特別研究報告

題目

C 言語を対象とした機能等価関数データセットの構築

指導教員
肥後 芳樹 教授

報告者
久名木 美香

令和 8 年 2 月 9 日

大阪大学 基礎工学部 情報科学科

令和 7 年度 特別研究報告

C 言語を対象とした機能等価関数データセットの構築

久名木 美香

内容梗概

近年，ソフトウェアの大規模化に伴い，保守性の確保は喫緊の課題となっている．特に大規模なコードベースには，リファクタリングや最適化の結果として，構文は異なるが機能が等価な機能等価クローンが多数混在している．これらは従来の静的解析技術では検出が困難であるため，仕様変更時の修正漏れを引き起こし，潜在的なバグの残留に繋がる．この問題を解決する新たな検出技術を発展させるためには，実世界の実装に基づいた高精度な評価用データセットが不可欠である．

そこで本研究では，C 言語のオープンソースソフトウェアを対象に，自動テスト生成を用いた動的解析により機能等価関数を抽出する手法を提案する．本手法は，対象関数に対してテストケースを生成し，その入出力挙動の一致を検証することで，記述構造に依存せずに機能的等価性を判定するものである．約 2.1 億行の C 言語ソースコードに本手法を適用した結果，最終的に 68 件の機能等価関数ペアを検出した．検出されたペアには，再帰とループの構造差や，配列操作とポインタ操作の違いなど，C 言語特有の実装パターンも含まれている．本研究で構築したデータセットは，従来の静的解析では検出困難な事例を含んでおり，コンパイラ最適化の挙動解析や将来的なクローン検出技術の評価基盤として寄与するものである．

主な用語

機能等価関数
コードクローン
動的解析

目 次

1　はじめに

近年，社会基盤としてのソフトウェアの重要性は増すと同時に，その大規模化と複雑化は急速に進行している．これに伴い，開発後のソフトウェアの保守性や信頼性を長期間にわたって確保することは，ソフトウェア工学における喫緊の課題となっている．実際の開発現場では，同一の機能要件に対し，異なるアルゴリズムやデータ構造が選択されるケースや，リファクタリングによってコードの内部構造が改善されるケースが多々見られる [12]．その結果，大規模なコードベース内には，構文的には異なるが機能的に等価なコードクローンが潜在的に多数存在することになる．システム内にこれらのコードクローンが存在する場合，仕様変更の際に一貫した変更が求められる．また，保守性の観点からは，異なる実装をより効率的な実装へと統一することが望ましい．しかし，構文的に異なるクローンは従来の検出手法では発見が困難である．その結果，システム内に仕様の不一致や潜在的な欠陥が残留するリスクがある．このような問題を解決する検出技術の発展には，検出困難なクローンの実例に基づいた評価用データセットの存在が不可欠である．したがって，コードクローンの収集と分析は，効率的な実装パターンの発見や，高度な検出技術の性能評価において極めて重要な役割を担っている．

コードクローンは，その類似度に応じて Type-1（完全一致）から Type-4（意味的一致）まで分類される [15]．中でも，Type-4 クローンはソースコードの記述構造は異なるが同一の機能を持つコード片を指し，実装の多様性を分析する上で極めて有用な情報源となる．例えば，同一仕様に対するアルゴリズムの差異や，実行速度向上のための最適化コードなどは，自動プログラミングやコード補完技術の発展に寄与する重要なデータである．

しかし，従来のコードクローン検出技術には本質的な限界が存在する．既存ツールの多くは，トークン列や抽象構文木といった構文的特徴の類似性に依存しているため [8][9]，変数の違いや軽微な変更には対応できても，制御構造が根本的に異なるクローンを検出することは困難である．特に C 言語は，ポインタ演算やメモリ管理の自由度が高く，開発者のスキルや最適化の意図によって実装の多様性が顕著になる傾向がある．したがって，C 言語における 同機能異構造コードの収集は，コンパイラ最適化や自動修正技術の発展において高い価値を持つ．しかしながら，既存ツールを用いてデータセットを作成しようとすると，ツールが検出しやすい構文的に類似したコードばかりが集まり，本来収集すべき C 言語特有の多様な同機能異構造コードが欠落するという問題が生じる．現状，C 言語を対象とした網羅的な同機能異構造コードのデータセットは十分に整備されていない．

本研究ではこの課題を解決するため，C 言語プログラムを対象とし，自動テスト生成技術を用いて同機能異構造コードを収集する手法を提案する．具体的には，ソースコードから抽出した関数群に対し，自動生成したテスト入力を適用し，その実行結果を比較することで機

能等価性を検証する．本手法は，コードの記述形式ではなく，実行時の入出力の一致を直接的な判定基準とするため，従来の静的解析技術では捉えきれなかった同機能異構造コードの検出を実現する．

本研究の貢献は，約 2.1 億行の C 言語ソースコードを対象とした大規模な調査により，最終的に 68 件の同機能異構造コードペアを含むデータセットを構築した点にある．本データセットには，再帰とループの構造差や，配列操作とポインタ操作の違いなど，C 言語特有の実装パターンが多数含まれている．これらのコードペアは，従来の静的解析技術では検出が困難であった実例であり，コンパイラ最適化の挙動解析や自動修正技術の評価・発展に向けた基盤を提供する．

本論文の構成は以下の通りである．第 2 章では，研究の背景として，コードクローン検出における既存技術の課題について述べ，特に C 言語を対象とした同機能異構造コードペアのデータセットが不足している現状について詳述する．第 3 章では，提案する同機能異構造コードペアの収集手法について述べる．具体的には，テストケースの自動生成から相互テストの実行による等価性判定に至るまでの一連の処理フローについて詳細に説明する．第 4 章では，提案手法を用いて構築されたデータセットの概要を説明する．収集された関数の規模や，特定されたペアの統計的な内訳について述べる．第 5 章では，構築されたデータセットに対する考察を行い，提案手法の有効性と課題について議論する．第 6 章では，本研究に関連する既存研究について概説し，本研究の位置付けと独自性を明らかにする．最後に，第 7 章で本論文を総括し，今後の課題と展望について述べる．

2　研究の背景

2.1　機能等価性の定義

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，あるいは類似したコード片のことである．ソフトウェア開発においては，開発効率の向上や既存機能の再利用を目的として，コードを複製することがある．その結果として，大規模なコードベース内には多数の類似したコード片が生成されることになる．これらコードクローンの存在は，一方のコードにバグが発見された際，対応する全てのコード片を特定して修正する必要が生じるため，ソフトウェアの保守性を低下させる要因となる．

一方で，ソースコードの複製を伴わずにクローンが発生するケースも存在する．例えば，同一の仕様に対して異なる開発者がそれぞれ独自のアルゴリズムで実装を行った場合や，保守性向上を目的としたリファクタリングによって内部構造のみが改善された場合である．このようにして生成されたクローンは，構文は異なるが振る舞いは等価である．コードクローンは，修正漏れによるバグの誘発や保守コストの増大など，ソフトウェア品質に多大な悪影響を及ぼす恐れがあるため，その適切な管理と分析が不可欠である．

ソフトウェア工学において，コードクローンはその類似性により，一般的に以下の 4 種類に分類される [15].

Type-1 クローン： 空白，改行，コメントの差異を除き，完全に一致するコード片

Type-2 クローン： Type-1 クローンに加え，変数名，型名などの識別子やリテラル値の変更を許容するコード片

Type-3 クローン： Type-2 クローンに加え，ステートメントの追加，削除，変更などの変更を許容するコード片

Type-4 クローン： 構文的特徴は大きく異なるが，機能が等価であるコード片

Type-1 から Type-3 クローンはいずれも構造的な類似性に基づいているのに対し，Type-4 クローンは振る舞いの等価性に基づいているという特徴がある．本研究では，この Type-4 クローンに該当する関数同士を機能等価関数ペアと呼ぶ．具体的には，同一のシグネチャを持つ 2 つの関数に対し，同一の入力を与えた際の出力および外部状態への副作用が一致する場合，これらを機能等価であるとみなす．

機能等価関数ペアの典型例としては，バブルソートとクイックソートのように，計算量や内部処理は異なるが同一の仕様を満たす実装や，リファクタリングによって内部構造が変更される前後のコードなどが挙げられる．これらは単なる重複コードではなく，同一の仕様に対する実装の多様性を示す，有用な情報源である．例えば，アルゴリズムの差異や最適化

コードの比較分析は，自動プログラミングやコード補完技術の発展に寄与すると考えられる．しかし，このようなコード片は従来の静的解析では検出が困難とされている．これは，静的解析手法がコードの表面的な記述の一致に着目しており，実行時の振る舞いまでは考慮しないためである．

2.2 既存の検出技術における課題

機能等価関数ペアを収集するためには，機能的な等価性を正確に判定する必要がある．しかし，既存の静的解析に基づくアプローチには限界が存在する．CCFinder [9] や NiCad [2], Deckard [8] をはじめとする従来のコードクローン検出ツールの多くは，トークン列や抽象構文木（AST）の類似性に基づいてコードを比較する．こうした既存のクローン検出ツールを用いてデータセットを作成しようとする試みには重大な課題が存在する．これらの手法は，Type-1 から Type-3 に分類されるクローンに対しては高い検出精度を誇るため，結果として検出可能な構文的に類似したコードが多く収集されてしまう一方で，ツールが苦手とする構造が異なる機能等価コード，すなわち Type-4 クローンが含まれないという問題がある．また，ソースコードから機能等価関数を手作業で見つけ出すことは，膨大な工数を要するため現実的ではない．以上のように，既存の検出技術や手作業によるアプローチでは，多様な実装構造を持つ機能等価関数ペアを効率的に収集することは難しい．

この課題に対し，近年では機械学習を用いた検出手法が提案されている [19]．しかし，それらの手法を学習・評価するためには，機能的な等価性が保証されたデータセットが必要不可欠である．

2.3 C 言語データセットの不足

機能等価コードのデータセットに関しては，Java を対象とした BigCloneBench [16] や，多言語対応の CodeNet [14] などが存在する．しかし，C 言語を対象とした既存のデータセットには偏りや不足が見られる．

具体的には，POJ-104 [13] や Codeforces-500（CF-500）[17] を含む多くの既存データセットは，プログラミングコンテストで提出されたコードを収集したものである．これらのコードは，標準入出力を用いて単一の課題を解決する特殊な形式をとっており，関数呼び出しや複雑なメモリ管理を含む実世界のソフトウェアにおける実装パターンとは性質が大きく異なる．加えて，OSS から収集されたデータセットの多くは，コメントや識別子の類似度に基づくヒューリスティックな基準で構築されており，実際にコードを実行して機能等価性を検証したものは言語を問わず少ないという課題がある [11].

2.4 Java を対象とした機能等価メソッドの収集の先行研究

ソースコードを動的解析することにより機能等価性を判定する試みは，これまでにも行われている．特に Higo らは，Java を対象に自動テスト生成ツール EvoSuite を用いた機能等価メソッドの収集手法を提案した [6]．彼らの手法は，抽出したメソッドに対してテストケースを生成し，それらを相互に実行することで，入出力関係に基づき等価性を判定するものである．このアプローチにより，従来の手法では検出困難であった，構文が全く異なるが機能が等価なメソッドペアの収集に成功している．

2.5 C 言語特有の技術的障壁

本研究では，前節で述べた動的解析アプローチを C 言語の大規模コードベースに適用することを目指す．しかし，このアプローチを C 言語に適用する場合，言語仕様に起因する固有の障壁が存在する．

第一に，コンパイル単位と依存関係の複雑さである．Java と異なり，C 言語では関数の型情報や外部シンボルが，ヘッダファイルおよびリンク処理を通じて解決される．そのため，関数を単体で切り出した場合，依存関係を満たしたコンパイル可能な状態を構成することが一般に困難である．第二に，ポインタと未定義動作である．`void*`型やメモリ操作を含む関数は，自動テスト生成において考慮すべきパターンが膨大になったり，実行時エラーを引き起こしたりするリスクが高い．これらの要因により，C 言語のソースコードに対して動的解析を適用し，機能等価性を検証することは，従来困難とされてきた．

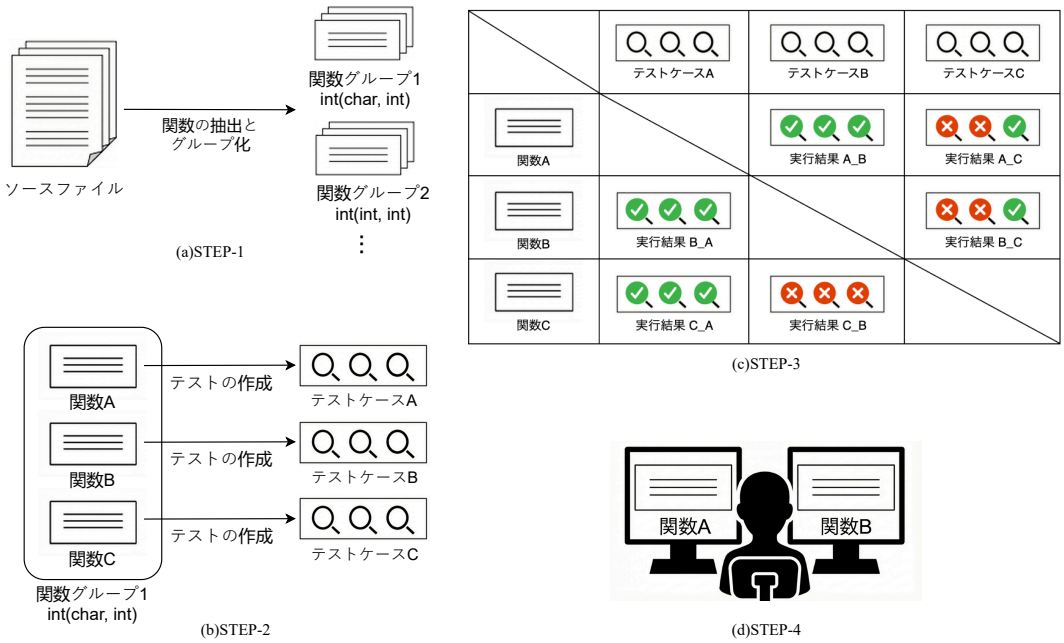

図 1: データセット構築の手順概要

3　データセットの構築手順

本章では，C 言語で記述された大規模なコードベースから，機能等価関数ペアを自動的に収集するための手法について述べる．次の 4 つのステップにより，機能等価関数ペアのデータセットを構築する．

STEP-1 ：ソースコードからの関数の抽出とフィルタリング

STEP-2 ：各関数に対するテストケースの生成

STEP-3 ：テストケースの相互実行による機能等価関数ペア候補の絞り込み

STEP-4 ：目視確認による真の機能等価の判定

3.1　STEP-1

STEP-1 では機能等価性を検証するための関数ペアを収集するにあたり，まずは比較の対象となり得る関数を大規模なコードベースから抽出する必要がある．本研究では，公開されているデータセット Raw C Code Corpus [10] に含まれる C 言語のソースコードを対象とし，関数の抽出，フィルタリング，および分類を行った．このデータセットは約 2.1 億行あり，722,364 のファイルからなる．

3.1.1 関数の抽出

抽出処理には，本研究のために作成した独自のスクリプトを用いた．このスクリプトは，対象となるソースファイルを走査し，関数定義の開始と終了を解析することで，個々の関数を独立したコード片として切り出す．データセット内には，プロトタイプ宣言やマクロ定義を含むファイルも混在している可能性がある．そのため，本スクリプトでは戻り値の型，関数名，引数リスト，および関数本体から構成される完全な関数定義のみを抽出対象とした．抽出された各関数は，STEP-2 のテスト生成処理のために一時的に個別のファイルとして保存した．なお，単に関数部分を切り出しただけの状態では，必要なヘッダファイルのインクルード記述や型定義が欠落しており，単体でのコンパイルは不可能である．そのため，次項で述べるフィルタリングと並行して，コンパイル可能なソースファイルへの再構成を行った．

3.1.2 フィルタリングとソースファイルの再構成

抽出された関数は多くの場合，外部の依存関係や複雑なコンテキストを必要とするため，単体でのコンパイルや実行が困難である．本研究では，後の工程であるテストケース生成を効率的かつ正確に行うため，外部依存を持たず自己完結している関数のみを抽出対象とした．また，テスト生成における処理の複雑化を避け，データセット構築の成功率を高めるために関数の引数や戻り値の型に対しても制約を設けた．具体的には，以下の条件を全て満たす関数を対象とした．

- 標準ライブラリ以外の関数を呼び出さない関数
- 標準ライブラリ以外のヘッダファイルを参照していない関数
- 引数を持ち，戻り値が `void` 型でない関数
- 引数や戻り値に `void*`を含まない関数

1 つ目の条件を用いる理由は，関数の振る舞いを完全に制御・観測するためである．他のユーザー定義関数や外部ライブラリの関数の呼び出しを含む場合，その関数の実行には呼び出し先の定義や適切なリンクが必要であり，コンパイルの事前準備が不可欠となる．これはテスト生成の自動化プロセスを著しく複雑化させる要因となる．そのため，本研究では対象関数内のロジックのみを評価対象とした．

2 つ目の条件を用いる理由は，解析環境における依存性を最小化するためである．独自のヘッダファイルに依存する関数は，特定のプロジェクト構造やビルド環境が整っていなければ解析が困難である．対象を標準ライブラリのみに限定することで，実行環境に左右されない汎用的なデータセット構築を可能にしている．

3 つ目の条件を用いる理由は，テスト入力の生成および実行結果の検証を，入出力関係に基づいて直接的かつ厳密に行うためである．引数を持たない関数や戻り値が `void` 型の関数は，多くの場合，グローバル変数の参照や更新といった関数外部の状態に依存した処理を主たる機能としている．これらの挙動を評価するには実行前後のメモリ状態の比較など高度な解析手法を要し，等価性判定の信頼性を十分に確保することが難しくなる．本手法では，入力値の組み合わせに対する戻り値の一致を機能等価性の判定基準とするため，明示的な引数を持ち，かつ戻り値が `void` 型でない関数のみを対象とした．

4 つ目の条件を用いる理由は，型の曖昧さを排除し，テストデータの自動生成の成功率を向上させるためである．`void*` 型はコンパイル時に型が確定せず，実行時のキャストによって初めて具体的なデータ構造として解釈されるものであり，テスト生成手法において，有効な入力値を自動で構成することが困難である．解析精度と自動化の安定性を優先し，型情報が静的に明示されている関数のみを採用した．

以上のようにして抽出・フィルタリングされた関数をデータセットとして保存する際，各関数が単体でコンパイル可能となるよう，ソースファイルの再構成を行った．具体的には，元のソースコード内で記述されていた標準ライブラリへの include ディレクティブを保持し，抽出された関数の先頭に配置した．さらに，基本的な型定義や入出力機能を利用可能にするため，表 1 に示す，一般的な標準ヘッダファイルのインクルード記述をすべてのファイルに一律で付与した．図 2 は本手法による対象関数の抽出およびソースコード整形の具体例である．図 2(a) に示す元のソースコードは，収集元となったソースファイルの一部である．関数の宣言以降のみを抽出してしまうと，ファイルの先頭に記述されていたインクルード文が引き継がれない．その結果，関数内で利用されている型やマクロの定義が参照できなくなり，単体でのコンパイルが不可能となる場合がある．これに対し，本手法を適用した (b) では，第 3 行から第 15 行の赤字部分に示す通り，解析に不可欠な標準ヘッダファイルを補完している．また，第 17 行から第 30 行の青字部分のように元ファイルの冒頭にあったインクルード文を保持している．

本前処理により，抽出された各関数ファイルは外部依存の問題を解決し，次のステップであるテスト生成ツールによる解析が即座に可能な状態となっている．

3.1.3 変数と戻り値に基づく分類

上記のフィルタリングを経て抽出された関数群は，機能等価性の比較候補ペアを作成するために分類される．本研究では，関数の戻り値の型および引数のリスト（数と型）の組み合わせをシグネチャと定義し，シグネチャが完全に一致する関数同士を同一のクラスタに分類した．これにより，機能が異なると思われる関数同士の比較を避けることができ，計算コストの削減に繋がった．

```
 57 #include <stdio.h>
  :
173 float Exp10(int n)
174 {
175         int i;
176         float result = 1;
177         for(i =n; i; i--)
178                 result *= 10;
179         return result;
180 }
```

(a) 元のソースコード

```
 3 // Essential headers
 4 #include <stddef.h>
 :
15 #include <assert.h>
16 // Original include statements
17 #include <stdio.h>
 :
30 #include <time.h>
31 float Exp10(int n)
32 {
33      int i;
34      float result = 1;
35      for(i =n; i; i--)
36              result *= 10;
37      return result;
38 }
```

(b) 抽出されたソースコード

図 2: 対象関数の抽出処理の例

表 1: 追加する標準ヘッダファイルとその機能

ヘッダファイル名	主な宣言内容
stddef.h	size_t や NULL といった標準的な型およびマクロの宣言
stdbool.h	bool 型や真理値を表す true および false の宣言
stdint.h	int32_t をはじめとするビット幅を指定した整数型の宣言
limits.h	整数型が表現可能な最大値や最小値といった制限値の宣言
string.h	文字列の操作やメモリ領域の操作を行う関数の宣言
stdlib.h	メモリ確保や数値変換を含む標準的なユーティリティの宣言
stdio.h	標準入出力に関連する機能の宣言
ctype.h	文字の分類や変換に関する機能の宣言
assert.h	プログラムの実行時診断を行うアサーション機能の宣言

3.2 STEP-2

STEP-2 では STEP-1 で抽出した各関数に対し，その機能的振る舞いを捉えるためのテストケースを生成する．本研究では，C 言語に対応した単体テスト自動生成ツールである UTBot [7] を採用した．

機能等価性を正しく判定するためには，関数のロジックを網羅的に実行するテスト入力が必要となる．UTBot はシンボリック実行エンジンである KLEE [1] を内部で利用しており，プログラムの実行パスを数式として扱い，分岐条件を満たす入力値を逆算することができる．これにより，複雑な条件分岐を持つ関数に対しても，高いコードカバレッジを持つテストケースを効率的に生成できるため，本研究の目的に適しているといえる．

STEP-1 で再構成された各ソースファイルを入力として，UTBot を用いたテストコードの自動生成を行った．解析に成功した場合，UTBot は Google Test [4] フレームワーク形式の C++ ソースファイルを出力する．Google Test は Google によって開発されたテスティングフレームワークであり，期待される出力と実際の実行結果を比較するためのアサーション機能や，複数のテストケースを統合的に実行するための実行環境を提供する．本研究では，生成されたテストコードに含まれる EXPECT_EQ 等のマクロを用いることで，異なる関数間に挙動の差異がないかを厳密に検証する．

UTBot による自動生成の過程において，一部の関数ではツールのタイムアウトやビルドエラー等により，正常にテストファイルが生成されない事例が確認された．本研究では，テストコードの生成に失敗した関数は解析不能と判断し，データセットから除外した．この除外処理に伴い，同一のシグネチャに分類されていた関数の数が減少し，有効な関数が 1 つしか残らないケースが発生した．次ステップで行う相互テストは，同一シグネチャ内の異なる関数間での比較を行う手法であるため，比較対象が存在しない単独の関数では検証を行うことができない．したがってテスト生成に成功した関数が 1 つ以下となったシグネチャについても，機能等価性の判定が不可能であるため，クラスタごと削除した．

3.3 STEP-3

STEP-3 では STEP-2 で生成されたテストケースを用い，同一クラスタ内に分類された関数のすべての組み合わせに対して相互テストを実施する．これにより，挙動が一致する機能等価関数ペアの候補を特定する．図 1(c) では，関数 A，関数 B，関数 C について相互テストを行っている．テストケース A，テストケース B，テストケース C は図 1(b) のように，それぞれ関数 A，関数 B，関数 C から生成されたテストケースである．例えば関数 A と関数 B に着目すると，関数 A は関数 B から生成されたテストケースを全て通過しており，同様に関数 B も関数 A のテストケースをすべて通過している．このように双方向のテストに成

功した場合，関数 A と関数 B は機能等価なペアの候補として判定される．一方，関数 C は関数 B のテストケースの一部で失敗している．このようにいずれか一方でもテストに通過しなかった場合，そのペアは機能等価関数ペアの候補から除外される．

相互テストを自動化するため, 本研究では C++の単体テストフレームワークである Google Test を利用した．

最終的に，それぞれのクラスタ内で双方向のテストを全てパスしたペアのみを抽出し，その情報を CSV 形式のリストとして保存した．

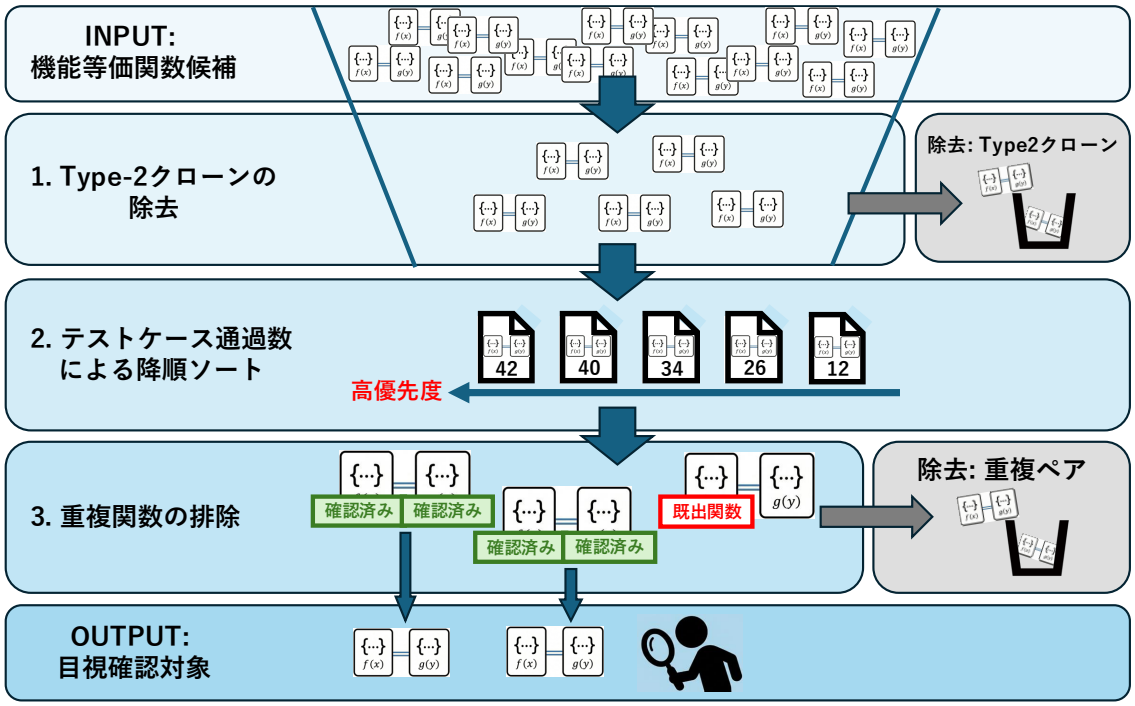

図 3: STEP-4 の処理フロー

3.4 STEP-4

STEP-4 では STEP-3 で保存した機能等価関数候補のソースコードを目視で確認し，真に同じ振る舞いをするかどうかを判定する．本研究では，実装は異なるが機能が等価なペアである Type-4 クローンの収集を目的としている．そのため，自明な一致や重複を含む全ての候補を無作為に確認するのではなく，以下の手順により検証対象の選別を行い，効率的に目視確認を実施した．図 3 に本ステップの処理フローを示す．

1. **Type-2 クローンの除去** ：STEP-3 までの工程で得られた機能等価候補ペアには，関数名や変数名は異なるものの，実装構造が完全に同一であるペア，すなわちコードクローンが含まれている可能性がある．本研究の目的は，実装は異なるが機能が等価であるペアを収集することであるため，このような自明な一致は除去する必要がある．そこで，本研究ではコードクローン検出ツールである NiCad を用い，候補ペアの中から Type-2 クローン（識別子は異なるが構造が同一なクローン）に該当するものを検出し除去した．

2. **テストケース通過数による降順ソート** ：残った候補ペアを，通過したテストケース数

の降順に整列した．これは，テストケース通過数が多いほど，関数ペアが機能等価である可能性が高いと考えられるためである．

3. **重複関数の排除と対象ペアの確定** ：整列後の上位のペアから順に走査し，ペアを構成する両方の関数が，いずれも過去に目視確認対象として選定されていない場合に限り，当該ペアを確認対象とした．具体的には，以下のルールに従って同一関数が複数のペアに重複して含まれることを防いだ．
   (a) ペアを構成する関数 f_A と f_B がいずれも過去に確認済みとして記録されていない場合，そのペアを確認対象とする．
   (b) f_A と f_B のいずれかが確認済みとして記録されている場合，そのペアはスキップする．
   (c) 対象として選定された場合，その場で f_A と f_B のシグネチャを記録し，以降は確認済みとして扱う．

選定された関数ペアに対して，著者が実際のソースコードを目視で確認し，アルゴリズムおよび処理ロジックが本質的に等価であるかを判定した．この判定プロセスを経て，最終的な機能等価関数ペアからなるデータセットを構築した．

4　実験結果

本章では，構築された機能等価関数ペアデータセットの詳細について述べる．

4.1　データセットの統計情報

第 3 章で述べた提案手法を，公開データセット Raw C Code Corpus [10] に適用した．このデータセットは合計約 2.1 億行のソースコードからなり，約 1,009 万関数を含んでいる．各工程におけるフィルタリングの結果および最終的に得られた関数ペアの統計情報を表 2 に示す．

STEP-1 でフィルタリングを行った結果，96,791 の関数が抽出され，2,824 のシグネチャに分類された．続いて STEP-2 では UTBot [7] を用いて 7,384 の関数から合計 20,084 のテストケースが生成された．テスト生成に成功した各関数に対し，同一シグネチャ内での組み合わせを行った結果，次ステップにおける等価性検証の対象ペアは合計 137,028 件となった．

STEP-3 では，双方向のテストを通過し，機能等価候補として残存したペアは 4,748 件であった．さらに STEP-4 において，NiCad [2] を用いた Type-2 コードクローンの除去を行った結果，候補ペアは 2,368 件まで絞り込まれた．これに対し，目視確認の効率化と重複排除を適用した結果，最終的な目視確認対象は 208 件となった．

抽出された 208 件の候補ペアに対し，著者による目視確認を実施した．確認作業には合計で 11 時間 21 分を要した．ソースコードの論理的な等価性を検証した結果，機能的に等価であると判定されたペアは 68 件，機能的に等価でないと判定されたペアは 140 件であった．

表 2: 各工程におけるフィルタリング結果と関数ペア数

工程	項目	件数
データセット本体	関数総数	10,087,637
STEP-1	抽出された関数	96,791
STEP-2	テスト生成成功関数	7,384
STEP-2	比較候補ペア	137,028
STEP-3	テスト通過ペア	4,748
STEP-4	非 Type-2 ペア	2,368
STEP-4	目視確認対象ペア	208
最終結果	**機能等価関数ペア**	**68**

4.2 判定事例

本手順による目視確認において，機能等価であると判定されたペアおよび非等価と判定されたペアの具体例を示す．

図 4 に, 機能等価と判定された関数ペアの例を示す. 図 4(a) の `biggishint_internal_bitsize` および (b) の `bitstream_need_bits_unsigned` は, いずれも引数として与えられた `unsigned long` 型の整数 `n` のビット長を算出する関数である．実装の詳細を見ると，(a) では `for` ループの更新式内でシフト演算 `n >>= 1` を行っているのに対し，(b) ではループ本体内で同一の演算が記述されている．そのため，両者は制御構造や使用される変数名において構文的に異なる実装となっている．しかし，アルゴリズムの本質はいずれも値が 0 になるまで右シフトを繰り返すという点で一致しており，同一の入力に対して常に同一の出力を返すため，これらを真の機能等価ペアと判定した．

一方，図 5 は機能等価ではないと判定されたペアの例である．(a) の `g2` は入力された `double` 型の値を単純に `long` 型へキャストして返却する関数である．これに対し，(b) の `fifidint` は同様のキャスト処理を行うものの，`if (a < 1.0)` という条件分岐を含んでおり，入力が `1.0` 未満の場合には必ず `0` を返す仕様となっている．例えば，負の値である `-5.0` を入力した場合，(a) は `-5` を返す一方で，(b) は `0` を返すため，振る舞いは等価ではない．このように，部分的に処理が類似していても，特定の入力範囲において出力が異なるペアについては非等価として扱った．

```
int biggishint_internal_bitsize(unsigned long n)
{
    int bitsize = 0;
    for ( ; n; n >>= 1)
        ++bitsize;
    return bitsize;
}
```

(a) 関数 biggishint_internal_bitsize

```
int bitstream_need_bits_unsigned(unsigned long n) {
    register int i;
    for (i = 0 ; n ; i++) {
        n >>= 1;
    }
    return i;
}
```

(b) 関数 bitstream_need_bits_unsigned

図 4: 機能等価な関数ペアの例

```
long g2 (double f)
{
  return f;
}
```

(a) 関数 g2

```
long fifidint(double a)
/* a - number to be truncated */
{
  if (a < 1.0) return (long) 0;
  else return (long) a;
}
```

(b) 関数 fifidint

図 5: 機能等価でない関数ペアの例

5　考察

本実験において，収集対象とした関数の総数に対し，最終的に機能等価であると判定された関数ペアの数は想定よりも限定的であった．例えば，Higo らによる Java を対象とした研究では，257,012 関数から機能等価と判定されたメソッドペアは 1,342 件であり，検出率は約 0.52% である．対して本研究では，96,791 関数から機能等価と判定された関数ペアは 68 件であり，検出率は約 0.07% である．同一のアプローチを採用しているにもかかわらず，Java と比較して C 言語における検出率が著しく低下した要因として，主に以下の 3 点が考えられる．

1. **C 言語 OSS におけるコード流用傾向**：本実験の結果において特筆すべき点は，Type-1 クローン（完全一致）や Type-2 クローン（識別子の変更）に相当するペアが多数検出された一方で，実装構造が大きく異なる機能等価ペアの割合が低かったことである．これは，C 言語の OSS 開発において，既存の実装が利用可能な場合には，再実装よりも既存コードの直接的な流用や，軽微な修正による再利用が選択されやすいことが背景にある．また，C 言語は低レイヤの処理を担うことが多く，特定の機能に対して効率的な実装パターンが広く共有されている場合が多い．その結果，機能等価であれば実装構造も類似しやすく，機能は同一であるが構造が全く異なる Type-4 に該当するケースが相対的に発生しにくかったと推察される．

2. **C 言語におけるコンパイルおよび依存関係解決の困難性**：これが検出率低下の最大の要因であると推察される．Java 等の言語と比較して，C 言語は関数単体を切り出して実行可能な状態にするための技術的障壁が高い．Higo らの手法が対象とした Java では，クラスファイル内に型情報やシンボル定義が保持されており，JVM の動的リンク機構によってクラスパス上の依存関係が自動的に解決される．そのため実行環境の構築が比較的容易である．対して C 言語は，コンパイルが成功しても，リンク時に未定義参照のエラーが多発する特性がある．これは，ヘッダファイルによる宣言と，ソースコードによる定義が分離している言語仕様に起因する．本手法では，この言語仕様上の制約により，テスト実行の段階で脱落する関数が Java と比較して著しく多く，その結果，機能等価性の判定まで到達できた候補ペアの母数が大幅に削減された．このことが，最終的な検出率の低さに直結したと考えられる．

3. **入出力一致の厳格な判定基準**：本手法では，ランダムテストに対する戻り値および副作用が一致することを求めている．Type-4 クローンの中には，主要な入力範囲では同一の振る舞いを示すが，境界値に対する挙動のみが異なるケースも存在する．本手法はこれらを機能等価ではないと判定して排除するため，見かけ上の検出数は減少する．

これは，偽陽性を排除し，厳密な意味で機能等価な関数のみを抽出できたという点で，データセットの品質向上に寄与しているといえる．

6　関連研究

本章では，機能等価コードに関する既存のデータセットおよび構築手法について概説し，本研究で構築したデータセットとの比較を行う．

6.1　機能等価コードのデータセット

コードクローン検出技術の評価において，機能等価性を含むベンチマークは不可欠である．その代表例として，Svajlenko らによって構築された BigCloneBench [16] が挙げられる．BigCloneBench は Java のオープンソースソフトウェアから構築された大規模データセットであり，Type-4 クローンを含む点が特徴である．その一方で，その構築過程はキーワード検索やヒューリスティックに依存しており，実行結果に基づく動的検証は行われていない．Krinke らは，BigCloneBench の Type-4 ペアの一部が実際には機能等価ではない可能性を指摘しており，実行検証を伴わない手法で機能等価性を正確に判定することの困難さが示唆されている [11].

一方，C 言語を含むデータセットとしては，POJ-104 [13] や CodeNet [14] のデータを用いたものが存在する．これらはプログラミングコンテストの提出コードを収集したものであり，含まれるコードの多くは特定のアルゴリズム課題を解決するために記述されたものである．そのため，実世界の大規模ソフトウェア開発において広く用いられているユーティリティ関数やデータ処理ロジックを必ずしも十分に反映したものとは言えない．

これらに対し，本研究で構築したデータセットは，C 言語の実用的なコードを対象とし，かつ自動テスト生成による動的検証を経ている点で独自性を持つ．これにより，既存のデータセットではカバーできない，実世界の実装パターンを含む機能等価関数ペアを提供している．

6.2　機械学習および事前学習モデルを用いたコードクローン検出

近年，深層学習技術の発展に伴い，ソースコードの構文情報や意味的特徴を学習することで，高精度なクローン検出を試みる手法が多数提案されている．これらは従来のトークン列比較やテキストベースの手法と比較して，より抽象度の高い類似性を捉えることが可能であり，Type-3 や Type-4 クローンの検出において一定の成果を上げている．

初期の研究として，Jiang らは抽象構文木に基づく特徴ベクトルを生成し，ユークリッド距離に基づくクラスタリングによりクローンを検出する Deckard を提案した [8]．また，Zhang らは，AST をステートメント単位の部分木に分割し，各部分木をベクトル化して RNN に入力する ASTNN (AST-based Neural Network) を提案した [18]．ASTNN は構文構造と文脈情報を統合的に学習でき，従来の AST ベース手法より高い分類性能を示している．

さらに近年では，自然言語処理分野で成功を収めた Transformer アーキテクチャを応用し，大規模なコードコーパスで事前学習を行ったモデルが主流となりつつある．代表的なモデルである CodeBERT [3] は，ソースコードと自然言語のペアを用いた事前学習により，コードの文脈情報を反映した表現を獲得している．CodeBERT や，データフロー情報を組み込んだ GraphCodeBERT [5] などのモデルは，クローン検出を含む様々なコードインテリジェンスのタスクにおいて，既存の機械学習手法と比較して高い性能を示している．

しかしながら，これら最新の事前学習モデルを用いた手法であっても，本質的にはコードの記述パターンに基づいた確率的な推論に留まっている．すなわち，Type-4 クローンの中でも，特に学習データに存在しない実装パターンを含むものや，アルゴリズムが根本的に異なるものに対しては，依然として検出が困難であるか，あるいは誤検知を生じる可能性がある．これに対し，本研究のアプローチは，実際の入出力挙動に基づく動的解析を採用しており，構文的特徴に依存せず，挙動に基づいた機能等価性の判定が可能である．本研究で構築したデータセットは，AI モデルの機能理解能力を検証するためのベンチマークとしても重要な役割を果たす．

7　おわりに

本研究では，C 言語で記述された大規模なソースコード群を対象に，入出力関係に基づく動的解析を用いて，構文的に異なるが機能的に等価な Type-4 コードクローンを抽出する手法を提案した．Type-4 クローンは静的な特徴が乏しく検出が困難であるが，提案手法では，関数の副作用による不確実性を排除するため，明示的な引数と戻り値を持つ関数に対象を絞り込み，テスト入力の自動生成と実行結果の比較を通じて機能等価性の判定を行った．

評価実験として，提案手法を約 1,000 万関数の大規模データセットに適用した．膨大な候補の中から，コンパイル可能性の確認，テストケースの生成，実行結果の照合といった多段階のフィルタリングを経ることで，誤検知を効果的に削減した．その結果，最終的に 68 件の機能等価関数ペアを特定した．これらは全て，異なるアルゴリズムやデータ構造を用いながらも同一の出力を返すペアであり，従来のコードクローン検出手法では発見が困難であった意味的な重複を捉えることに成功した．これらのペアは，開発者が意図せず実装した重複コードの実例として，クローンの発生メカニズムを知る上でも貴重な知見となる．

本研究で得られた機能等価関数ペアは，ソフトウェアの保守性向上に向けたリファクタリングの指針となるだけでなく，近年発展が著しい機械学習を用いたクローン検出技術の学習データや，検出精度の性能評価におけるベンチマークとして有用である．

今後の課題としては，大きく二点が挙げられる．第一に，適用範囲の拡大である．現在は副作用を持つ関数を除外しているが，静的解析と組み合わせることでグローバル変数の参照関係を特定し，対象となる関数を拡張する必要がある．第二に，判定精度の向上である．ランダムテストでは条件分岐の網羅率に限界があるため，ファジング技術を導入し，より厳密な等価性判定を実現することが求められる．

謝辞

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 教授には，毎週のミーティングをはじめ，本研究の遂行から論文の執筆に至るまで，終始懇切丁寧な御指導を賜りました．先生の熱心な御指導のおかげで，無事に本論文を完成させることができました．心より深く感謝申し上げます．

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 准教授には，研究活動に対して多くの貴重な御助言や御指導を賜りました．心より深く感謝申し上げます．

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 Kula Raula Gaikovina 教授には，研究の進捗における議論を通じて，多角的な視点から多くの貴重な御助言を賜りました．心より深く感謝申し上げます．

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 Olivier Nourry 助教には，本研究に対する有益なコメントや，研究活動全体を通じて温かい御指導を賜りました．心より深く感謝申し上げます．

最後に，様々な御指導・御助言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肥後研究室の皆様に，心より深く感謝いたします．

参考文献

[1] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*, pp. 209–224, 2008.

[2] James R. Cordy and Chanchal K. Roy. The NiCad clone detector. In *Proceedings of the 19th IEEE International Conference on Program Comprehension (ICPC '11)*, pp. 219–220, 2011.

[3] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[4] Google. GoogleTest: Google Testing and Mocking Framework. Accessed: 2026-02-01.

[5] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow, 2021.

[6] Yoshiki HIGO. Dataset of functionally equivalent java methods and its application to evaluating clone detection tools. *IEICE Transactions on Information and Systems*, Vol. E107.D, No. 6, pp. 751–760, 2024.

[7] Dmitry Ivanov, Alexey Babushkin, Saveliy Grigoryev, Pavel Iatchenii, Vladislav Kalugin, Egor Kichin, Egor Kulikov, Aleksandr Misonizhnik, Dmitry Mordvinov, Sergey Morozov, Olga Naumenko, Alexey Pleshakov, Pavel Ponomarev, Svetlana Shmidt, Alexey Utkin, Vadim Volodin, and Arseniy Volynets. Unittestbot: Automated unit test generation for c code in integrated development environments. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 380–384, 2023.

[8] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pp. 96–105, 2007.

[9] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.

[10] Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Raw C Code Corpus, January 2020. [Data set]. 42nd International Conference on Software Engineering (ICSE 2020).

[11] Jens Krinke and Chaiyong Ragkhitwetsagul. How the misuse of a dataset harmed semantic clone detection, 2025.

[12] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, Vol. 30, No. 2, pp. 126–139, 2004.

[13] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2016.

[14] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

[15] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, Vol. 74, No. 7, pp. 470–495, 2009.

[16] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 476–480, 2014.

[17] Zhipeng Xue, Zhijie Jiang, Chenlin Huang, Rulin Xu, Xiangbing Huang, and Liumin Hu. Seed: Semantic graph based deep detection for type-4 clone. In Gilles Perrouin, Naouel Moha, and Abdelhak-Djamel Seriai, editors, *Reuse and Software Quality*, pp. 120–137, Cham, 2022. Springer International Publishing.

[18] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering*, pp. 783–794. IEEE Press, 2019.

[19] Gang Zhao and Jeff Huang. Deepsim: deep learning code functional similarity. New York, NY, USA, 2018. Association for Computing Machinery.