ソースコードの構文木表現による構造類似性を用いた自動関数生成方式

北 椋太† 岡田 龍太郎† 峰松 彩子† 中西 崇文†

†武蔵野大学データサイエンス学部データサイエンス学科 〒135-8181 東京都江東区有明 3-3-3

E-mail: †s2022007@stu.musashino-u.ac.jp,
{ryotaro.okada, ayako.minematsu, takafumi.nakanishi}@ds.musashino-u.ac.jp

あらまし 本稿では，ソースコードの構文木表現による構造類似性を用いた自動関数生成方式について示す．一般的に，プログラミングにおいて，構造を簡略化することは，生産性やのちの保守性の向上において重要であり，これを自動化することは，プログラマにとって労力の軽減につながる．構造を簡略化する方法の一つとして，コードクローンを関数としてまとめることが挙げられる。本方式は，任意のソースコードを入力として，そのソースコードを構文木として表現した上で、各コード片間の類似度を計量することにより，類似した構造をもつコード片を抽出し，コードクローンを新たに生成した関数に置き換える．本方式を実現することにより，プログラマにとって，より簡略化した保守性の高いソースコードを書くことが可能になる．

キーワード ソースコード, コードクローン，構文木, 構造類似性, 関数生成

1. はじめに

近年，長期的な運用を目的とするソフトウェア大規模化や，ソフトウェアの利用分野の拡大にともない，プログラムの不具合やそれにともなうシステムの故障が社会的に問題となることが多くなっている．ソフトウェア開発ライフサイクル全体において，保守のコストが占める割合は非常に高く，プロジェクトの大規模化にともなって,その割合はさらに増加する.そこで，ソフトウェアの保守作業の効率化が重要な課題となっている．ソフトウェアの開発や，保守作業を行うにあたり，開発者はソースコードから，プログラムの構造を理解し，修正する必要があり，この作業がソフトウェアのメンテナンスの多くの割合を占めている．そのため,ソフトウェアの保守コストを削減するためには，ソースコードの理解や，修正を容易にすることが重要である．これらの作業の効率を下げている一因として，コードクローンがある．コードクローンとは，ソースコード中の一部分(コード片)のうち，類似または一致したコード片がソースコード中に複数存在するもののことである．あるコード片にて不具合が発見された場合には，そこだけでなく，そのコードクローンすべてについて検査し,修正を検討する必要がある.しかし，大規模なソフトウェア開発において．それらすべてのコードクローンに対し，手作業で発見し，修正を行うには，非常に大きなコストを要する．また，一般に開発者は，コードクローン検出に grep[]を用いると考えられる．grep を用いたコードクローン検出には，適切なキーワードを選定し，検索を行わなければ，期待する結果を得られないことがある．そのため，コードクローンをキーワードではなく，類似構造から検出し，自動で関数化することは，プログラマにとって労力の軽減につながる．

本稿では，ソースコードを構文木で表現することで，構文木内の構造類似性に基づいて，類似した構造をもつコード片を検出する．これによって，コードクローンを抽出し，自動で関数化を行うことによって，ソースコードの構造の簡略化を実現する．

本方式を実現することにより，コードクローンを再利用可能な関数に変換し，保守性が高いソースコードを書くことが可能になる．

本稿の構成は，次の通りである．2 章では，関連研究について紹介する．3 章では，本提案方式であるソースコードの構文木表現による構造類似性を用いた自動関数化方式について示す．4 章では，本方式を実現するシステムを構築し，実験を行う．5 節で本稿をまとめる．

2. 関連研究

本節では，本方式に関連する研究について挙げる．本節の構成は，次の通りである．2.1 節では，コードクローン検出に関する研究について挙げる.2.2 節では，関数生成に関する研究について挙げる．

2.1. コードクローン検出に関する研究

コードクローン検出の研究では，これまでに，さまざまなモデルでソースコードを表現し，研究および比較されてきた[1]．ここでは，主にテキストやトークンなどの字句を解析して行う手法と，構文木やグラフな

どの構文を解析して行う手法の 2 つに分類し，関連する研究について挙げる．

2.1.1. 字句を用いた検出手法に関する研究

神谷ら[2]は，プロググラムテキストをトークンの並びで表現し，そのトークン列の等価性に基づいてコードクローンを検出する CCFinder を開発した．

吉田ら[3]は，コード片に含まれる識別子の類似性に基づいてコードクローンを検出する手法を提案している．この研究では，ソースコードの関数が含む語を抽出し，共起関係に基づいて類義語を特定し，入力コード片が含む語と一致する，もしくは類似する語を含む関数を提示している．

2.1.2. 構文を用いた検出手法に関する研究

Gabel ら[4]は，ソースコードを program dependence graph(PDG)から類似する部分グラフを取り出し，構造化することで，より単純な類似問題にし，コードクローンを検出する手法を提案している．

Jiang ら[5]は，ソースコードを抽象構文木で表現し，木構造の類似度を算出することでコードクローンを検出する DECKARD を開発した．

2.2. 簡略化に関する研究

石原ら[6]は，ファイル単位にまたがって存在するコードクローンのライブラリ化を目的に，大規模なソフトウェア群を対象に，メソッド単位のコードクローンを検出する手法を提案している．

Baxter ら[7]は，抽象構文木を用いたコードクローンの検出手法を提案する中で，任意の類似度を設定し，該当する類似コード片をマクロとしてまとめることで，コードクローンに該当する箇所だけを比較することを可能とした．

2.3. 本研究の位置付け

本研究では，入力されたソースコードを構文解析することで，抽象構文木で表現する．抽象構文木内の各ノードには，親子関係だけでなく，ソースコード内での行番号や,葉までの最大距離などの情報を保持する．その後，任意の深さ以上の部分木を全て抽出したのちに，一致または類似する構造をもつ部分木のみを検出する．検出した部分木に該当するコード片をコードクローンとし，関数の生成または推薦を行う．

これまでのコードクローン検出およびに関する研究では，類似するコード片を検出し，該当位置または抽出したコードクローンを出力とすることが主であった．そのため，あるコード片にて不具合が発見された場合には，出力されたコードクローン全てを確認し，修正を行う必要がある．一方，本研究では，検出したコードクローンを新たに生成した関数に置き換えることで，一元的に管理をすることが可能となる．また，本研究では，抽出する部分木の深さをパラメータとすることで，コードクローンを検出するスコープを変更することができる．

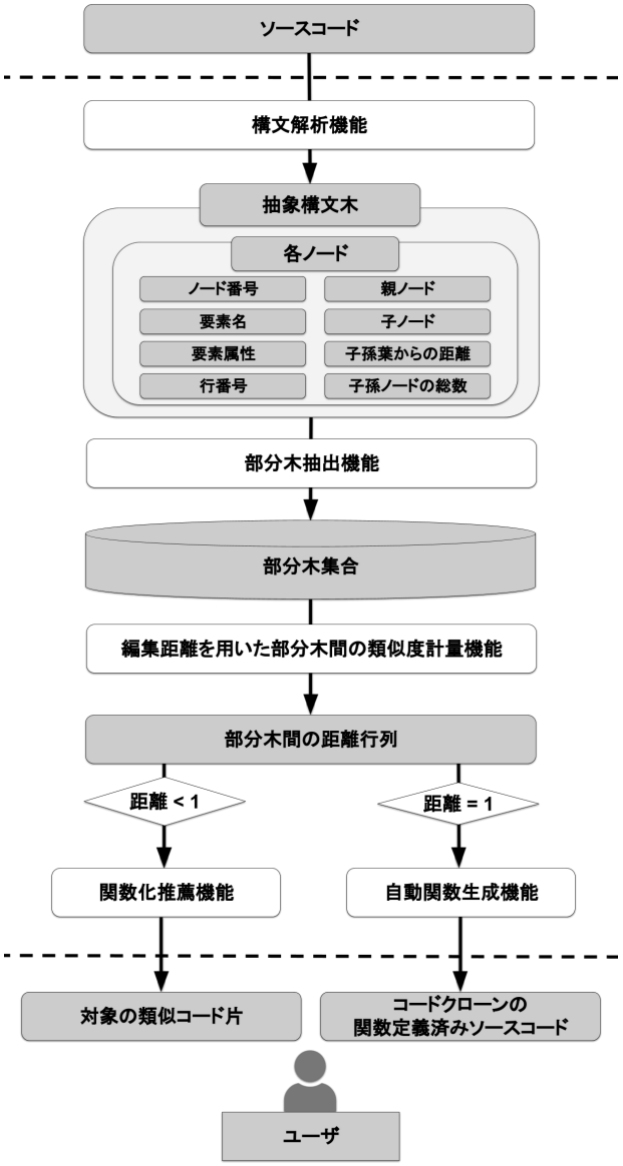

図 1 提案システムの全体像

3. ソースコードの構文木表現による構造類似性を用いた自動関数生成方式

本章では，提案方式である，ソースコードの構文木表現による構造類似性を用いた自動関数生成方式について示す．

3.1. 提案手法の概要

本節では，提案手法の全体像について述べる．提案システムの全体像を図 1 に示す．本稿では，ソースコードを対象に，関数化による構造の簡略化を実現させる手法として，構文木の構造類似性に基づいてコード

クローン検出することを目的としている.本研究では，ソースコードを表現する抽象構文木から，任意の深さ以上の部分木を全て抽出したのちに，その部分木間の類似度を計量することでコードクローン検出を実現する．その部分木間の類似度計量手法として，構文木の編集距離による類似度計量手法を用いる．

本手法は，構文解析機能，部分木抽出機能，編集距離を用いた部分木間の類似度計量機能，関数化推薦機能，自動関数生成機能からなる．

本方式では，入力データとして任意のソースコードを与え，構文解析を行い，抽象構文木を生成する．次に，生成した抽象構文木から指定した最大深さ k 以上の深さを持つ部分木を全て抽出し，部分木集合を作成する．そして，部分木間の構造類似性に着目し，類似度が大きい部分木同士を関数として定義できる可能性が高いと仮定し，関数の生成または関数化の推薦を行う．

3.2. 構文解析機能

本節では，ソースコードから抽象構文木を生成する構文解析について述べる．構文解析は，ソースコードから要素の最小単位であるトークンを取り出す字句解析を行なったのち，そのトークン間の関係を明確にする手続きである．この際，その結果を木構造で表現したものが構文木であり，特にトークンの詳細な情報を取り除き，動作の意味に関係ある情報のみを取り出した構文木を抽象構文木と呼ぶ．本システムでは，変数名などの動作に関与しない要素を省略するために，抽象構文木を用いることで構造のみから類似度を計算する．本方式では，出力時にソースコードへ再変換するために，各ノードはソースコードの行番号をノードに保持する．

3.3. 部分木抽出機能

本節では，抽象構文木から対象となるすべての部分木を抽出する方法について述べる．関数生成を行うにあたって，あまり小さなコード片を関数に置き換えても可読性や保守性の向上には繋がらないため，関数化の対象となるコード片は，ある程度大きさの処理のまとまりである必要がある．そこで，本方式では，最大深さ k を設定し，指定した k 以上の深さを持つ部分木のみを抽出し，部分木集合に格納する．これを実現するため，各ノードは，子ノードを再帰的に探索することで，葉との最大距離を算出し，保持する．

3.4. 編集距離を用いた部分木間の類似度計量機能

本節では，作成した部分木の集合から各部分木間の類似度を計量する方法について述べる．構文木同士の類似度を計量する手法として，それぞれの構文木の部分木を全て抽出し，完全一致する個数を基にする方法と，部分木の追加や削除の操作をおこない，その編集距離を基にする方法などがあげられる．本研究では，同一機能の関数生成を出力とするにあたって，操作回数に着目して類似度を計量するために，編集距離を用いた類似度計量を行う．3.3 で抽出した部分木全てに類似度計量を行うことで，部分木間の距離行列を作成する．

3.5. 関数化推薦機能

本節では，類似度 1 未満における出力である関数化を推薦する方法について述べる．3.4 で用意した距離行列を用いて，ユーザの設定した任意の類似度を超える部分木のみを抽出する．部分木が保有する行番号から元のコード片を参照し，コード片間における差分を抽出する．対象となる類似コード片およびそれらの差分をユーザに出力する．

3.6. 自動関数生成機能

本節では，類似度 1 における出力である関数生成を実現する方法について述べる．はじめに，関数を定義するにあたって，引数および処理を空欄としたテンプレートを作成し，ソースコードの 1 行目に挿入する．3.4 で用意した距離行列のうち， 距離が 1 となる部分木のみを抽出する．部分木が保有する行番号から元のコード片を参照する．同一構造をもつコード片に含まれる変数の個数をカウントし，作成したテンプレートに同数の引数を設定する．コード片をテンプレートの処理部に格納し，変数を全て設定した引数に置き換える．元のコード片を削除し，コード片に含まれる全ての変数を入力および出力として，作成した関数を呼び出す．

```
(前略)

for _ in range(cannon1):
    if rate1 > random.random():
        cannon2_ -= 1
    if cannon2_ == 0:
        break
for _ in range(cannon2):
    if rate2 > random.random():
        cannon1_ -= 1
    if cannon1_ == 0:
        break

cannon1, cannon2 = cannon1_, cannon2_
print(cannon1, cannon2)
```

図 2 コードクローンの例

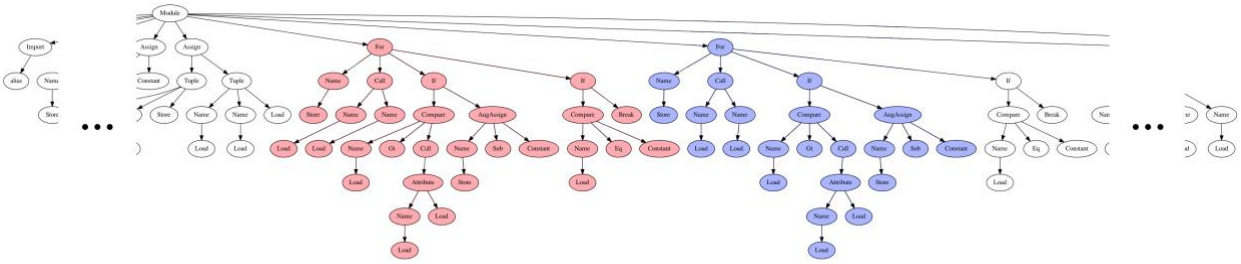

図 3 構文解析機能によって生成した抽象構文木

(関数部)

```
def define1(var1, var2):
    if var1 > random.random():
        var2 -= 1
    return var1, var2
```

(実行部)

(前略)

```
for _ in range(cannon1):
    rate1, cannon2_ = define1(rate1, cannon2_)
    if cannon2_ == 0:
        break
for _ in range(cannon2):
    rate2, cannon1_ = define1(rate2, cannon1_)
    if cannon1_ == 0:
        break

cannon1, cannon2 = cannon1_, cannon2_
print(cannon1, cannon2)
```

(a) k = 5 の場合

(関数部)

```
def define1(var1, var2, var3):
    for _ in range(var1):
        if var2 > random.random():
            var3 -= 1
        if var3 == 0:
            break
    return var1, var2, var3
```

(実行部)

(前略)

```
cannon1, rate1, cannon2_ =
        define1(cannon1, rate1, cannon2_)
cannon2, rate2, cannon1_ =
        define1(cannon2, rate2, cannon1_)

cannon1, cannon2 = cannon1_, cannon2_
print(cannon1, cannon2)
```

(b) k = 6 の場合

図 4 関数生成機能の実行例

4. 実験

本章では，同一ソースコード内に存在するコードクローン検出による自動関数生成方式の評価実験について述べる．4 節の構成について述べる．4.1 節では実験環境について述べる．4.2 節では，実験 1 として，用意したソースコードに対して構文解析機能を適用し，抽象構文木を抽出し，抽出された構文木を観察することで，関数化における置き換えのための最適な深さについて検討する．4.3 節では，実験 2 として，提案方式である自動関数化方式をソースコードに対して適用し，提案方式の有効性を検証する．またその際，関数化において置換する部分木の深さパラメータとして変化させて実験を行い，最適な深さについて検討を行う．

4.1. 実験環境

提案方式を Python 言語を用いて構築した．

実験対象であるコードクローンを含むソースコードは図 2 に示す．抽象構文木の抽出には Python の標準ライブラリである ast を用いた．部分木間の類似度計量，編集距離を用いて 2 つの木構造間の類似度を計量する Python 用ライブラリとして公開されている zss を用いた．

4.2. 実験 1

4.2.1. 実験目的

実験 1 として，用意したソースコードに対して構文解析機能を適用し，抽象構文木を抽出し，抽出された構文木を観察することで，関数化における置き換えのための最適な深さについて検討する．

4.2.2. 実験結果

構文解析機能によって抽出された抽象構文木を図 3 に示す．ノード数 112，根ノードから子孫ノードの最大深さ 7 の抽象構文木が生成された．本ソースコードの主な処理を表す部分木における子孫ノードの最大深さ k について，ライブラリのインポートは k=1，変数の定義は k=2，packing を使用した変数のスワップは k=3，for による繰り返し処理は k=6，if による条件分岐は k=5，print 関数による出力は k=3 となった．

4.2.3. 考察

本研究の目的は，コードクローンを検出し，関数化することによって，一元的に管理することで保守作業の効率を上げることである．実験 1 で得られた k が 3 以下の処理については，実際のソースコードを確認すると代入文などの 1 行の簡単な処理である．これらは，保守作業の効率化に有用でなく，関数化する価値が小さい．そのため，実験 1 より，本ソースコードにおける最大深さ k は 5 以上が適切であると考えられる．

4.3. 実験 2

4.3.1. 実験目的

本研究では，コードクローン検出の対象となる部分木において，子孫ノードの最大深さ k を任意の値で設定する．このとき，k の値を比較してどの程度違いがあるかを調査する実験を行った．

本実験では，実験 1 の結果を踏まえて最大深さ k を 5 としたときと，6 としたときの場合についてシステムを実行した．

4.3.2. 実験結果

図 2 に示すソースコードに対して，自動関数生成機能を実行した結果を，k を 5 としたとき場合について図 4(a)に，6 としたときの場合について図 4 (b)に示す．k を 5 とした場合には， if 文による条件分岐文のみが関数化された．このとき，関数は 2 つの引数を指定する必要とする．6 とした場合には，先ほどの条件分岐文を含む for 文による繰り返し文が関数化された．このとき，関数は 3 つの引数を必要とする．

4.3.3. 考察

本実験では，k が 5 としたときと，k が 6 としたときの場合について，どちらも正常に置き換えることに成功した．元のソースコードは 18 行であったのに対して，k を 5 とした場合については，実行部が 16 行となり，視覚的な変化は十分に得られなかったと考えられる．また，簡略化における保守性の向上についても，有用な結果は得られなかった．k を 6 とした場合については，実行部が 10 行まで減少した．処理についても，まとまりの全体を関数に置き換えることに成功している．

5. まとめ

本稿では，ソースコードの構文木表現による構造類似性を用いた自動関数生成方式について述べた．本方式では，ソースコードを構文木で表現することで，構文木内の構造類似性に基づいて，類似した構造をもつコード片を検出する．これによって，コードクローンを抽出し，新たに生成した関数に置き換えることによって，ソースコードの構造の簡略化を実現した．

また，本方式を検証するための実験システムを構築し，本方式の有効性を検証する実験を行った．

ソースコードの構文木表現による構造類似性を用いた自動関数生成方式を実現したことによって，同一処理をもつコード片を一元管理することが可能になり，より効率的な保守作業の一助となりうる．

今後の課題としては，完全一致だけでなく類似構造における自動関数生成，適切な戻り値の設定，元のソースコードを参照することによる適切な変数の命名の実現，子孫ノードの最大深さの自動調整機能の実装があげられる．

参 考 文 献

[1] 神谷年洋，肥後芳樹，吉田則裕，“コードクローン検出技術展開”，コンピュータソフトウェア，28(3), pp.28‐42, 2011.

[2] T. Kamiya, S. Kusumoto, K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code", IEEE Transactions on Software Engineering, 28(7), pp. 654-670, 2002.

[3] 吉田則裕，服部剛之，早瀬康裕，井上克郎，"類義語の特定に基づく類似コード片検索法"，情報処理学会論文誌，50(5), pp.1506-1519, 2009.

[4] M. Gabel, L. Jiang, Z. Su, "Scalable detection of semantic clones", In Proceedings of the 30th International Conference on Software Engineering, pp. 321-330, 2008.

[5] L. Jiang, G. Misherghi, Z. Su, S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones", In 29th International Conference on Software Engineering, pp. 96-105, 2007.

[6] 石原知也，堀田圭佑，肥後芳樹，井垣宏，楠本真二，"大規模ソフトウェア群に対するメソッド単位のコードクローン検出"，電子情報通信学会技術研究報告，111(481), pp. 31-36, 2012.

[7] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, "Clone detection using abstract syntax trees", In Proceedings of International Conference on Software Maintenance, pp. 368-377, 1998.