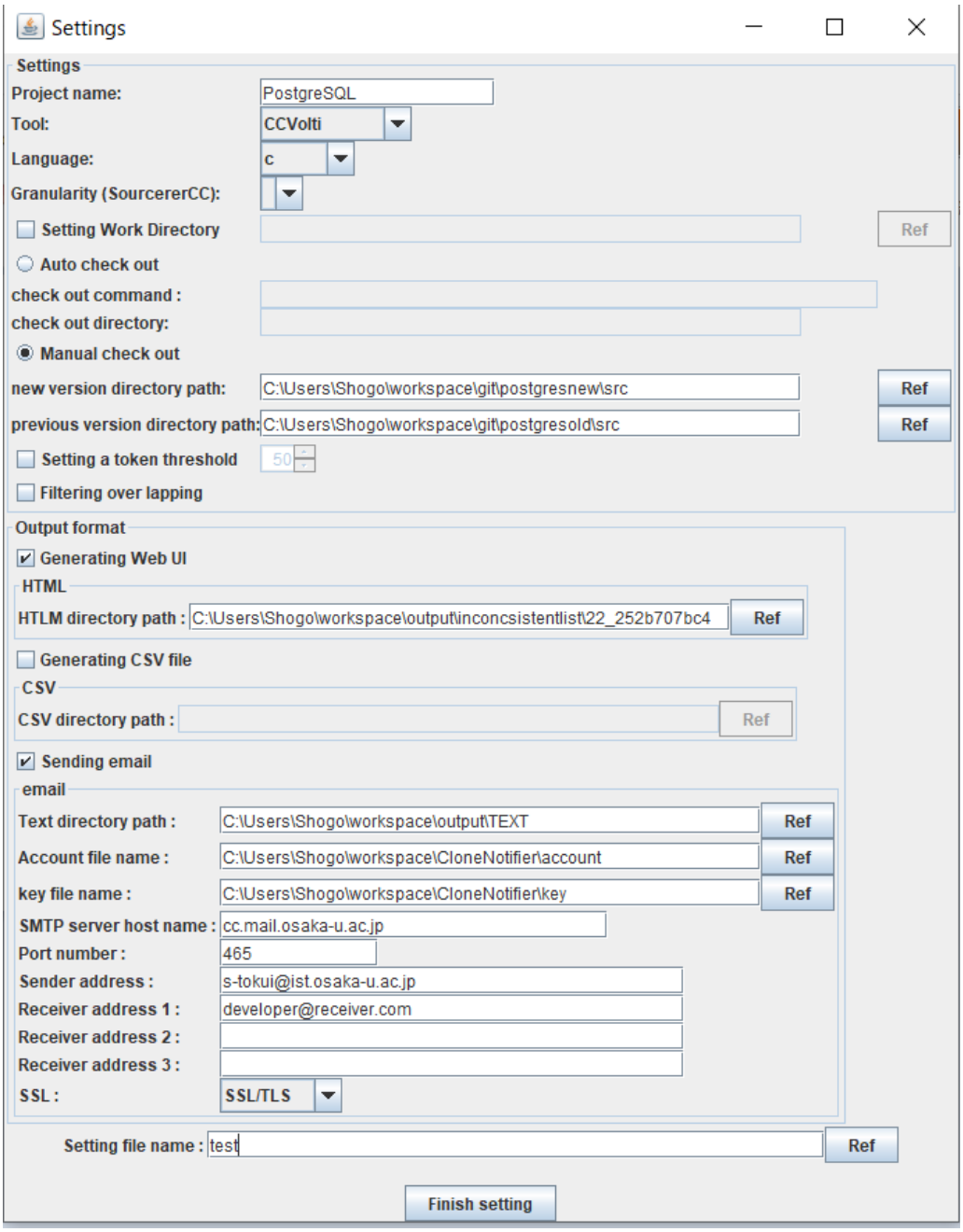


図 3.3　Clone Notifier 設定変更を行うウィンドウ

ここで，旧リビジョンのコードクローンの位置 l_1 の (開始行, 終了行) は (o_s, o_e)，新リビジョンのコードクローンの位置 l_2 の (開始行, 終了行) は (n_s, n_e) とする．各リビジョンのコードクローン (l_1, l_2) の行の重複度が 70% 以上のとき，新旧クローンとして関係付ける．コードクローンの変化を定量的に計測し，追跡の精度を向上させた．

山中らが提案したコードクローン変更管理システムは，追跡したクローンセットを

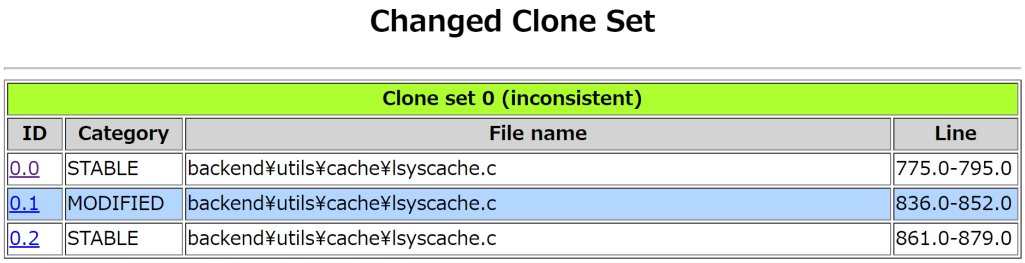
Project information	
File information	
Total files	1086
Add files	0
Deleted files	0
files contaning clones	595
Clone set categories information	
Total clone sets	2745
Stable clone sets	2744
Changed clone sets	1
New clone sets	0
Deleted clone sets	0
Code clone information	
Total code clones	9470
Stable code clones	9469
Modified code clones	1
Moved code clones	0
Added code clones	0
Deleted code clones	0
Deleted and modified code clones	0

図 3.4　トップページ

図 3.1 に示す分類基準で Stable, New, Deleted, Changed の 4 つに分類した [48]．本研究ではさらに，Changed Clone Set に含まれるコードクローンの編集の種類に応じて，*add*, *subtract*, *shift*, *consistent*, *inconsistent* のラベルをクローンセットに付与する [17]．これにより，Clone Notifier はクローンセットに関するより詳細な変更情報を開発者に通知する．

3.2.3　通知と分析結果

Clone Notifier はクローンセットの変更情報の分析結果を HTML 形式と CSV 形式で出力し，開発者に電子メールを送信する．開発者は事前に電子メール情報を設定することで，Clone Notifier から分析結果の概要や分析結果へのファイルリンクが記された電子メールを受け取れる．分析結果の概要には，追加または削除されたファイルやコードクローンを含むファイル数，クローンセットやコードクローンの数が含まれ



Changed Clone Set

Clone set 0 (inconsistent)			
ID	Category	File name	Line
0.0	STABLE	backend¥utils¥cache¥lsyscache.c	775.0-795.0
0.1	MODIFIED	backend¥utils¥cache¥lsyscache.c	836.0-852.0
0.2	STABLE	backend¥utils¥cache¥lsyscache.c	861.0-879.0

図 3.5　クローンセット分類結果ページ

る．開発者は Clone Notifier を定期自動実行することで，一定の期間ごとのコードクローン変更情報を検出して分析することができる．

Clone Notifier の分析結果の確認方法について述べる．図 3.4 は HTML 形式の分析結果ファイルのトップページであり，分析結果の概要を確認できる．クローンセットの詳細情報は表 3.1 の分類ごとに分かれており，例えば，図 3.4 の ‘Changed clone set’ をクリックすると，Changed Clone Set の詳細情報が記されたページ (図 3.5) を閲覧できる．クローンセットの詳細情報のページでは，クローンセットに含まれるコードクローンの位置情報や，コードクローンの変更有無（Stable, Modified, Add, Deleted）を調べることができる．特に，クローンセットは表 3.2 のラベルごとにソートされており，*inconsistent* クローンセットはリストの最上位に表示される．コードクローンの ID をクリックすると，コードクローンのソースコードを確認できる．図 3.6，図 3.7，図 3.8 に示すように，ソースコードページは 2 つのリビジョンの差分に基づいて記述されており，開発者はコードクローンがどのように変更されたかを容易に確認できる．

3.3　ユースケースシナリオ

本節では，一貫性のない変更が行われたクローンセットを検出するための Clone Notifier の利用シナリオを示す．我々は実験対象である PostgreSQL にて，一貫性のない変更を検出し，修正すべきコードクローンを 3 つ発見した．

3.3.1　利用シナリオ

開発者は，事前に GUI を通じて Clone Notifier の設定を行う．Clone Notifier をダウンロード後，setting.jar を実行して設定変更を行うウィンドウ（図 3.3）を開く．GUI を利用して，コードクローン検出器や対象言語，対象ソースコードのディレクトリパス，電子メール情報などの実行環境を記述し，設定ファイルを生成する．

設定ファイルを指定して Clone Notifier を実行する．実行完了後，開発者は Clone

Stable Code Clone

File: src/backend/executor/execMain.c

```
2097    {
2098            TupleDesc       old_tupdesc = RelationGetDescr(rel);
2099            AttrNumber *map;
2100
2101            rel = resultRelInfo->ri_PartitionRoot;
2102            tupdesc = RelationGetDescr(rel);
2103            /* a reverse map */
2104            map = convert_tuples_by_name_map_if_req(old_tupdesc,
2105                                                    tupdesc,
2106                                                    gettext_noop("could not convert row type"));
2107
2108            /*
2109             * Partition-specific slot's tupdesc can't be changed,
2110             * so allocate a new one.
2111             */
2112            if (map != NULL)
2113                    slot = execute_attr_map_slot(map, slot,
2114                                                 MakeTupleTableSlot(tupdesc, &TTSOpsVirtual));
2115    }
```

Modified Code Clone

File: src/backend/executor/execMain.c

```
1852    {
1853  +         TupleDesc       old_tupdesc;
        -         TupleDesc       old_tupdesc = RelationGetDescr(rel);
1854              AttrNumber *map;
1855
1856  +         root_relid = RelationGetRelid(resultRelInfo->ri_PartitionRoot);
1857  +         tupdesc = RelationGetDescr(resultRelInfo->ri_PartitionRoot);
1858  +
1859  +         old_tupdesc = RelationGetDescr(resultRelInfo->ri_RelationDesc);
        -         rel = resultRelInfo->ri_PartitionRoot;
        -         tupdesc = RelationGetDescr(rel);
1860              /* a reverse map */
1861              map = convert_tuples_by_name_map_if_req(old_tupdesc, tupdesc,
1862                                                      gettext_noop("could not convert row type"));
1863
1864              /*
1865               * Partition-specific slot's tupdesc can't be changed, so allocate a
1866               * new one.
1867               */
1868              if (map != NULL)
1869                      slot = execute_attr_map_slot(map, slot,
1870                                                   MakeTupleTableSlot(tupdesc, &TTSOpsVirtual));
1871      }
```

図 3.6　Inconsistent change (旧コミット ID: f7ea1a4233, 新コミット ID: e8b0e6b82d)

Stable Code Clone

File: src/backend/utils/cache/lsyscache.c

```
775     get_attname(Oid relid, AttrNumber attnum, bool missing_ok)
776     {
777             HeapTuple       tp;
778
779             tp = SearchSysCache2(ATTNUM,
780                                  ObjectIdGetDatum(relid), Int16GetDatum(attnum));
781             if (HeapTupleIsValid(tp))
782             {
783                     Form_pg_attribute att_tup = (Form_pg_attribute) GETSTRUCT(tp);
784                     char       *result;
785
786                     result = pstrdup(NameStr(att_tup->attname));
787                     ReleaseSysCache(tp);
788                     return result;
789             }
790
791             if (!missing_ok)
792                     elog(ERROR, "cache lookup failed for attribute %d of relation %
793                          attnum, relid);
794             return NULL;
795     }
```

Modified Code Clone

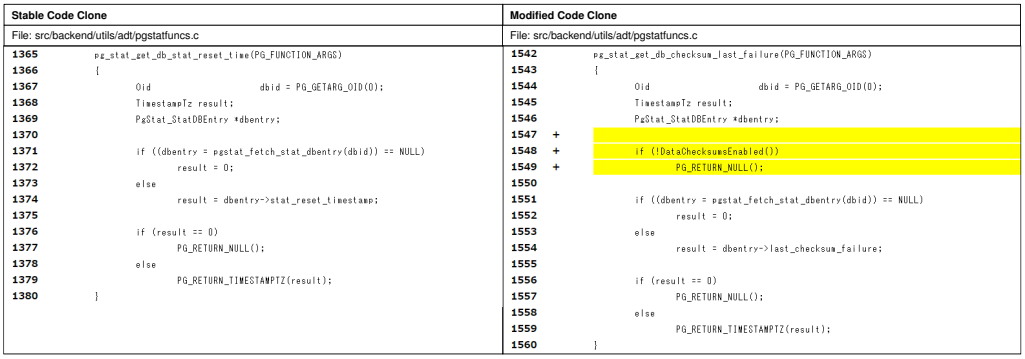
File: src/backend/utils/cache/lsyscache.c

```
836     get_attgenerated(Oid relid, AttrNumber attnum)
837     {
838             HeapTuple       tp;
839   +         Form_pg_attribute att_tup;
840   +         char            result;
841
842             tp = SearchSysCache2(ATTNUM,
843                                  ObjectIdGetDatum(relid),
844                                  Int16GetDatum(attnum));
845   +         if (!HeapTupleIsValid(tp))
        -         if (HeapTupleIsValid(tp))
        -         {
        -                 Form_pg_attribute att_tup = (Form_pg_attribute) GETSTRUCT(tp);
        -                 char            result;
        -
        -                 result = att_tup->attgenerated;
        -                 ReleaseSysCache(tp);
        -                 return result;
        -         }
        -         else
846                      elog(ERROR, "cache lookup failed for attribute %d of relation %u",
847                           attnum, relid);
848   +         att_tup = (Form_pg_attribute) GETSTRUCT(tp);
849   +         result = att_tup->attgenerated;
850   +         ReleaseSysCache(tp);
851   +         return result;
852     }
```

図 3.7　Inconsistent change (旧コミット ID: 82150a05be, 新コミット ID: edda32ee25)

Notifier から分析結果の概要が書かれたメールを受け取る．*inconsistent* クローンセットが検出されるなど，コードクローンに欠陥が残っていないかを調査する必要がある場合，メールに記載された分析結果のファイルパスにアクセスして詳細な分析結果を確認する．

最初に，トップページである図 3.4 が表示される．図 3.4 の ‘Changed clone set’ をクリックすると，Changed Clone Set の詳細情報が記されたページ (図 3.5) が開かれ，コードクローンの ID をクリックすると，コードクローンのソースコードページを閲覧できる．ソースコードページ（図 3.6, 図 3.7, 図 3.8）では，ソースコード中のコードクローンがある箇所を表示され，変更箇所は色分けされる．



Stable Code Clone

File: src/backend/utils/adt/pgstatfuncs.c

```
1365    pg_stat_get_db_stat_reset_time(PG_FUNCTION_ARGS)
1366    {
1367            Oid                     dbid = PG_GETARG_OID(0);
1368            TimestampTz result;
1369            PgStat_StatDBEntry *dbentry;
1370
1371            if ((dbentry = pgstat_fetch_stat_dbentry(dbid)) == NULL)
1372                    result = 0;
1373            else
1374                    result = dbentry->stat_reset_timestamp;
1375
1376            if (result == 0)
1377                    PG_RETURN_NULL();
1378            else
1379                    PG_RETURN_TIMESTAMPTZ(result);
1380    }
```

Modified Code Clone

File: src/backend/utils/adt/pgstatfuncs.c

```
1542    pg_stat_get_db_checksum_last_failure(PG_FUNCTION_ARGS)
1543    {
1544            Oid                     dbid = PG_GETARG_OID(0);
1545            TimestampTz result;
1546            PgStat_StatDBEntry *dbentry;
1547  +
1548  +         if (!DataChecksumsEnabled())
1549  +                 PG_RETURN_NULL();
1550
1551            if ((dbentry = pgstat_fetch_stat_dbentry(dbid)) == NULL)
1552                    result = 0;
1553            else
1554                    result = dbentry->last_checksum_failure;
1555
1556            if (result == 0)
1557                    PG_RETURN_NULL();
1558            else
1559                    PG_RETURN_TIMESTAMPTZ(result);
1560    }
```

図 3.8　Inconsistent change (旧コミット ID: 9010156445, 新コミット ID: 252b707bc4)

3.3.2　一貫性のない変更の検出事例

本節では，Clone Notifier によって検出された *inconsistent* クローンセットの事例を示す．実験対象として，PostgreSQL の 2018 年 6 月 23 日から 2019 年 6 月 22 日の 1 年間のコミット履歴に対して，コミット単位で Clone Notifier を適用した．その結果，全 2,152 件のコミットのうち 160 件において *inconsistent* クローンセットが検出された．また，30,382 個の変更されたクローンセットのうち 299 個が，*inconsistent* クローンセットとして分類された．これにより，従来のコードクローン変更管理システムと比べて，開発者による結果確認コストを大幅に削減できることが示された．

検出した 299 個の *inconsistent* クローンセットを目視で分析した結果，処理内容に影響を及ぼす一貫性のない変更を複数確認した．また，過去に一貫性のない変更が行われたクローンセットに対し，後のコミットで一貫性を復元するような変更が行われた事例も確認した．一方で，多くの *inconsistent* クローンセットは，コメントの追加や削除，空白や空行の変更など，処理内容に影響しない変更に起因していた．本節では，処理内容にかかわる変更の漏れによって生じた *inconsistent* クローンセットの事例を 3 件示す．

inconsistent クローンセットの 1 つ目の例は，図 3.6 に示すように，2018 年 12 月 29 日のコミット時の 3 つのコード片からなるクローンセットである．これらのコード片は同じファイル ‘src/backend/executor/execMain.c’ で検出された．このコミットの開発者は，クローンセット内の 1 つのコードクローンのみリファクタリングした．コミットメッセージによると，開発者は 2 週間前に他の開発者と電子メールでこのコードの可読性について議論し，該当箇所のコードが非常に複雑で保守するのが難しいためリファクタリングを施した[*3]．しかし，クローンセット内の 1 つのコードク

[*3] https://www.postgresql.org/message-id/20181206222221.g5witbsklvqthjll@alvherre.pgsql

ローンが複雑であった場合，他のコードクローンも複雑であるとみなされるため，他のコード片についてもリファクタリングすべきである．

inconsistent クローンセットの 2 つ目の例は，図 3.7 に示すように，2019 年 4 月 5 日のコミット時の 3 つのコード片からなるクローンセットである．これらのコード片は同じファイル ‘src/backend/utils/cache/lsyscache.c’ で検出された．このコミットの開発者は，クローンセット内の 1 つのコードクローンのみ，リファクタリングを施した．コミットメッセージによると，開発者はコンパイラの警告を避けるために get_attgenerated() を修正した．しかし，条件文が条件否定形に変わるようにリファクタリングされており，タイプ 2 コードクローンから，類似した処理を実行するが構文上の実装が異なるタイプ 4 コードクローンに変化した．他のコードクローンではコンパイラ警告が発生せずとも，可読性や保守性の観点から他のコードクローンについても条件文を修正するリファクタリングすべきである．

inconsistent クローンセットの 3 つ目の例は，図 3.8 に示すように，2019 年 4 月 17 日のコミット時の 2 つのコード片からなるクローンセットである．これらのコード片は同じファイル ‘src/backend/utils/adt/pgstatfuncs.c’ で検出された．このコミットの開発者は，クローンセット内の 1 つのコードクローンのみ処理を追加した．コミットメッセージによると，0 を返すことは問題がないことを誤って示す可能性があるが，NULL を返すことは潜在的な問題についての情報がないことを正しく示すため，DataChecksumsEnabled が false の場合に NULL を返す処理が追加された．もう一方のコードクローンも類似した処理内容であり，同様の処理を追加すべきか検討する余地がある．

3.4 考察

本章で提案したコードクローンの変更情報を開発者に通知するシステム Clone Notifier について，本手法の有効性と妥当性について考察する．本手法では，山中らが提案したコードクローン変更管理システムにおけるクローンセットの分類の 1 つである Changed Clone Set をさらに詳細に分類し，それぞれのクローンセットにラベルを付与することで，開発者の確認作業の効率化を図った．特に，本研究では，一貫性のない変更を含むクローンセット（*inconsistent*）を定義し，それを検出してラベル付けすることで，開発者はすべての Changed Clone Set を確認することなく，重要な変更のみを効率的に把握できるようになる．実際，Clone Notifier は，3.3.2 節で示した通り，30,382 個の Changed Clone Set のうち 299 個を *inconsistent* として検出しており，開発者の検出結果の確認工数を大幅に削減することができたと考えられる．

一方で，本手法では 2 つのリビジョンそれぞれに対してコードクローン検出する必要があるため，大規模なソースコードに適用する際には処理時間が大きな課題となる．コードクローン検出の実行時間や精度は，使用するコードクローン検出器のアルゴリ

ズムや実装に依存するが，インクリメンタルなコードクローン検出 [68,69] など，効率よくコードクローンを追跡する手法も研究されている．インクリメンタルなコードクローン検出を用いた追跡手法の検討が今後の課題である．

また，Clone Notifier では，リファクタリング候補を探すためには New Clone Set などを手動で分析する必要がある．JDeodorant のリファクタリング可能性判定 [33] に基づいたリファクタリング可能なクローンセットを特定することで，コピーアンドペーストなどによって発生したコードクローンを早期に発見・除去できると考えられる．

3.5 まとめと今後の課題

本章では，コードクローンの変更情報を開発者に通知するシステム Clone Notifier について述べた．また，一貫性のない変更を検出して通知する Clone Notifier のユースケースを示した．我々は，Clone Notifier を用いて，PostgreSQL の 1 年分のコミット履歴の 2152 個のコミットから，160 個のコミットで一貫性のない変更が行われたクローンセットを検出した．また，30,382 個の変更されたクローンセットのうち，299 個のクローンセットが一貫性のない変更として分類された．これにより，従来のコードクローン変更管理システムと比べて，大規模ソフトウェアに対する検出結果の確認コストを軽減した．本評価実験で検出した一貫性のない変更のうち，明確な実例を 3 件発見した．

今後の課題として，大規模なソースコードに対してコードクローン検出すると時間がかかるため，インクリメンタルなコードクローン検出による追跡手法の検討が挙げられる．また，リファクタリング可能なクローンセットを特定する機能の追加することが挙げられる．

第 4 章

コードクローン集約によるファジングの実行効率調査

4.1 まえがき

ソフトウェア開発において，ソフトウェアが顧客の要求にしたがって動作するかを確認するために，ソフトウェアテストが実施される．ソフトウェアテストでは漏れのないテストを実施するために，テスト観点の洗い出しや単体テストの追加などが行われる．しかし，ソフトウェア規模の増大に伴い，開発工数におけるソフトウェアテストの割合は増加し，時間的コストをかけてテストケースを作成しても検出が困難な不具合が存在する可能性がある．従来の手動テストだけでなく，自動テスト技術が多く研究されている [50].

自動テスト手法の 1 つであるファジングは，自動で大量のテストを生成・実行する [51]．ファジングの代表的なツールとして American Fuzzy Lop (AFL)[*1] が挙げられる．AFL は未発見であった多くの不具合を発見した実績がある [1][*2]．AFL は，for 文や if 文，関数などの基本ブロック単位での遷移を実行パスと捉え，実行パスを観測しながらテストケースを生成・実行する．生成したテストケースが新しいパスを実行すると，そのテストケースを保存し，保存したテストケースをもとに新たなテストケースを生成する．このように，AFL は遺伝的アルゴリズムを用いたテストケース生成を行うため，24 時間以上かけて探索することが多い [1,51].

ファジング対象のソースコードに含まれるコードクローンは，AFL のパス探索効率を低下させる可能性がある．コードクローンがプログラム中に複数存在する場合，実行時の挙動はほぼ同一であるにもかかわらず，それぞれのコードクローンに対して別々の実行パスとして探索を繰り返すことになる．その結果，実質的に同じ処理を行うパスが多数生成されるため，パス探索が冗長になると考えた．冗長なパス探索が続

[*1] https://lcamtuf.coredump.cx/afl
[*2] https://lcamtuf.coredump.cx/afl/#bugs

くことで，より深い階層にある未知のパスや潜在的な障害箇所への到達が遅れ，特に大規模なソフトウェアでは長時間ファジングを実行しても探索深度を十分に広げられない可能性がある．この問題に対して，基本ブロックを含むコードクローンを集約することで，AFL が観測するパスの総数を削減し，未発見のパスに到達しやすくなるという仮説のもと，コードクローン集約前後のプログラムに対する AFL の比較評価を実施した．

本章では，GNU Binutils を対象に CCFinderX を用いてコードクローンを検出し，AFL がコードクローン集約により短時間で多くのパスを効率的に検出するかどうかを調査した．加えて，集約前後の比較では，ソフトウェアテストで通常用いられるブランチカバレッジではなく，基本ブロックの遷移に基づく AFL カバレッジを用いた．実験では，初期テストケース，乱数生成関数の初期値，AFL が生成するテストケースの個数を固定し，コードクローン集約前後のプログラムを AFL の入力として実験を行った．

実験の結果，コードクローン集約前後でパス数に統計的な有意差がないことを確認した．コードクローン集約による AFL の実行効率の向上は難しいと考えられる．一方で，コードクローン集約は AFL が生成するテストケースに変化を及ぼし，未発見のクラッシュを 1 件検出した．AFL が生成したテストケースの一致率を調査し，コードクローン集約により AFL の挙動が変化していることを確認した．コードクローン集約はファジングの挙動を変化させ，大規模ソフトウェアの潜在的な障害の早期発見に寄与する可能性がある．

以降，4.2 節では，背景として CCFinderX と AFL について述べる．4.3 節では，本実験について説明する．4.4 節では，実験結果の考察と妥当性の脅威について論じる．4.5 節では，まとめと今後の課題について述べる．

4.2 背景

4.2.1 CCFinderX

CCFinderX は，プログラム中に含まれる同一または類似したコード断片であるコードクローンを検出する手法の 1 つである [20]．字句単位でコードクローンを検出し，空白や改行を除いて一致するタイプ 1 のコードクローン，および，識別子（変数名，関数名，変数の型など）のみが異なるタイプ 2 のコードクローンを検出する．

コードクローンの存在は，バグの潜在要因になることがあるため，ソフトウェア保守者や開発者はコードクローンに対するリファクタリングを実施する [15, 18, 70]．1.2.2 節に述べたように，コードクローンに対するリファクタリングは，1 つの関数に集約するなどして，ソースコード中からコードクローンを除去することを指す．

本章では，CCFinderX で検出される，構文的に一致するタイプ 1,2 のコードクロー

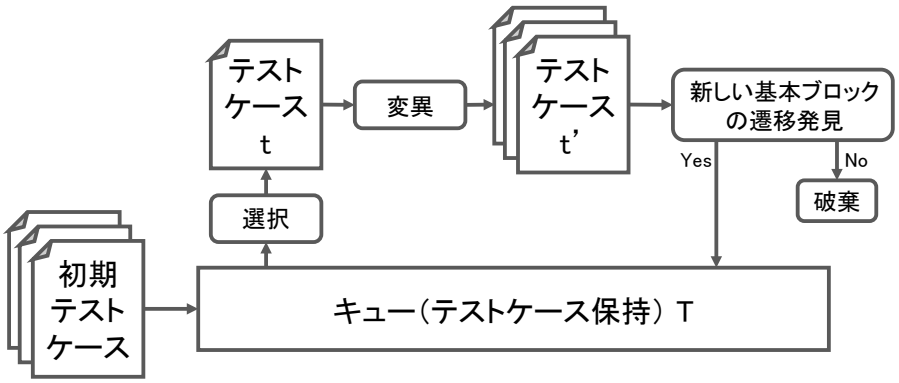


図 4.1　テストケースに着目した AFL の動作フロー [1]

ンを，本章の集約対象とする．一方で，タイプ 3 のコードクローンは集約の対象としない．これは，タイプ 3 のコードクローンを集約する際に，一部の行がコードクローンの範囲外へ移動される可能性があり，その結果として，パスの削減に寄与しない場合があるためである．

4.2.2　AFL(American Fuzzy Lop)

AFL は，ファジングの代表的なツールの 1 つである．AFL はプログラム実行中の情報を利用して実行パスを効率的に探索し，可能な限り多くのコードカバレッジを網羅するテストを生成・実行する [1,57]．このように実行中の情報を用いてファジングする手法は，グレイボックスファジングと呼ばれる．グレイボックスファジングは，プログラム解析を用いて静的にテストケース生成するホワイトボックスファジングに比べ，プログラミング言語依存などの制約が少なく，入出力のみでテストケースを生成するブラックボックスファジングよりも効率的にテストケースを探索する．

AFL の動作フローについて説明する．図 4.1 はその一連の流れを示している．AFL はユーザが用意した初期テストケースをキューに保存する．次に，キューからテストケースを 1 つ選択し，数値の加減算やビット反転などの変異戦略に基づいて新たなテストケースを生成する．新たに生成したテストケースをテスト対象に入力し，基本ブロックに基づく遷移などの実行中の情報を取得する．取得した情報から，これまでのテストケースでテストしていなかったプログラムの実行パスをテストしたかを判断する．

AFL は判断基準として AFL カバレッジ[*3]を用いる． AFL カバレッジは基本ブロック単位での遷移をもとに実行パスを分類する．AFL 実行中に発見されていない遷移を検出した場合，新たな実行パスをテストしたと判断する．AFL カバレッジに基づ

[*3] https://github.com/google/AFL/blob/master/docs/technical_details.txt

表 4.1　実験対象

プログラム	引数	機能
nm	-C	低位レベルのシンボル名をユーザーレベルの名前にデコードして列挙する.
objdump	-d	機械語命令に対応するアセンブラのニーモニックを表示する.
readelf	-a	すべてのファイルの情報を表示する.

いて判断した結果，生成したテストケースが新たなパスをテストした場合，そのテストケースをキューに保存する．そうでないテストケースは，同じパスをテストする他のテストケースが既に存在することを意味するため，破棄する．一方で，クラッシュを発生させたテストケースは，クラッシュ情報とともに報告する．ファジングの比較指標として，一定時間内に発見したパス数やクラッシュ数が用いられることが多い．

AFL は遺伝的アルゴリズムを用いたテストケース生成を行うため，24 時間以上かけて探索することが多い．短時間でより多くの欠陥を検出することが求められており，効率的に探索する手法が多く研究されている．本研究では，コードクローン集約によって AFL の実行効率を向上できるという仮説をもとに実験調査を行った．

4.3　実験

本節では，コードクローンを集約することで AFL の実行効率を向上させられるかを調査するための実験方法や実験結果について述べる．

4.3.1　研究課題

4.2.2 節で述べたように，AFL は AFL カバレッジに基づき，短時間で多くのパスを検出するようにテストケースを生成し実行する．大規模なソフトウェアでは，長時間ファジングを実行してもより深い階層にある未知のパスや潜在的な障害箇所へ到達することが難しい．コードクローンがプログラム中に複数存在する場合，実行時の挙動はほぼ同一であるにもかかわらず，それぞれのコードクローンに対して別々の実行パスとして探索を繰り返すことが冗長であることが原因であると考えた．基本ブロックを含むコードクローンを集約することで，そのコードクローンに含まれるパスが 1 度しか検出されなくなり，新たなパスを発見したときに保存されるテストケースのユニーク性が増し，未発見のパスに到達しやすくなると考えられる．

本実験では，ファジングがより効率的に行われるために，コードクローン集約が有効かどうかを調査する．ファジングには終了条件がないため，一定時間で探索したパスやクラッシュの数を用いて評価することが多い [57]．本研究では，同じ回数のテス

トケース生成によって探索できたパスやクラッシュの数が，コードクローン集約によって増加するかどうかが重要であると考え，テストケース生成数を固定して実験を実施した．結果として，コードクローン集約前後のパス数に統計的な有意差は認められなかった．生成したテストケースの一致率やクラッシュについての調査結果と考察は 4.3.4 節で述べる．

4.3.2 実験対象

本実験の対象プログラムおよび初期テストケースについて述べる．本実験では，ファジングの評価方法に関する論文 [57] を参考に選択した．

本実験では，GNU Binutils の 2 バージョン v2.26, v2.32 において，3 プログラム nm, objdump, readelf を実験対象とした．ただし，これらの機能をすべて対象とするのではなく，表 4.1 に示すオプションで実行した場合を対象とした．これらのプログラムはコードクローンを一定数保有しており，本研究の対象として適切であると考えた．

また，初期テストケースの形式の違いを考慮するため，実行可能な入力 (valid) として 10 行程度の簡易なプログラムと，無効な入力 (invalid) として空行を初期テストケースとした．nm のバージョン v2.26 に対して invalid な初期テストケースを入力とした実験結果を nm_26_invalid と表す．

4.3.3 実験方法

本実験では，以下の手順でコードクローンを集約し，AFL を実行した．

1. CCFinderX を用いてコードクローンを検出
2. 集約可能なコードクローンを集約
3. 各プログラムに一定回数で停止する AFL を実行
4. 出力結果のパス数とクラッシュ数を比較

CCFinderX を用いて実験対象プログラムのコードクローンを検出した．CCFinderX はトークン単位で一致するコードクローンを検出する．本実験では，トークン単位から行単位のコードクローンに変換した．

次に，集約可能なコードクローンを集約した．本実験の実験対象は１つのファイル内でプログラムがほとんど完結しているため，ファイル内コードクローンのみ集約対象とした．コードクローンは，関数名や変数の型が不一致である場合１つの関数に集約できない．そのため，関数名や変数の型が一致するコードクローンを集約可能なコードクローンとして１つの関数に集約した．

最後に，各プログラムに対して AFL を実行し，コードクローン集約前後で検出されたパス数とクラッシュ数を比較した．本実験では，AFL の乱数生成関数の初期値を固

表 4.2　実験結果: 1 千万回テストケース生成を実行する AFL 実行により検出されたパス数

プログラム	nm		objdump		readelf	
refactoring	origin	refactored	origin	refactored	origin	refactored
26_invalid	2,446	**2,571**	**2,219**	2,102	17	**18**
26_valid	831	**835**	1,560	**1,562**	1,671	**1,984**
32_invalid	**1,925**	825	**912**	841	**17**	**17**
32_valid	**709**	701	**1,664**	1,645	1,878	**1,984**

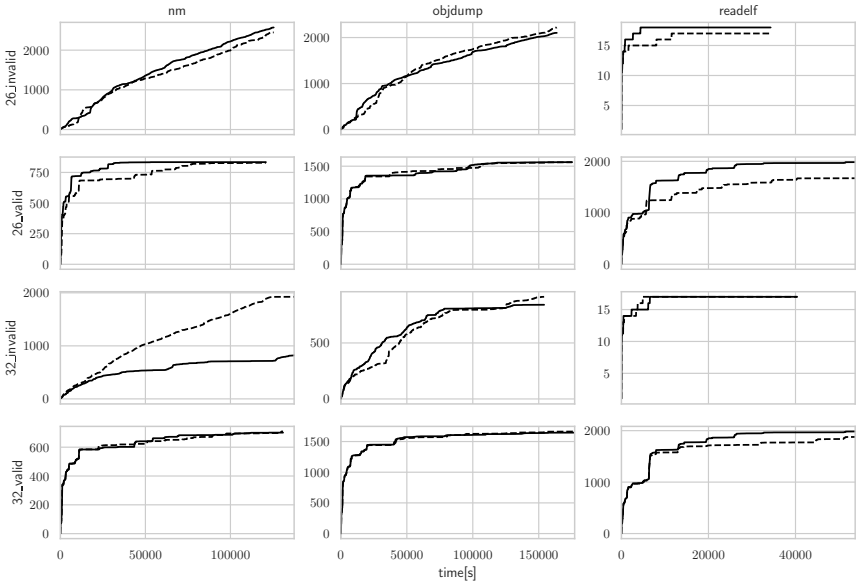


図 4.2　AFL 実行により検出されたパス数 (破線: コードクローン集約前, 実線: コードクローン集約後)

定し，一度の AFL 実行におけるテストケース生成回数を一千万回に固定した．

本研究では，同一回数のテストケース生成での探索でより多くのパスが検出できるかどうかが重要であると考え，パスとクラッシュの検出数をコードクローン集約前後で比較した．コードクローン集約前後で検出したパス数の差について，統計的な有意差を確認するため，t 検定を用いて比較した．

4.3.4　実験結果

本実験では，各プログラムに対してコードクローン集約を行い，AFL を実行してパス数とクラッシュ数を計測し，コードクローン集約前後でパス数やクラッシュ数が増

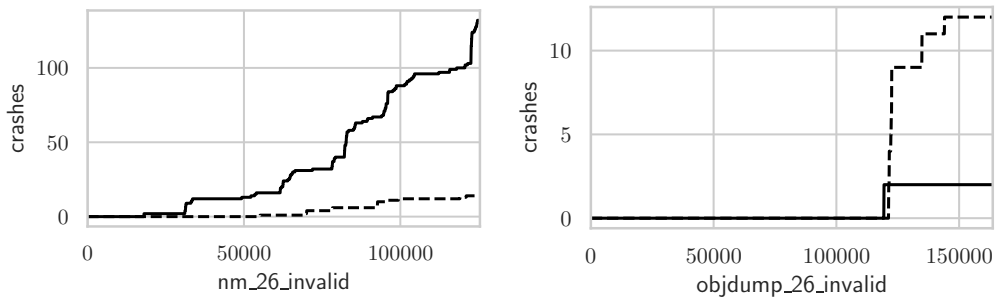


図 4.3　AFL 実行により検出されたクラッシュ数
(破線:orginal, 実線:refactored)

表 4.3　AFL が保存したキューの一致率

program	nm	objdump	readelf
26_invalid	0.00%	0.05%	0.00%
32_invalid	0.11%	0.00%	0.00%
26_valid	79.64%	76.71%	41.63%
32_valid	71.26%	81.96%	46.93%

加したかどうかを調査した．また，ユニークなパスを発見したテストケースの一致率を調べることで，AFL の挙動が変化したかどうかを調査した．本節では，実験結果について述べる．

検出されたパスの総数を表 4.2 に示し，AFL で検出されたパス数の推移を図 4.2 に示した．ほとんどのプログラムでコードクローン集約前後で異なるグラフが形成されており，コードクローン集約により AFL の挙動が変化していることが分かった．しかし，そのグラフの差異は大きくなく，パス数の推移は nm_32_invalid の場合を除き，検出されたパス数の推移に大きな違いは見られなかった．

コードクローン集約によるパス数の変化について統計的な分析を行った．実験対象ごとに，1 時間間隔で新たに検出されたパス数を計測し，各時間においてコードクローン集約前後のパス数の差分を取得した．この差分をもとに各時間ごとに，帰無仮説をコードクローン集約前後のパス数に差がないとして，有意水準 5% の t 検定を実施した．その結果，ある 1 時間を除いて帰無仮説が棄却されなかった．実験開始から 22 時間後から 23 時間後の区間では $p = 0.03$ となり，コードクローン集約前の方が新たに検出したパス数が統計的に多いことが示された．しかし，それ以外の 47 区間では帰無仮説が棄却されず，ほとんどの場合でコードクローン集約前後のパス数に統計的な有意差がみられなかった．

クラッシュ検出数のグラフを図 4.3 に示した．本実験でクラッシュを検出したプログラムは 2 つの実験対象のみであり，ほとんどのクラッシュは Segmentation Fault

であることを確認した．AFL が検出したクラッシュは複数のテストケースが同一の欠陥を検出しているケースが多い．しかし，コードクローン集約後の nm_26_invalid でメモリエラーを発生するクラッシュを検出した．メモリエラーはコードクローン集約前では検出されておらず，このクラッシュを発生させたテストケースをコードクローン集約前の nm_26_invalid に入力した場合でもメモリエラーが発生したことから，nm_26_invalid ではコードクローン集約によって未知のクラッシュを発見したと言える．

表 4.3 は，ユニークなパスを発見したテストケースとして保存されたキューのコードクローン集約前後の一致率を示す．無効な初期テストケースにおいて，コードクローン集約前後で一致したキューの割合は 0.1% 以下であり，有効な初期テストケースにおける一致率は 41% 以上 82% 以下だった．さらに，一致したキューのテストケースを入力として実験対象プログラムを直接実行した結果，すべての一致したテストケースが集約したコードクローンを実行していないことが確認された．一方で，集約したコードクローンを実行したテストケースは，コードクローン集約前後で一致しないテストケースであることが確認された．このことから集約したコードクローンを実行した際に，AFL の挙動に変化を及ぼすと考えられる．すべての実験対象において，無効な初期テストケースを入力としたとき，集約したコードクローンを実行することを確認した．

4.4 考察

本実験では，コードクローン集約によって AFL の実行効率を向上できるかどうかを調査するため，コードクローン集約前後で検出されたパスやクラッシュの数を比較した．結果として，パス数に統計的有意差が見られなかった．一方で，nm_26_invalid でコードクローン集約によってメモリエラーを 1 件新たに検出した．しかし，実験対象全体としてはクラッシュ数に変化はほとんどなく，コードクローンの集約によってクラッシュ数が増加するとは言えないことが示された．

AFL は遺伝的アルゴリズムを用いてテストケースを生成するため，生成元のテストケースが異なると異なるテストケースが生成される．コードクローン集約前後のキューの一致率の実験において，集約したコードクローンを実行したテストケースから，異なるテストケースが生成されたことが確認された．表 4.3 の結果から，すべての実験対象において，一部のテストケースがコードクローン集約前後で一致しなかった．このことからコードクローン集約は AFL の挙動に変化を及ぼすことが示された．開発者はコードクローンのリファクタリングを実施することによって，プログラムの挙動を変化させずに AFL の挙動を変化させることができ，本評価実験のように未発見のクラッシュを検出できる可能性がある．

本評価実験では，コードクローン集約の有無による影響を同一条件下で公平に比

較するため，AFL のテストケース実行を一千万回で停止する設定とした．一般的に，ファジングの評価実験では，6 時間以上の一定時間実施する場合が多いが，本実験では，最短でも 9 時間以上実施しており，実行回数としては十分であると考えられる．実行回数をさらに増加させると，ファジングの探索範囲が広がり，集約後にのみ検出されたクラッシュが集約前でも検出される可能性がある．一方で，より多様な入力の生成により，集約前後のテストケースの一致率はさらに低下する可能性がある．

本研究では，C 言語形式のファイルを入力とする GNU Binutils のプログラムのみを実験対象とした．入力形式が異なるプログラムや，コードクローンを多く含むプログラムを対象とした場合，本実験と同様の結果が得られるとは限らない．

4.5 まとめと今後の課題

本章では，コードクローン集約による AFL の実行効率の調査を実施した．結果として，AFL が検出したパス数に有意な差は見られず，コードクローン集約による AFL の実行効率が向上するとは言い難いことが示された．一方で，集約したコードクローンを実行したテストケースから，異なるテストケースが生成されることを確認した．これは，コードクローン集約が AFL の入力生成の挙動に一定の影響を及ぼすことを示しており，大規模ソフトウェアの潜在的な障害の早期発見に寄与する可能性がある．

今後の課題として，異なるプログラムに対する実験や AFL 以外のファジングツールでの実験の拡張が挙げられる．また，コードクローン集約の手法として，JDeodorant [33] などのコードクローン自動集約ツールなどを用いた場合の実験評価を行うことで，より実用的な適用可能性の検証が求められると考えられる．さらに，コードクローンを含むパスの検出数や，クラッシュが発生したコード片がコードクローンかどうかを分析することで，ファジング実行時のコードクローンの影響調査をさらに進めることができると考える．最後に，リファクタリング困難なコードクローンに対して重点的にファジングする手法の提案が今後の課題として挙げられる．

第 5 章

おわりに

5.1 まとめ

本論文では，ソフトウェア保守の品質を低下させる要因の 1 つである，コードクローンの管理手法である検出・追跡・集約に着目して 3 つの研究を実施した．

1. Cross-Polytope LSH を用いたコードクローン検出のためのパラメータ決定手法
2. コードクローン変更管理システムの開発と改善
3. コードクローン集約によるファジングの実行効率調査

1 については，LSH を用いたコードクローン検出において，利用者が与えた再現率の目標値を満たしつつ可能な限り検出時間を短縮することを目的として，プロジェクトの規模に応じた Cross-Polytope LSH の適切なパラメータを線形回帰モデルにより自動決定する手法を提案した．これにより，CCVolti の利用者は，再現率の目標値を下げて高速なパラメータを選択することで，大規模なプロジェクトに対して頻繁にコードクローン検出し，修正の即時対応やコードクローンの早期発見，追跡手法への応用を可能とする．さらに，既存のデフォルトのパラメータ値よりも高速にコードクローンを検出可能であることを示した．

2 については，コードクローンに対する一貫性のない変更を検出し，開発者に通知するシステムを提案した．これにより，大規模ソフトウェアの開発者は，膨大なコードクローン変更情報から保守作業が必要なコードクローンを手作業で識別するコストを大幅に削減した．本評価実験では，変更されたクローンセットのうち一貫性のない変更が行われたクローンセットは 1% であることを確認し，処理内容に影響を及ぼす修正すべき一貫性のない変更の実例を 3 件発見した．

3 については，テスト対象のソースコード内のコードクローンを集約することによる，ファジングのパス探索の実行効率について調査した．調査の結果，ファジングで発見したパス数に統計的な有意差は見られなかったが，未発見のクラッシュを 1 件検

出し，コードクローン集約はファジングが生成するテストケースを変化させることを確認した．この結果は，コードクローン集約が AFL の入力生成挙動に一定の影響を及ぼすことを示しており，大規模ソフトウェアの潜在的な障害の早期発見に寄与する可能性がある．

これらの研究により，大規模ソフトウェア保守における，潜在的な障害の早期発見を支援することが可能となる．

5.2 今後の研究方針

今後の研究方針として，本論文で述べた研究を応用し，ソフトウェア保守の品質や効率をより向上させる支援をしていきたいと考えている．本論文で提案した Clone Notifier にインクリメンタルなコードクローン検出手法 [68, 69] を応用して効率的な追跡手法を検討し，大規模言語モデル（Large Language Model，以下 LLM）を利用して一貫した修正方法を提示する手法を検討したいと考えている．さらに，本論文で提案した LSH のパラメータ決定手法やコードクローン変更管理手法，コードクローン集約によるファジングの実行効率について得られた知見を実プロジェクトに適用し，手法の有用性を評価を進めたいと考えている．

コードクローン検出の分野では，タイプ 4 のコードクローン検出が課題として残されている [71]．本論文では，任意の目標再現率に対して LSH のパラメータを自動的に決定することで，LSH を利用するコードクローン検出器の検出精度を保ちつつ実行速度を短縮する手法を提案した．近年では，深層学習や LLM を用いたコードクローン検出手法が提案されており，タイプ 4 のコードクローンを高い精度で検出できたことが報告されている [72, 73]．一方で，LLM のモデルやプロンプトの違いによって検出精度が大きく変化する点は重大な課題の 1 つであると考えている [74–76]．LLM のプロンプト開発には一定の知識と経験が必要であり，LLM を用いたコードクローン検出をより安定させるためのプロンプトの調査研究を進めたいと考えている．

コードクローン管理の分野では，追跡の精度向上が課題として残されている [16]．本論文では，既存手法のコードクローン変更管理システム Clone Notifier の追跡と分類を改善し，一貫性のない変更が行われたコードクローンの特定を行った．近年では，テストコードにコードクローンが多く含まれることが示されており [77–79]，テストコードのコードクローンを追跡し，一貫した変更を支援することが求められている．テストコードはプロダクトの対応する関数などが変更された場合，関数名や引数の修正が必要になり，テストコード内のコードクローンだけでなくプロダクトとの依存関係も追跡する必要がある．そこで，Clone Notifier の変更管理手法に，テストコードのテスト対象の依存関係追跡と LLM を用いた同期案の提示を加えることで，プロダクト変更時に関連するテストコードの一貫した変更を支援する手法を提案したいと考えている．

また，多言語のコードクローン追跡や大規模ソフトウェアのコードクローン追跡も課題として残されており [16]，Git ログを調査した結果からクローンペアの約半数が潜在的に一貫性のない変更であると懸念されるという報告があった [80]．近年では，多言語コードクローン検出手法 [81,82] や，DNN を用いて効率よく一貫性のない変更を検出する手法 [83] が提案されている．しかし，多言語のコードクローンを DNN に学習させることは，学習データを用意する観点や精度の保証について困難が伴う．そこで，多言語コードクローン検出を Clone Notifier に組み込み，コミットログを用いてコードクローンに対する一貫性のない変更を素早く効率的に検出する手法を提案したいと考えている．

ソフトウェアテストにおいて重要なソフトウェアの品質保証の観点が，ファジングにおいて重視されていないことはソフトウェアテスト従事者にとって課題の 1 つであり，コードカバレッジを増加させることを目的とした AFL の拡張ツールの研究開発が今後の課題だと考えている [58,84]．本論文では，リファクタリング可能なコードクローンを集約することにより，ファジングにおけるテストケース生成結果を変化させ，これまで未発見であった障害を検出する可能性を示した．一方で，コードクローンの多くはリファクタリング困難であり，ソースコード中のすべてのコードクローンを除去することはできない．そこで，リファクタリング困難なコードクローンの存在を活用し，クラッシュが発生したコードクローンを中心に探索を行ったり，コードクローンを実行するテストケースを初期入力データとして用いたりすることで，コードカバレッジの向上を図る新たなファジング手法の提案を目指したいと考えている．

参考文献

[1] M. Böhme, V. T. Pham, A. Roychoudhury, Coverage-based greybox fuzzing as markov chain, in: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (ACM CCS'16), ACM, 2016, pp. 1032–1043.

[2] I. S. Committee, IEEE standard glossary of software engineering terminology, Vol. 610, IEEE Std, 1990.

[3] Standard for software maintenance, IEEE Std 1219-1998 (1998) 1–56.

[4] Guideline on Software Maintenance, Federal Information on Processing Standards, 1984.

[5] Y. Singh, B. Goel, A step towards software preventive maintenance, ACM SIGSOFT Software Engineering Notes 32 (4) (2007) 10—es.

[6] Software engineering – Software life cycle processes – Maintenance (2006).

[7] ソフトウェア技術–ソフトウェアライフサイクルプロセス–保守 (2008).

[8] B. P. Lientz, Issues in software maintenance, ACM Computing Surveys (CSUR) 15 (3) (1983) 271–278.

[9] M. Page-Jones, The practical guide to structured systems design, Yourdon Press, 1988.

[10] S. W. Yip, T. Lam, A software maintenance survey, in: Proceedings of the 1st Asia-Pacific Software Engineering Conference (APSEC'94), IEEE, 1994, pp. 70–79.

[11] A. April, A. Abran, Software Maintenance Management: Evaluation and Continuous Improvement, John Wiley & Sons, 2008.

[12] C. K. Roy, J. R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, Science of Computer Programming 74 (7) (2009) 470–495.

[13] D. Rattan, R. Bhatia, M. Singh, Software clone detection: A systematic review, Information and Software Technology 55 (7) (2013) 1165–1199.

[14] K. Inoue, C. K. Roy, Code Clone Analysis: Research, Tools, and Practices, Springer, 2021.

[15] 肥後芳樹, 吉田則裕, コードクローンを対象としたリファクタリング, コンピュー

タソフトウェア 28 (4) (2011) 4_43–4_56.

[16] M. Mondal, C. K. Roy, K. A. Schneider, A survey on clone refactoring and tracking, Journal of Systems and Software 159 (2020) 110–429.

[17] M. Kim, V. Sazawal, D. Notkin, G. Murphy, An empirical study of code clone genealogies, in: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05), Vol. 30, ACM, 2005, pp. 187–195.

[18] 肥後芳樹, 楠本真二, 井上克郎, コードクローン検出とその関連技術, 電子情報通信学会論文誌 D 91 (6) (2008) 1465–1481.

[19] B. S. Baker, Parameterized duplication in strings: Algorithms and an application to software maintenance, Journal on Computing 26 (5) (1997) 1343–1362.

[20] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: A multilinguistic token-based code clone detection system for large scale source code, Transactions on Software Engineering (TSE'02) 28 (7) (2002) 654–670.

[21] 横井一輝, 崔恩瀞, 吉田則裕, 井上克郎, 情報検索技術に基づく細粒度ブロッククローン検出, コンピュータソフトウェア 35 (4) (2018) 16–36.

[22] R. Komondoor, S. Horwitz, Using slicing to identify duplication in source code, in: Proceedings of the 8th International Static Analysis Symposium (SAS'08), Springer, 2001, pp. 40–56.

[23] J. Mayrand, C. Leblanc, E. Merlo, Experiment on the automatic detection of function clones in a software system using metrics, in: Proceedings of the International Conference on Software Maintenance (ICSM'96), IEEE, 1996, pp. 244–253.

[24] R. Baeza-Yates, B. Ribeiro-Neto, Modern information retrieval: The concepts and technology behind search, Addison-Wesley, 2011.

[25] A. Andoni, I. Piotr, L. Thijs, R. Ilya, S. Ludwig, Practical and optimal LSH for angular distance, in: Proceedings of the Advances in Neural Information Processing Systems (NeurIPS'15), 2015, pp. 1225–1233.

[26] L. Jiang, G. Misherghi, Z. Su, S. Glondu, DECKARD: Scalable and accurate tree-based detection of code clones, in: Proceedings of the 29th International Conference on Software Engineering (ICSE'07), IEEE, 2007, pp. 96–105.

[27] A. Gionis, P. Indyk, R. Motwani, et al., Similarity search in high dimensions via hashing, in: Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'99), Vol. 99, VLDB Endowment, 1999, pp. 518–529.

[28] P. Indyk, R. Motwani, Approximate nearest neighbors: towards removing the

curse of dimensionality, in: Proceedings of the 30th annual ACM symposium on Theory of computing (STOC'98), 1998, pp. 604–613.

[29] N. Ailon, B. Chazelle, The fast Johnson–Lindenstrauss transform and approximate nearest neighbors, Journal on computing 39 (1) (2009) 302–322.

[30] K. Q. Weinberger, A. Dasgupta, J. Langford, A. J. Smola, J. Attenberg., Feature hashing for large scale multitask learning, in: Proceedings of the 26th Annual International Conference on Machine Learning (ICML'09), ACM, 2009, pp. 1113–1120.

[31] 徳井翔梧, 吉田則裕, 崔恩瀞, 井上克郎, 局所性鋭敏型ハッシュを用いたコードクローン検出のためのパラメータ決定手法, in: 電子情報通信学会技術研究報告, Vol. 117, 2018, pp. 57–62.

[32] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 2018.

[33] D. Mazinanian, N. Tsantalis, R. Stein, Z. Valenta, JDeodorant: clone refactoring, in: Proceedings of the 38th international conference on software engineering (ICSE'16), IEEE, 2016, pp. 613–616.

[34] 石津卓也, 吉田則裕, 崔恩瀞, 井上克郎, コードクローンのリファクタリング可能性に基づいた削減可能ソースコード量の分析, 情報処理学会論文誌 60 (4) (2019) 1051–1062.

[35] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, E. Lee, Experience of finding inconsistently-changed bugs in code clones of mobile software, in: Proceedings of the 6th International Workshop on Software Clones (IWSC'12), IEEE, 2012, pp. 94–95.

[36] L. Jiang, Z. Su, E. Chiu, Context-based detection of clone-related bugs, in: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE'07), ACM, 2007, pp. 55–64.

[37] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, Do code clones matter?, in: Proceedings of the 31th International Conference on Software Engineering (ICSE'09), IEEE, 2009, pp. 485–495.

[38] R. C. Miller, B. A. Myers, Interactive simultaneous editing of multiple text regions, in: Proceedings of the USENIX Annual Technical Conference (USENIX ATC'01), 2001, pp. 161–174.

[39] M. Toomim, A. Begel, S. L. Graham, Managing duplicated code with linked editing, in: Proceedings of the Symposium on Visual Languages-Human Centric Computing (VL/HCC'04), IEEE, 2004, pp. 173–180.

[40] M. De Wit, A. Zaidman, A. Van Deursen, Managing code clones using dy-

namic change tracking and resolution, in: Proceedings of the International Conference on Software Maintenance (ICSM'09), IEEE, 2009, pp. 169–178.

[41] Y. Lin, X. Peng, Z. Xing, D. Zheng, W. Zhao, Clone-based and interactive recommendation for modifying pasted code, in: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE'15), 2015, pp. 520–531.

[42] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, T. N. Nguyen, Clone management for evolving software, transactions on software engineering (TSE'11) 38 (5) (2011) 1008–1026.

[43] X. Cheng, H. Zhong, Y. Chen, Z. Hu, J. Zhao, Rule-directed code clone synchronization, in: Proceedings of the 24th International Conference on Program Comprehension (ICPC'16), IEEE, 2016, pp. 1–10.

[44] J. Harder, N. Göde, Efficiently handling clone data: Rcf and cyclone, in: Proceedings of the 5th International Workshop on Software Clones (IWSC'11), 2011, pp. 81–82.

[45] R. K. Saha, C. K. Roy, K. A. Schneider, gCad: A near-miss clone genealogy extractor to support clone evolution analysis, in: Proceedings of the International Conference on Software Maintenanc (ICSM'13), IEEE, 2013, pp. 488–491.

[46] E. Duala-Ekoko, M. P. Robillard, Clone region descriptors: Representing and tracking duplication in source code, Transactions on Software Engineering and Methodology (TOSEM'10) 20 (1) (2010) 1–31.

[47] 山中裕樹, 崔恩瀞, 吉田則裕, 井上克郎, 佐野建樹, コードクローン変更管理システムの開発と実プロジェクトへの適用, 情報処理学会論文誌 54 (2) (2013) 883–893.

[48] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, T. Sano, Applying clone change notification system into an industrial development process, in: Proceedings of the 21st International Conference on Program Comprehension (ICPC'13), IEEE, 2013, pp. 199–206.

[49] M. Sutton, A. Greene, P. Amini, Fuzzing: brute force vulnerability discovery, Pearson Education, 2007.

[50] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, An orchestrated survey of methodologies for automated software test case generation, Journal of systems and software 86 (8) (2013) 1978–2001.

[51] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, M. Woo, The art, science, and engineering of fuzzing: A survey, Transactions on Software Engineering (TSE'19) 47 (11) (2019) 2312–2331.

[52] P. Godefroid, N. Klarlund, K. Sen, DART: Directed automated random test-

ing, in: Proceedings of the Programming language design and implementation (PLDI'05), ACM, 2005, pp. 213–223.

[53] K. Sen, D. Marinov, G. Agha, CUTE: A concolic unit testing engine for C, Software Engineering notes (SEN'05) 30 (5) (2005) 263–272.

[54] P. Amini, Fuzzing framework, Black Hat USA 14 (2007) 211–217.

[55] T. Y. Chen, F.-C. Kuo, R. G. Merkel, T. Tse, Adaptive random testing: The art of test case diversity, Journal of Systems and Software 83 (1) (2010) 60–66.

[56] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, X. Xia, A survey on adaptive random testing, Transactions on Software Engineering (TSE'19) 47 (10) (2019) 2052–2083.

[57] 都築夏樹, 吉田則裕, 戸田航史, 山本椋太, 高田広章, カバレッジに基づくファジングツールの比較評価, コンピュータソフトウェア 39 (2) (2022) 2_101–2_123.

[58] 宮木龍, 吉田則裕, 藤原賢二, 都築夏樹, 山本椋太, 高田広章, Fuzz4B: ファジングツール AFL の利用支援ツール, コンピュータ ソフトウェア 39 (2) (2022) 2_124–2_142.

[59] A. Andoni, I. Piotr, Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions, Communications of the ACM 51 (1) (2008) 117–122.

[60] R. Koschke, S. Bazrafshan, Software-clone rates in open-source programs written in c or c++, in: Proceedings of the 23rd International Conference on Software Analysis, Evolution and Reengineering (SANER'16), Vol. 3, IEEE, 2016, pp. 1–7.

[61] Z. Li, S. Lu, S. Myagmar, Y. Zhou, CP-Miner: Finding copy-paste and related bugs in large-scale software code, Transactions on software Engineering (TSE'06) 32 (3) (2006) 176–192.

[62] C. K. Roy, J. R. Cordy, A survey on software clone detection research, Tech. Rep. 115, Queen University (2007).

[63] P. Thongtanunam, W. Shang, A. E. Hassan, Will this clone be short-lived? Towards a better understanding of the characteristics of short-lived clones, Empirical Software Engineering 24 (2019) 937–972.

[64] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, C. K. Roy, CCAligner: A token based large-gap clone detector, in: Proceedings of the 40th International Conference on Software Engineering (ICSE'18), ACM, 2018, pp. 1066–1077.

[65] S. Xie, F. Khomh, Y. Zou, An empirical study of the fault-proneness of clone mutation and clone migration, in: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13), IEEE, 2013, pp. 149–158.

[66] K. Yokoi, E. Choi, N. Yoshida, K. Inoue, Investigating vector-based detection

of code clones using BigCloneBench, in: Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC'18), IEEE, 2018, pp. 699–700.

[67] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, C. V. Lopes, SourcererCC: Scaling code clone detection to big-code, in: Proceedings of the 38th International Conference on Software Engineering (ICSE'16), ACM, 2016, pp. 1157–1168.

[68] N. Göde, R. Koschke, Incremental clone detection, in: Proceedings of the 13th European conference on software maintenance and reengineering (CSMR'09), IEEE, 2009, pp. 219–228.

[69] Y. Higo, U. Yasushi, M. Nishino, S. Kusumoto, Incremental code clone detection: A PDG-based approach, in: Proceedings of the 18th working conference on reverse engineering (WCRE'11), IEEE, 2011, pp. 3–12.

[70] 石津卓也, 吉田則裕, 崔恩瀞, 井上克郎, コードクローンのリファクタリング可能性に基づいた削減可能ソースコード量の分析, 情報処理学会論文誌 60 (4) (2019) 1051–1062.

[71] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, B. Maqbool, A systematic review on code clone detection, IEEE Access 7 (2019) 86121–86144.

[72] E. Choi, N. Fuke, Y. Fujiwara, N. Yoshida, K. Inoue, Investigating the generalizability of deep learning-based clone detectors, in: Proceedings of the 31st International Conference on Program Comprehension (ICPC'23), IEEE, 2023, pp. 181–185.

[73] S. Dou, J. Shan, H. Jia, W. Deng, Z. Xi, W. He, Y. Wu, T. Gui, Y. Liu, X. Huang, Towards understanding the capability of large language models on code clone detection: a survey, arXiv preprint arXiv:2308.01191.

[74] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, H. Wang, Large language models for software engineering: A systematic literature review, Transactions on Software Engineering and Methodology (TOSEM'24) 33 (8) (2024) 1–79.

[75] H. Jin, L. Huang, H. Cai, J. Yan, B. Li, H. Chen, From llms to llm-based agents for software engineering: A survey of current, challenges and future, arXiv preprint arXiv:2408.02479.

[76] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, J. M. Zhang, Large language models for software engineering: Survey and open problems, in: Proceedings of the International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE'23), 2023, pp. 31–53.

[77] B. van Bladel, S. Demeyer, Clone detection in test code: an empirical evaluation, in: Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER'20), IEEE, 2020, pp. 492–

500.

[78] W. Hasanain, Y. Labiche, S. Eldh, An analysis of complex industrial test code using clone analysis, in: Proceedings of the International Conference on Software Quality, Reliability and Security (QRS'18), IEEE, 2018, pp. 482–489.

[79] B. van Bladel, S. Demeyer, A comparative study of code clone genealogies in test code and production code, in: Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER'23), IEEE, 2023, pp. 913–920.

[80] R. Yokomori, K. Inoue, An empirical analysis of git commit logs for potential inconsistency in code clones, in: Proceedings of the International Conference on Source Code Analysis and Manipulation (SCAM'24), IEEE, 2024, pp. 1–12.

[81] Y. Semura, N. Yoshida, E. Choi, K. Inoue, CCFinderSW: Clone detection tool with flexible multilingual tokenization, in: Proceedings of the Asia-Pacific Software Engineering Conference (APSEC'17), IEEE, 2017, pp. 654–659.

[82] W. Zhu, N. Yoshida, T. Kamiya, E. Choi, H. Takada, Development and benchmarking of multilingual code clone detector, Journal of Systems and Software 219 (2025) 112–215.

[83] Y. Wu, Y. Chen, X. Peng, B. Hu, X. Wang, B. Fu, W. Zhao, CloneRipples: predicting change propagation between code clone instances by graph-based deep learning, Empirical Software Engineering 30 (1) (2025) 1–31.

[84] H. Xu, W. Ma, T. Zhou, Y. Zhao, K. Chen, Q. Hu, Y. Liu, H. Wang, A code knowledge graph-enhanced system for LLM-based fuzz driver generation, arXiv preprint arXiv:2411.11532.

付録

付録 1: パラメータ分析のための調査

本付録では，2.3.2 節でラベルとハッシュテーブル数 L がそれぞれ再現率と探索時間に与える影響を調べるために実施した予備実験の内容と結果を示す．実験環境は本手法の評価実験と同じ環境であり，約 15MLOC の Linux Kernel 4.19 をコードクローン検出対象とした．LSH ライブラリ FALCONN のパラメータの値を一定間隔で与え，CCVolti に適用してコードクローン検出を行い，パラメータごとの再現率と類似探索の探索時間を計測した．

ラベル $(K-1)\ln d + \ln T$

2.3.1 節で説明した通り，ハッシュ関数の数 K は区画の分割数 T の組をラベル $(K-1)\ln d + \ln T$ として扱う．ハッシュテーブルの数は $L = 1$ に固定し，ラベルの値を取りうる値の範囲で 1 刻みで変化させて実験を行った．ラベルが再現率と探索時間に与える影響を表すグラフを図 1 に示す．横軸はラベルの値を表す．再現率のグラフより，ラベルを増加させると再現率が減少していることがわかる．これは，L を固定しているとき，ラベルを増加すると式 2.7 より Cross-Polytope LSH の衝突確率が単調に減少し，それによって定理 1 から再現率の期待値が減少することと一致する．また，探索時間のグラフより，ラベルの値が 10 以下のとき探索時間が大幅に減少しており，10 以上のラベルでは探索時間の大きな変化がない．これは，区画の分割数が少ないとき，類似するベクトル対の候補が多く探索され，類似度を計算するベクトル対の数が大幅に増えるからだと考えられる．つまり，探索時間はラベルの値に依存して大幅に変化すると考えられる．

ハッシュテーブル数 L

ベクトル対が L 個のハッシュテーブルの内，いずれかのハッシュテーブルで衝突するとき，類似するベクトル対の候補となる．ラベルの値が 10 となるように $K = 1, T = 1024$ とし，ハッシュテーブルの数 L には 30 以下の自然数を与えて実験を行った．

ハッシュテーブル数 L が再現率と探索時間に与える影響を表すグラフを図 2 に示す．横軸はハッシュテーブルの個数を表す．再現率のグラフより，ハッシュテーブルの増加に従って，再現率が増加していることがわかる．これは，ラベルを固定してい

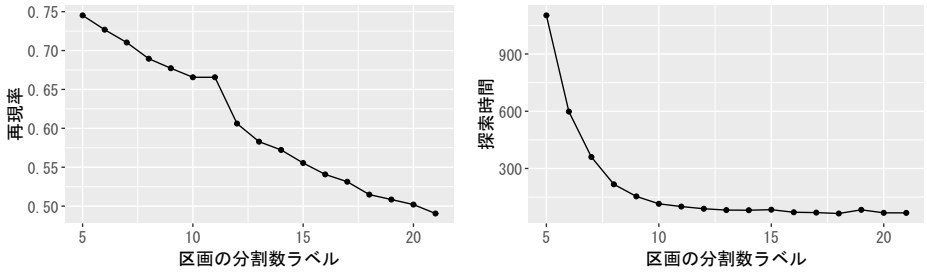


図 1　ラベル $(K-1)\ln d + \ln T$ と再現率や探索時間 [s] の関係

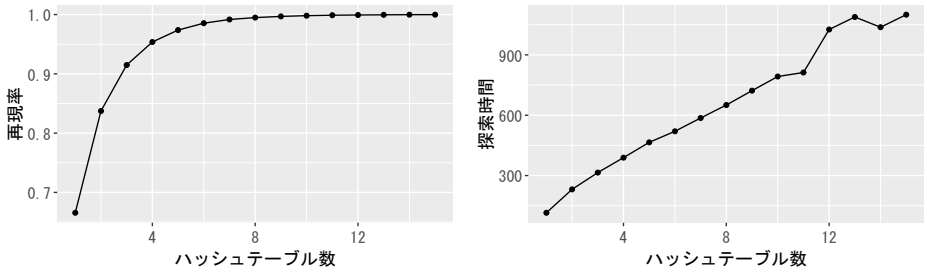


図 2　ハッシュテーブル数 L と再現率や探索時間 [s] の関係

るとき，ハッシュテーブルを増加すると式 2.7 より Cross-Polytope LSH の衝突確率が単調に増加し，それによって定理 1 から再現率の期待値が増加することと一致する．また，探索時間のグラフより，ハッシュテーブルの数を増加すると探索時間は線形に増加する．これは，ハッシュテーブルごとに，ハッシュ値の計算と衝突の判定処理をそれぞれ実行しているからだと考えられる．どのハッシュテーブルにも同じ T と K の値が与えられるため，式 2.5 より同じ衝突確率 $P_{T,K}$ での類似探索が実行される．つまり，L は他のパラメータに依存せず，探索時間を線形に増加させると言える．

付録 2: 本手法によるクローン検出時間の削減

本付録では，本手法を用いて決定したパラメータを用いた場合，実際の CColti でのコードクローン検出時間について調査するために，表 2.1 に示した実験対象である各プロジェクトごとに，各目標再現率に対してコードクローン検出時間を計測したグラフをそれぞれ図 3 から図 22 に示す．実験環境は 2.4.3 節に示した環境である．

グラフの横軸は 0.8 以上 0.99 以下の 0.01 刻みで与えた目標再現率の値を表し，縦軸は CCVolti のクローン検出時間 [s] を表す．棒グラフは目標再現率に対する CCVolti のクローン検出時間を示し，直線は，デフォルトのパラメータで CCVolti を実行した場合のクローン検出時間を示す．

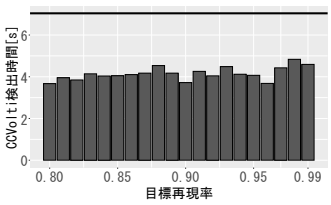


図 3　Antlr

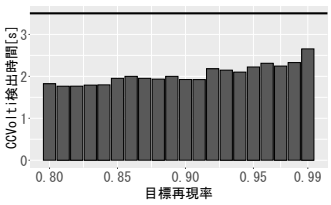


図 4　SNNs

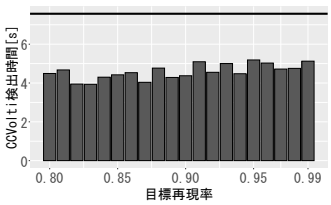


図 5　Maven

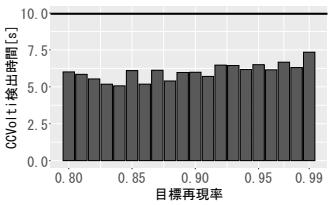


図 6　Ant

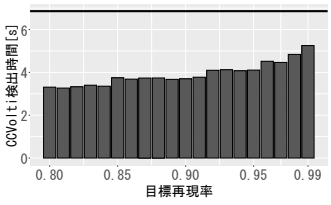


図 7　zfs-linux

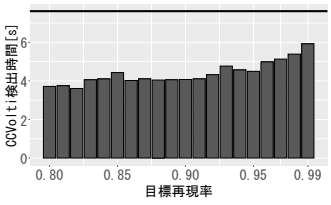


図 8　HTTPD

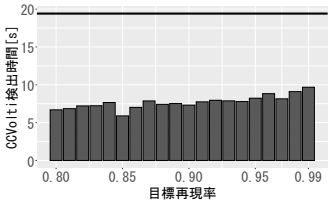


図 9　ArgoUML

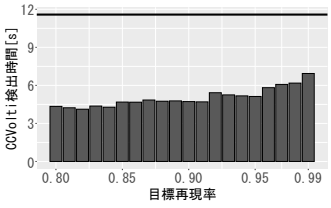


図 10　Python

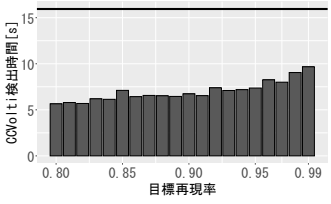


図 11　heimdal

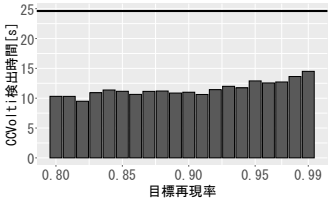


図 12　Pig

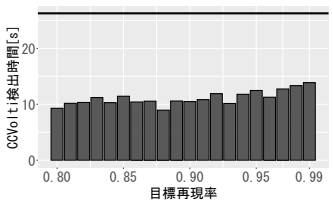


図 13　Tomcat

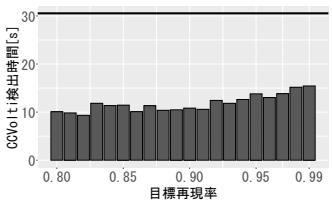


図 14　Jackrabbit

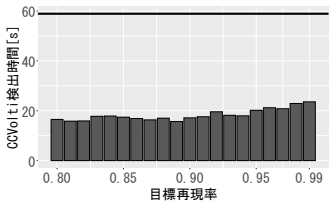


図 15　WildFly

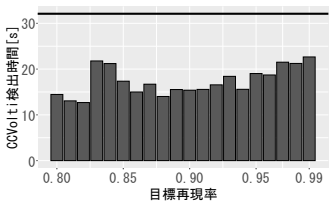


図 16　PostgreSQL

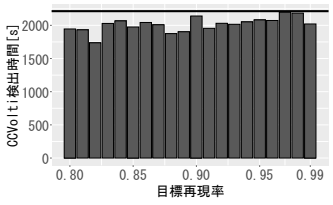


図 17　Camel

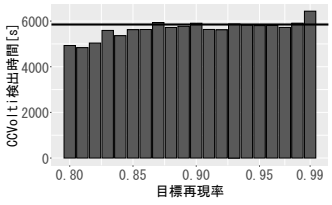


図 18　gcc

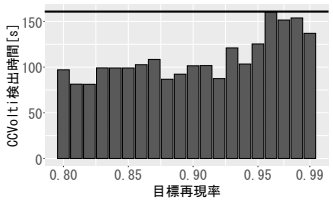


図 19　FireFox

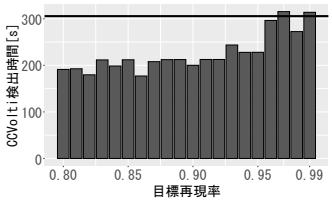


図 20　Jackrabbit

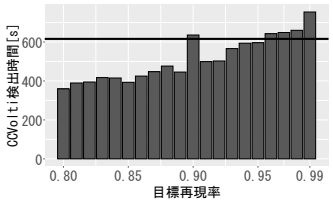


図 21　Linux Kernel

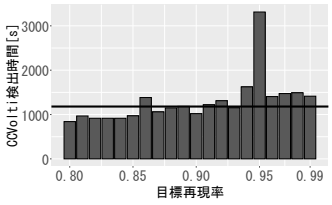


図 22　FreeBSD