コードクローン集約によるファジングの実行効率調査

徳井 翔梧　吉田 則裕　崔 恩瀞　井上 克郎　肥後 芳樹

ファジングは高速なテストケース生成と実行によって脆弱性を検出する手法である．AFL は基本ブロックレベルの実行パスを観測し，未発見のパスを通過するテストケースを効率的に探索するファジングツールである．本研究では，対象のソースコード内のコードクローン集約による，AFL のパス探索効率の変化を調査した．基本ブロックを含むコードクローンを集約することで，パスが単純化され，AFL が観測するパスの数が減少し，未発見のパスに到達しやすくなると考えた．実験の結果として，AFL が発見したパス数に統計的な有意差は見られなかったが，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．

Fuzzing is a technique for detecting vulnerabilities through rapid test case generation and execution. AFL, a well-known fuzzing tool, efficiently explores test cases that pass through previously undiscovered paths by observing execution paths at the basic block level. In this study, we investigate the change in AFL path search efficiency by aggregating code clones of the target source code. We hypothesize that aggregating code clones that contain basic blocks would aggregate paths that AFL can detect, reducing the number of paths observed in AFL and making it easier to reach undiscovered paths. The experimental results showed no statistically significant difference in the number of paths discovered by AFL, but code clone aggregation did change the test cases generated by AFL, detecting one undiscovered crash.

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従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 [1]．

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 [7]．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 [2][†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 [2][7]．

ソースコード中には，類似または一致した部分が存在することがあり，それらはコードクローンと呼ばれている [3]．CCFinder は，コードクローン検出技術

Investigation of Code Clone Aggregation on Fuzzing Execution Efficiency.
Shogo Tokui, Yoshiki Higo, 大阪大学, Osaka University.
Norihiro Yoshida, 立命館大学, Ritsumeikan University.
Eunjong Choi, 京都工芸繊維大学, Kyoto Institute of Technology.
Katsuro Inoue, 南山大学, Nanzan University.
コンピュータソフトウェア, Vol.42, No.2 (2025), pp.135–141.
[研究論文 (レター)] 2024 年 9 月 5 日受付.

†1　https://lcamtuf.coredump.cx/afl
†2　https://lcamtuf.coredump.cx/afl/#bugs

の代表的なツールであり，変数名を除いて字句単位で一致するコードクローンを検出する [8]．CCFinder が検出したコードクローンは 1 つの関数に集約できる場合がある．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 が観測する一部のパスが集約され，未発見のパスに到達しやすくなり，ファジングを効率化できると考えた．

本研究では，GNU Binutils を対象に CCFinder 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．すなわち，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．

2 背景

2.1 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 [2][9]．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．

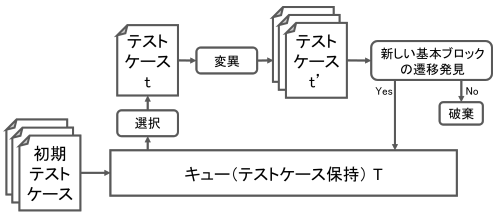


図 1　テストケースに着目した AFL の動作フロー　[2]

AFL の動作フローについて説明する．図 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ために効率的に探索する手法が多く研究されている．

†3 https://github.com/google/AFL/blob/master/docs/technical_details.txt

2.2 CCFinder

CCFinder は，プログラムテキスト中の同一または類似したコードの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 [6]．CCFinder は，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コード片や，変数名や関数名，変数の型などの識別子のみが異なるコード片を検出する．互いに一致または類似しているコード片の対をクローンペアと呼び，コードクローンの集合をクローンセットと呼ぶ．

コードクローンの存在はバグを潜在化させるため，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や監視を行う [3]–[5]．コードクローンに対するリファクタリングとは，コードクローンをソースコード中から除去することを意味する．例えば，コードクローンを 1 つの関数に集約することで除去が可能である．しかし，引数や返り値の型が異なる場合や，コードクローン同士の位置に重なりがある場合，関数の抽出は難しい．このように，関数抽出が困難なコードクローンに対しては，クラスの継承を使って関連性を高めることで似た処理であることを明示するなど，コードクローンへの変更や進化を追跡監視する手法が研究されている．

2.1 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．コードクローンに欠陥が含まれている場合，同一のコードクローンにも同様の欠陥が含まれる可能性が高く，同一のコードクローンに対するファジングは 1 度で十分だと考えられる．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．

3 実験

本研究では，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．

3.1 研究課題

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 [9]．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 章で述べる．

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 [9] 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 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．

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. CCFinder 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

CCFinder を用いて実験対象プログラムのコードク

表 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.

ローンを検出した. CCFinder 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, クローンセットごと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.

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 2 に示し, AFL で検出されたパス数の推移を図 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 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 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

表 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

program	nm		objdump		readelf	
refactoring	original	refactored	original	refactored	original	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**

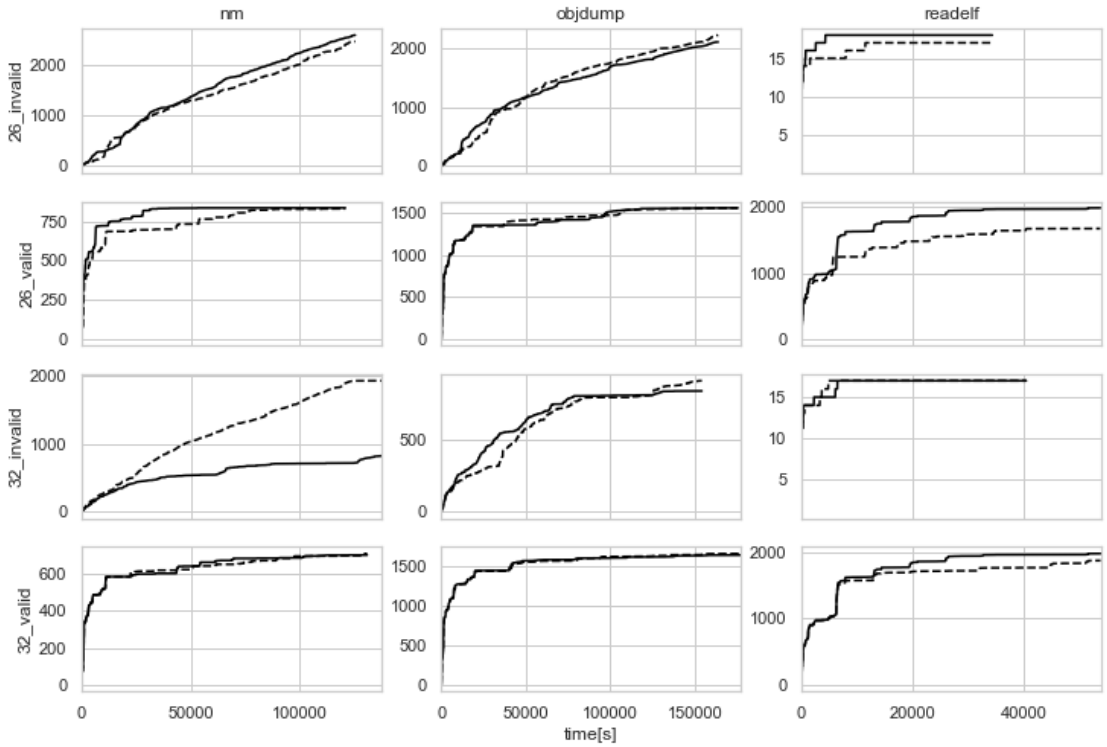


図 2 AFL 実行により検出されたパス数 (破線:コードクローン集約前, 実線:コードクローン集約後)

表 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%

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実行すると，すべての一致したテストケースは集約したコードクローンを実行しないことを確認した．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

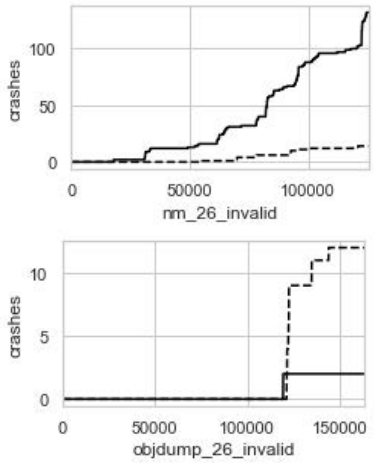


図 3　AFL 実行により検出されたクラッシュ数 (破線:orginal, 実線:refactored)

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　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るクラッシュ数の増加は期待でき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 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．

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，コードクローンが多く内在する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．

5　まとめと今後の課題

本研究では，コードクローン集約による AFL の実行効率に与える影響について調査した．結果として，AFL が検出したパス数に有意差は確認されず，コードクローン集約による AFL の実行効率の向上は期待できないことが示された．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．

謝辞　本研究は，JST さきがけ JPMJPR21PA ならびに JSPS 科研費 JP24K02923, JP20K11745, JP23K11046, JP24H00692, JP21K18302, JP21H04877, JP23K24823, JP22K11985 の支援を受けた．

参 考 文 献

[1] Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., and McMinn, P.: An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, Vol. 86, (2013), pp. 1978–2001.

[2] Böhme, M., Pham, V. T., and Roychoudhury, A.: Coverage-based greybox fuzzing as markov chain. in *Proc. of CCS*, 2016, pp. 1032–1043.

[3] 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, Vol. 91, No. 6, (2008), pp. 1465–1481.

[4] 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, Vol. 28, No. 4, pp. 4_43–4_56, 2011.

[5] 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, Vol. 60, No. 4, (2019), pp. 1051–1062.

[6] Kamiya, T., Kusumoto, S., and Inoue, K.: CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *TSE*, Vol. 28, No. 7, (2002), pp. 654–670.

[7] Manes, V. J. M., Han, H., Han, C-W., Cha, S. K., Egele, M., Schwartz, E. J., and Woo, M.: The art, science, and engineering of fuzzing: A survey. *TSE*, Vol. 47, 2021, pp. 2312–2331.

[8] Rattan, D., Bhatia, R., and Singh, M.: Software clone detection: A systematic review. *Inf. Softw. Technol.*, Vol. 55, No. 7, (2013), pp. 1165–1199.

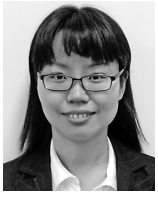
[9] 都築夏樹, 吉田則裕, 戸田航史, 山本椋太, 高田広章: カバレッジに基づくファジングツールの比較評価. コンピュータソフトウェア, Vol. 39, No. 2, (2022), pp. 2_101–2_123.

徳 井 翔 梧

2019 年大阪大学大学院情報科学研究科博士前期課程修了．現在，同大学大学院情報科学研究科博士後期課程．および富士通株式会社研究員．コードクローン管理や生成 AI を利用した SE 技術に関する研究に従事．情報処理学会，日本ソフトウェア科学会．

吉 田 則 裕

2009 年大阪大学大学院情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員 (PD)．2010 年奈良先端科学技術大学院大学情報科学研究科助教．2014 年名古屋大学大学院情報科学研究科附属組込みシステム研究センター准教授．2022 年より立命館大学情報理工学部教授．博士 (情報科学)．ソフトウェア保守やソフトウェアテストに関する研究に従事．



崔 恩瀞

2015 年大阪大学大学院情報科学研究科博士後期課程修了．同年同大学大学院国際公共政策研究科助教．2016 年奈良先端科学技術大学院大学情報科学研究科助教．2018 年より同大学先端科学技術研究科助教 (改組による)．2019 年より京都工芸繊維大学情報工学・人間科学系助教を経て 2024 年より同大学の准教授．博士 (情報科学)．コードクローン管理やリファクタリング支援手法に関する研究に従事．

井 上 克 郎

1979 年大阪大学基礎工学部情報工学科卒業．1984 年博士課程了．同年同大学基礎工学部助手．2002 年情報科学研究科教授．2022 年南山大学理工学部教授．工学博士．ソフトウェア工学, 特にソフトウェア開発手法, プログラム解析, 再利用技術の研究に従事．

肥 後 芳 樹

2002 年大阪大学基礎工学部情報科学科中退．2006 年同大学大学院博士後期課程了．2007 年同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助教．2015 年同准教授．2022 年同教授．博士 (情報科学)．ソースコード分析，特にコードクローン分析，リファクタリング支援，ソフトウェアリポジトリマイニング及び自動プログラム修正に関する研究に従事．情報処理学会，日本ソフトウェア科学会，IEEE 各会員．