修士学位論文

題目

R を対象とした再現可能な欠陥データセットの構築

指導教員

楠本 真二 教授

報告者

石野 太一

令和 6 年 2 月 1 日

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

令和 5 年度 修士学位論文

R を対象とした再現可能な欠陥データセットの構築

石野 太一

内容梗概

ソフトウェア開発中に発生した様々な欠陥を再現可能な形で収集した欠陥データセットが公開されている．欠陥データセットは欠陥研究を支える 1 つの重要な要素となっており，欠陥箇所の自動特定や欠陥の自動修正などの研究で利用されている．特に Java や C 言語などのプログラミング言語では複数の欠陥データセットが公開されており，その言語における欠陥研究を大きく促進している．一方で，R 言語には再現可能な欠陥データセットが存在せず，R 言語における欠陥研究が全く行われていない．そこで，本研究では GitHub 上の R プロジェクトから開発過程で発生した欠陥を収集し，欠陥データセットを構築する．本研究では 3 プロジェクトの計 7,633 コミットを対象として欠陥収集を実行した．その結果，過去 5 年間の開発履歴から 172 個の欠陥を収集した．収集した欠陥に対してメトリクスに関する分析を行い，欠陥の規模が多様であることを確認した．また，欠陥を再現するためのコマンドラインインタフェースを実装した．収集した欠陥と実装したコマンドラインインタフェースは欠陥データセットとして GitHub 上で公開している（https://github.com/kusumotolab/Rbugs）．

主な用語

R 言語，欠陥データセット

目次

図目次

表目次

1　はじめに

ソフトウェア開発におけるデバッグコストの削減を目的として，プログラム中に存在する欠陥を対象とした研究が盛んに行われている．具体的には，欠陥の傾向の分析 [1] や，欠陥箇所の自動特定 [2] [3] [4]，欠陥の自動修正 [5] [6] などが挙げられる．これらの欠陥研究を支える土台として，欠陥データセット [7] [8] [9] [10] [11] が存在する．欠陥データセットとは，ソフトウェア開発中に発生した様々な欠陥を再現可能な形で収集したデータセットである．欠陥データセットには欠陥を含むソースコード，欠陥を検出できるテストケース，テスト実行環境の定義情報が含まれており，欠陥を再現可能である．広く用いられる欠陥データセットとして Defects4J [7] が挙げられる．Defects4J は Java の有名な 17 プロジェクトから 835 個の欠陥を収集した研究であり，欠陥箇所の自動特定 [2] [3] [4] や欠陥箇所の自動修正 [5] [6] などの研究で幅広く使用されている．2024 年 1 月時点で Defects4J の被引用数は 1,200 を超えており，Java における欠陥研究を大きく促進している．Java 以外のプログラミング言語においても欠陥データセットは公開されており，C 言語では ManyBugs [8]，Python では BugsInPy [9]，JavaScript では BugsJS [10] などが挙げられる．

上記以外のプログラミング言語として，主に統計解析で使用される R 言語が存在する．R 言語は統計学，生態学，地理情報学，経済学など様々な分野で使用されている．特に生態学の分野においては R 言語の使用頻度が高く，R 言語は生態学分野における分析の重要な要素となっている [12]．2024 年 1 月現在，プログラミング言語の人気指標の 1 つである PYPL 指数 [13] において R 言語は 6 位となっている．R 言語は人気の高い言語の 1 つであり，現在でも頻繁に使用されている．

R 言語は一定の需要を持つ言語であるにもかかわらず，ソフトウェア工学の観点で十分に研究されているとはいえない [14]．R 言語における研究課題の 1 つとして，再現可能な欠陥データセットの構築が挙げられる．Java や C 言語などのプログラミング言語では多数の欠陥データセットが提案されている一方，R 言語においては再現可能な欠陥データセットが存在しない．そのため，R 言語を対象とした欠陥研究が全く行われていない．他の言語と同様に，R 言語の開発プロジェクトでは issue 管理や単体テストが行われており，過去に発生した欠陥を収集可能である．しかし，我々の知る限りでは R 言語の再現可能な欠陥データセットは存在しない．

本研究の目的は，R 言語における欠陥研究で利用可能な欠陥データセットの構築である．そのため，GitHub 上の R プロジェクトから開発過程で発生した欠陥を収集する．欠陥収集過程においてテストを実行し，テストケースによって欠陥の再現性を保証する．本研究では 3 プロジェクトの計 7,633 コミットを対象として欠陥収集を実行した．その結果，過去 5 年間の開発履歴から 172 個の欠陥を収集した．収集した欠陥に対してメトリクスに関する分析を行い，欠陥の規模が多様であることを確認した．また，欠陥を再現するためのコマンドラインインタフェースを実装した．収取した欠陥と実装したコマン

ドラインインタフェースは欠陥データセットとして公開している．本データセットを利用することで，欠陥研究を行う研究者は欠陥を容易に再現できる．

以降，2 節では本研究で対象とする R 言語と，構築する欠陥データセットについて説明する．3 節では提案手法について述べ，4 節では行った実験を示し，結果を分析する．5 節では実験結果について考察し，6 節では本研究における妥当性への脅威について述べる．7 節では関連研究を示し，最後に 8 節で本研究のまとめと今後の課題について述べる．

2 準備

2.1 R 言語

R 言語はパッケージベースの言語であり，R リポジトリで多数のパッケージが公開されている．最も主要な R リポジトリは CRAN[*1]である．CRAN は R 言語の公式リポジトリであり，2024 年 1 月時点で 20,000 以上のパッケージが公開されている．CRAN の他には Bioconductor[*2]，R-Forge[*3], GitHub などでパッケージが公開されている．

ユーザは R リポジトリからパッケージをインストールし，第三者が開発した機能を利用できる．パッケージのインストールには依存関係の解決が必要となるが，R 標準の機能や外部パッケージにより自動で行われる．依存関係の解決において，標準では最新バージョンの依存パッケージがインストールされる．また，古いバージョンのパッケージ本体をインストールする場合においても最新バージョンの依存パッケージが自動でインストールされる．しかし，R 言語は後方互換性のない変更の頻度が高く，依存パッケージの後方互換性のない変更により R パッケージが頻繁に壊れてしまう [15]．そのため，古いバージョンのパッケージ本体をインストールするためには適切なバージョンの依存パッケージを手動でインストールする必要がある．

2.2 欠陥データセット

欠陥データセットとは，ソフトウェア開発中に発生した様々な欠陥を再現可能な形で収集したデータセットである．欠陥を再現するためには欠陥を含むソースコード，欠陥を検出できるテストケース，テスト実行環境の定義情報が必要であり，欠陥データセットにはこれらの情報が含まれている．さらに，欠陥に関するメタ情報として実際のコミットへのリンク，欠陥に関連する issue，テスト結果，コードメトリクスなどが含まれる．

[*1] https://cran.r-project.org/
[*2] https://www.bioconductor.org/
[*3] https://r-forge.r-project.org/

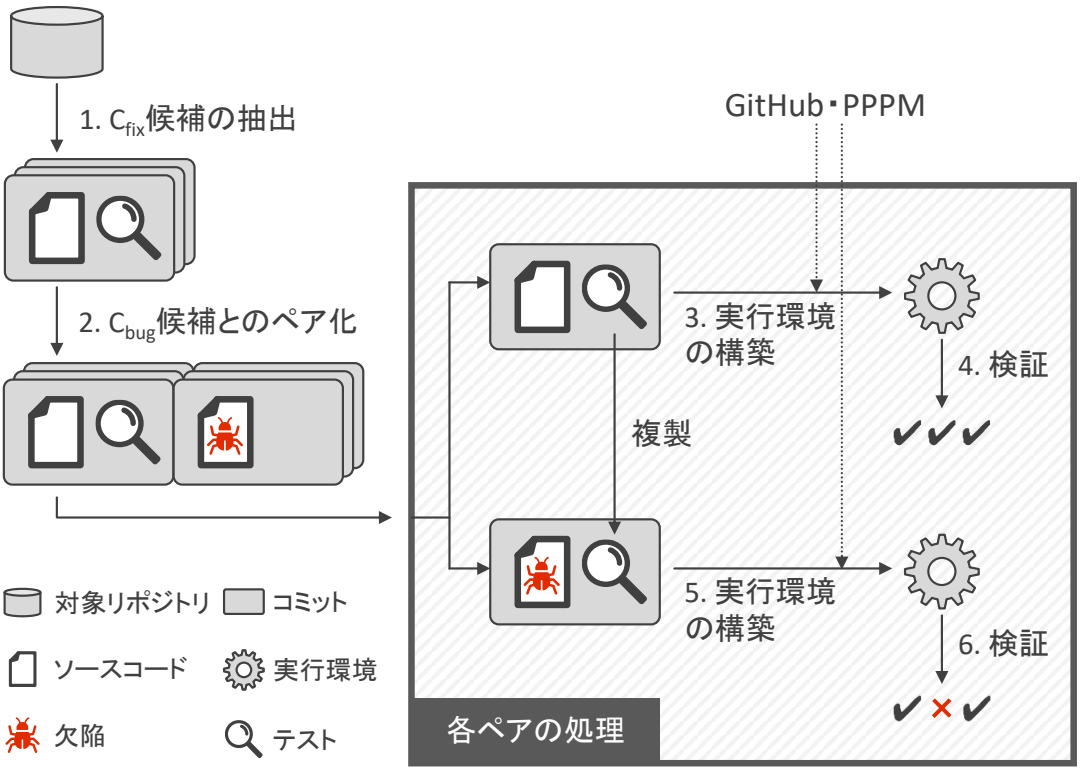

図 1　欠陥収集の流れ

3　提案手法

3.1　概要

本研究では，欠陥限局や自動プログラム修正などの欠陥研究で利用可能な欠陥データセットの構築を目的として，GitHub 上の R プロジェクトから開発過程で発生した欠陥を収集する．収集対象とする欠陥はドキュメントの不備などではなく，R のソースコードに含まれる欠陥である．本研究で収集する欠陥データには欠陥を含むソースコード，欠陥が修正されたソースコード，テストが含まれる．欠陥を含むソースコードは 1 つ以上のテストケースに失敗する．このテストケースによって欠陥の再現性が保証される．また，欠陥を含むソースコードと欠陥が修正されたソースコードの差分が可能な限り小さくなるように欠陥を収集する．これは，欠陥研究で欠陥を使用する際，差分にリファクタリングなどの欠陥修正以外の変更が含まれないことが望ましいためである．収集した欠陥に対しては手動検証を行い，本当に欠陥であることを確認する．最後に，欠陥を再現するためのコマンドラインインタフェースを実装し，利用者が欠陥を容易に再現できるようにする．

3.2 欠陥の収集

本研究では単体テストを用いて欠陥を自動的に収集する．基本的な流れとしては Benton ら [11] の手法に従う．まず，GitHub 上の R プロジェクトから欠陥含有コミット（C_{bug}）候補と欠陥修正コミット（C_{fix}）候補のペアを抽出する．C_{bug} はソースコードに欠陥が含まれているコミットであり，C_{fix} は C_{bug} のソースコードに含まれている欠陥が修正されているコミットである．次に，単体テストを用いて欠陥の有無を自動的に検証する．図 1 に欠陥収集の流れを示す．欠陥収集は次の 6 つのステップから構成される．

ステップ 1: C_{fix} 候補の抽出

本ステップでは GitHub 上の大量のコミットから C_{fix} 候補を抽出する．以下の 2 つの条件をともに満たすコミットを C_{fix} 候補とする．1 つ目の条件は，「コミットメッセージに issue 番号もしくはキーワードのいずれかが含まれる」である．大量のコミットから欠陥修正に関連するコミットのみに絞り込むためにこのフィルターを採用した．issue 番号は#数字で表される．キーワードは error，issue，fix，repair，solve，remove，problem の 7 語である．これらのキーワードは欠陥修正に関連するキーワードとして Benton ら [11] の手法で採用されたキーワードである．2 つ目の条件は，「R のソースコードに変更がある」である．本研究では R のソースコードに含まれる欠陥を収集対象としているため，このフィルターを採用した．

ステップ 2: C_{bug} 候補と C_{fix} 候補のペア作成

それぞれの C_{fix} 候補に対して，そのコミットで修正した欠陥を元々含んでいた C_{bug} 候補を特定する必要がある．本ステップではステップ 1 で抽出した C_{fix} 候補から対応する C_{bug} 候補を特定し，C_{bug} 候補と C_{fix} 候補のペアを作成する．本手法では既存の欠陥データセット研究 [7] [10] [11] と同様に，C_{fix} 候補の親コミットを C_{bug} 候補とした．この方法を採用した理由は 2 つある．1 つ目は，C_{fix} 候補で欠陥が修正された場合，C_{fix} 候補の親コミットは欠陥を含んでいるためである．2 つ目は，C_{bug} と C_{fix} の差分を最小とするためである．マージコミットにより C_{fix} 候補の親コミットが複数存在する場合は，1 つの C_{fix} 候補から複数のペアを作成する．

ステップ 3: C_{fix} 候補の実行環境の構築

C_{fix} 候補のソースコードに対してテストを実行するために，R 本体と依存パッケージをインストールする必要がある．2.1 項で述べた通り，古いバージョンのパッケージ本体をインストールするためには適切なバージョンの依存パッケージをインストールする必要がある．本ステップでは C_{fix} 候補開発

時点における最新バージョンの R 本体と依存パッケージをインストールし，C_{fix} 候補開発時点と同じ実行環境を構築する．依存パッケージのインストール元として複数の R リポジトリが存在するが，本手法ではインストール元が CRAN と GitHub の場合に対応した．依存パッケージのインストール元が CRAN の場合，CRAN の代わりに Posit Public Package Manager（PPPM）[*4]から依存パッケージをインストールする．PPPM は CRAN のミラーサイトであり，日付ごとのアーカイブを保存している．C_{fix} 候補のコミット日時と同じ日付のアーカイブを参照し，開発時点における最新バージョンの依存パッケージをインストールする．コミット日時と同じ日付のアーカイブが存在しなかった場合，開発日時に最も近い過去の日付のアーカイブから依存パッケージをインストールする．依存パッケージのインストール元が GitHub の場合，GitHub からリポジトリをクローンし，GitHub 上の依存関係をローカル上の依存関係に置き換えて依存パッケージをインストールする．依存パッケージのバージョンは GitHub 上のコミット履歴からコミット日時を参照し，開発日時における最新バージョンを選択した．依存関係を GitHub 上からローカル上に置き換えた理由は，インストール元が GitHub のパッケージは再帰的な依存関係の制御が難しいためである．一部の C_{fix} 候補については GitHub 上の依存関係をローカル上の依存関係に置き換えることができず，実行環境の構築に失敗した．R 本体のインストールやバージョン変更には rig[*5]を使用した．

ステップ 4: C_{fix} 候補の検証

本ステップではテストを実行し，C_{fix} 候補のソースコードに欠陥が含まれていないことを確認する．テスト実行においてテストが終了しない場合があるため，テスト実行時のタイムアウトを 30 分とした．C_{fix} の必要条件は，「C_{fix} のソースコードが C_{fix} のテストにすべて成功」である．C_{fix} 候補のソースコードが 1 つでもテストに失敗した場合は C_{fix} 候補から排除する．

ステップ 5: C_{bug} 候補の実行環境の構築

本ステップではステップ 3 と同様に，C_{bug} 候補開発時点と同じバージョンの R 本体と依存パッケージをインストールし，C_{bug} 候補開発時点の実行環境を構築する．

ステップ 6: C_{bug} 候補の検証

本ステップではテストを実行し，C_{bug} 候補のソースコードに欠陥が含まれていることを確認する．ステップ 4 と同じく，テスト実行時のタイムアウトは 30 分とした．C_{bug} の必要条件は，「C_{bug} のソースコードが C_{fix} のテストに 1 つ以上失敗」である．開発者は欠陥を修正するときにその欠陥を検出でき

[*4] https://packagemanager.posit.co/client/#/
[*5] https://github.com/r-lib/rig

るテストを追加する場合があるため，C_{fix} 候補のテストを C_{bug} 候補のソースコードに適用している．C_{bug} 候補のソースコードがすべてのテストに成功した場合は C_{bug} 候補から排除する．また，すべてのテストケースが実行完了せず，テストが実行途中で終了した場合も C_{bug} 候補から排除する．

ステップ 6 まで通過した C_{bug} 候補と C_{fix} 候補のペアは C_{bug} と C_{fix} のペアとなる．C_{bug} には欠陥を含むソースコードが含まれ，C_{fix} には欠陥が修正されたソースコードと欠陥を検出できるテストが含まれる．

3.3 欠陥の手動検証

収集した欠陥候補が本当に欠陥であるか手動で検証する．検証を行うにあたり，自動で収集された全ての欠陥候補に対してコミットメッセージと issue，ソースコード，テスト結果を目視で確認した．

まず，コミットメッセージを確認し，コミットメッセージが issue を参照している場合は issue も確認した．コミットメッセージにはそのコミットにおける変更内容，issue にはその変更に至るまでの議論が主に書かれている．コミットメッセージと issue を読み，変更の意図が欠陥修正であることを確認した．逆に，変更の意図が機能開発やリファクタリング，コメントやドキュメントの更新であるペアは欠陥候補から排除した．

次に，ソースコードを確認し，変更内容がコミットメッセージや issue に記述された欠陥修正であることを確認した．ソースコードの変更内容がコミットメッセージや issue で参照している欠陥修正ではないペアは欠陥候補から排除した．また，ソースコードの変更内容が重複しているペアがいくつか存在したため，そのペアも欠陥候補から排除した．

最後に，テスト結果を確認し，欠陥が原因でテストが失敗していることを確認した．ステップ 6 では C_{bug} のソースコードが C_{fix} のテストに 1 つ以上失敗していることを確認したが，欠陥以外が原因でテストが失敗している可能性がある．そのため，テスト結果を確認することでテスト失敗の原因が欠陥であることを保証する．新機能として実装したメソッドの不足やテストの不備など，欠陥以外が原因でテストが失敗しているペアは欠陥候補から排除した．

手動検証において欠陥だと認められた C_{bug} と C_{fix} のペアは欠陥として欠陥データセットに保存する．

3.4 欠陥データセットの実装

収集した欠陥は GitHub 上で公開している[*6]．それぞれの欠陥には以下の情報が含まれる．

- C_{fix} へのリンク
- C_{bug} へのリンク

[*6] `https://github.com/kusumotolab/Rbugs`

- 関連する issue へのリンク
- 欠陥を再現する際に使用した R バージョン
- R のソースコードの修正行数
- R のソースコードの修正ファイル数
- ハンク数（連続したコード変更の数）
- C_{bug} が失敗したテストケース数

C_{fix} へのリンクから欠陥を修正したソースコードと欠陥を検出できるテストケース，C_{bug} へのリンクから欠陥を含むソースコードを取得できる．また，関連する issue へのリンクから欠陥に関する議論を確認できる．

本欠陥データセットは欠陥を再現するためのコマンドラインインタフェースを提供している．欠陥データセットの利用者はこのコマンドラインインタフェースを使用することで欠陥を容易に再現できる．まず，各欠陥の C_{bug} と C_{fix} を識別するため，欠陥 ID を定義する．各欠陥に一意の番号を振り，C_{bug} であれば番号の末尾に b を付けた識別子，C_{fix} であれば番号の末尾に f を付けた識別子を欠陥 ID と定義する．例えば，番号が 1 の欠陥の C_{bug} は `1b` で表され，番号が 2 の欠陥の C_{fix} は `2f` で表される．次に，利用者が使用するコマンドを説明する．本欠陥データセットで利用可能なコマンドは次の 4 つである．

コマンド 1：`init`

引数はない．

このコマンドでは初期化処理を行う．具体的には，欠陥の再現に必要なリポジトリをクローンし，コミット履歴をダウンロードする．このコマンドは本データセットを初めて利用する際に 1 度だけ実行する．

コマンド 2：`checkout`

引数はプロジェクト名と欠陥 ID の 2 つであり，両方必須である．

このコマンドでは指定した C_{bug} または C_{fix} にチェックアウトする．このコマンドは C_{bug} または C_{fix} の内容を確認する際に使用する．

コマンド 3：`install-deps`

引数はプロジェクト名と欠陥 ID の 2 つであり，両方必須である．このコマンドを実行する前に `checkout` を実行する必要がある．

このコマンドでは指定した C_{bug} または C_{fix} の実行に必要な依存パッケージをインストールする．

このコマンドは C_{bug} または C_{fix} の実行環境を構築する際に使用する．

コマンド 4：`test`

引数はプロジェクト名と欠陥 ID の 2 つであり，両方必須である．このコマンドを実行する前に `checkout` と `install-deps` を順に実行する必要がある．

このコマンドでは指定した C_{bug} または C_{fix} に対してテストを実行する．C_{bug} はテストが 1 つ以上失敗し，C_{fix} は全てのテストが成功する．このコマンドは欠陥を再現する際に使用する．

4 実験

4.1 実験概要

提案手法で欠陥が収集可能であることを確認するため，提案手法を実際のプロジェクトに適用し，欠陥を収集する．対象とするプロジェクトは dplyr，ggplot2，tibble の 3 プロジェクトである．対象プロジェクトの概要を表 1 に示す．スター数，コミット数，issue 数，テストケース数はいずれも 2024 年 1 月時点の値である．いずれのプロジェクトもスター数，コミット数，issue 数が多く，人気があるプロジェクトである．テストケース数は 900 を超えており，テストを使用した欠陥の収集に適しているプロジェクトである．収集対象とするコミットは 2017 年 10 月 10 日以降の全コミットである．

4.2 実験結果

4.2.1 欠陥収集過程の分析

提案手法を 3 プロジェクトに適用した結果，172 個の欠陥を収集した．ステップごとのフィルター通過数を表 2 に示し，各ステップごとに分析する．

表 1　対象プロジェクトの概要

プロジェクト名	概要	スター数	コミット数	issue 数	テストケース数
dplyr	データフレームの操作	4,600	7,757	4,927	3,496
ggplot2	データの可視化	6,200	5,216	3,802	2,428
tibble	データフレームの拡張	641	5,155	720	981

表 2　ステップごとのフィルター通過数

ステップ	dplyr	ggplot2	tibble	合計
対象期間の総コミット数	3,429	1,374	2,830	7,633
ステップ 1：C_{fix} 候補の抽出	1,050	877	555	2,482
ステップ 2：C_{bug} 候補と C_{fix} 候補のペア作成	1,218	909	740	2,867
ステップ 3：C_{fix} 候補の実行環境の構築	835	596	465	1,896
ステップ 4：C_{fix} 候補の検証	562	510	273	1,345
ステップ 5：C_{bug} 候補の実行環境の構築	541	499	260	1,300
ステップ 6：C_{bug} 候補の検証	266	207	106	579
手動検証	66	87	19	172

ステップ 1 ではコミットの件数が 7,633 件から 2,482 件に減少しており，コミットの探索範囲を全体の約 33% に削減できた．しかし，全コミットのうち約 33% が欠陥修正というのは割合として高いと思われる．これは，対象としたプロジェクトは開発期間が長く，現在は保守に関するコミットが多いことが理由の 1 つであると考える．また，ステップ 1 ではコミットメッセージに issue 番号が含まれていれば C_{fix} 候補として抽出されるが，issue の内容は考慮していない．そのため，欠陥修正以外に関する issue を扱うコミットが C_{fix} 候補に混入し，フィルター通過数が多くなった可能性がある．ステップ 1 におけるコミットメッセージのフィルター内容と該当件数を表 3 に示す．ステップ 1 を通過したコミットの中で issue 番号をコミットメッセージに含むコミットは約 84% であり，大半を占めていた．これは各プロジェクトの issue 管理が十分に行われているからだと考える．キーワードによるフィルターでは fix，error，remove が順に多かった．それに対し，problem と solve はコミットメッセージ中にほとんど現れなかった．

ステップ 2 では 2,482 件のコミットから 2,867 件の C_{fix} 候補と C_{bug} 候補のペアを得た．マージコミットにより 1 つのコミットが複数の親コミットを持つ場合があるため，ステップ 2 ではステップ 1 より件数が多くなっている．

ステップ 3 では，ステップ 2 を通過したコミットのうち約 66% の C_{fix} 候補に対して開発当時の実行環境を構築できた．ステップ 3 で排除された理由としては特定の依存パッケージのインストール失敗が最も多かった．特定の依存パッケージのインストールが失敗した原因として，R の実行環境に問題があると考える．R におけるパッケージインストールの成否は R の環境だけでなく，R の実行環境にも影響される．そのため，R 本体だけでなく，R を実行する環境も過去に戻すと依存パッケージのインストールに成功すると予想する．

ステップ 4 では，ステップ 3 を通過したコミットのうち約 71% の C_{fix} 候補に対してテストがすべて

表 3　ステップ 1 におけるコミットメッセージのフィルター内容と該当件数

フィルター内容	dplyr	ggplot2	tibble	合計
issue 番号	951	842	303	2,096
fix	308	343	111	762
error	134	59	145	338
remove	89	76	58	223
issue	32	30	9	71
repair	6	1	56	63
solve	8	8	5	21
problem	8	5	5	18

成功した．逆に，残りの約 29% の C_{fix} 候補は何らかの理由でテストがすべて成功しなかった．ステップ 4 における排除理由と該当件数を表 4 に示す．ステップ 4 での排除理由としてはテストが 1 つ以上失敗が最も多く，約半数を占めていた．テストが失敗した原因としては依存パッケージのバージョンが適切でない，テストに不備があるなど様々な可能性がある．ステップ 4 で排除された残りの半数は，テスト実行時間のタイムアウトが主な理由だった．テスト実行時間でタイムアウトが発生した原因としてはテストに不備がある，テスト実行時に何らかの問題が発生していたなどの可能性が考えられる．

ステップ 5 では，ステップ 4 を通過したコミットのうち約 97% の C_{bug} 候補に対して開発当時の実行環境を構築できた．ステップ 3 とは異なり，非常に高い割合で開発当時の実行環境の構築に成功した．これは，C_{bug} 候補は C_{fix} 候補の親コミットであり，多くの場合開発日時は近いからだと考える．ステップ 5 では C_{fix} 候補の開発環境を構築できた C_{bug} 候補のみを対象としているため，C_{bug} 候補の実行環境の構築に成功した割合は高くなったと予想される．

ステップ 6 では，ステップ 5 を通過したコミットのうち約 45% の C_{bug} 候補に対してテストが 1 つ以上失敗した．逆に，残りの約 55% の C_{bug} 候補は何らかの理由でテストが失敗しなかった．ステップ 6 における排除理由と該当件数を表 4 に示す．ステップ 6 での排除理由としてはすべてのテスト成功が最も多く，約 94% と大半を占めていた．排除理由のうちすべてのテスト成功が大半を占めたのは，ステップ 1 のフィルターが原因であると考える．ステップ 1 では欠陥修正以外に関する issue を扱うコミットも C_{fix} 候補に混入するため，実際には欠陥が含まれていない C_{bug} 候補がステップ 6 でテストがすべて成功したと考える．また，ステップ 1 では R のテスト追加を条件にしていないため，欠陥を修正したがその欠陥を検出するテストを追加しなかった場合にテストがすべて成功した可能性がある．その他の排除理由としてはテスト実行時間のタイムアウトがあったが，ステップ 4 と比較すると件数はかなり少

表 4　ステップ 4 における排除理由と該当件数

排除理由	件数
テストが 1 つ以上失敗	294
テスト実行時間のタイムアウト	229
その他	28

表 5　ステップ 6 における排除理由と該当件数

排除理由	件数
全テスト成功	676
テスト実行時間のタイムアウト	23
その他	22

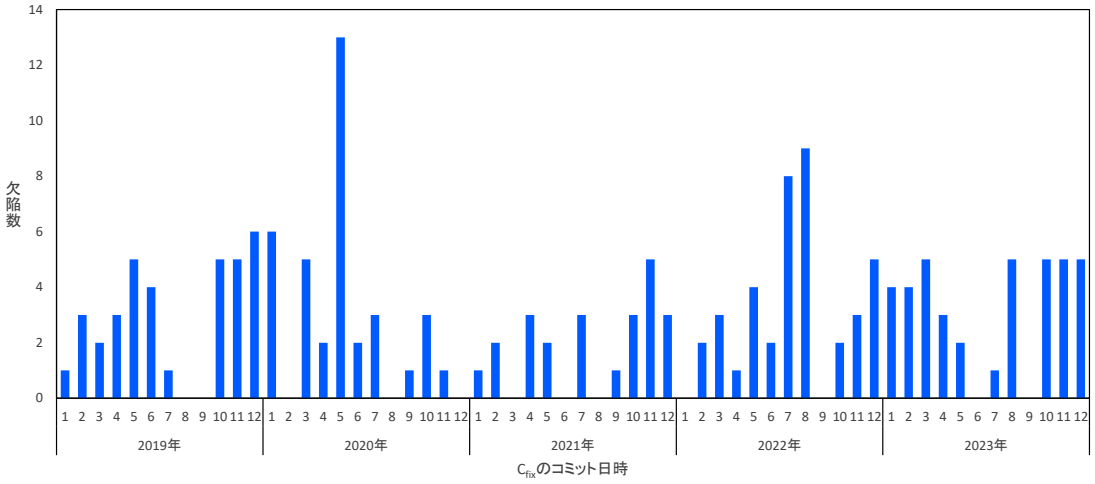

図 2　期間ごとの欠陥数

なかった．

手動検証では，ステップ 6 を通過したペアのうち約 30% が欠陥として収集された．手動検証において確認できた欠陥修正以外の変更としては機能開発が大きな割合を占めていた．その他にはリファクタリング，ソースコード中のコメントの修正，ドキュメントの更新などが確認できた．

4.2.2　期間ごとの欠陥数の分析

約 6 年間のコミット履歴から欠陥収集を行った結果として，期間ごとに収集した欠陥数を分析する．図 2 に期間ごとの欠陥数を示す．期間は C_{fix} のコミット日時で分類した．2019 年 1 月から 2023 年 12 月までの約 5 年間にかけて，欠陥を収集できていることがわかる．2018 年以前のコミットからは欠陥を全く収集できなかった．これは古いコミットほど実行環境の構築が難しく，欠陥収集のステップ 3 で実行環境の構築に失敗してしまうためである．

4.2.3　欠陥データの分析

収集した欠陥データの分析を行う．分析内容は修正の大きさ，修正箇所の数，欠陥の複雑さの 3 つである．

まず，修正の大きさを分析する．図 3 に修正行数ごとの欠陥数を示す．修正行数は R の各ソースファイルの変更行数を合計した値であり，空行やコメント行は除外して計算した．修正行数が 1 行以上 10 行以下の欠陥は 106 個であり，割合として半分以上を占める．このことから，修正の大きさが小さい欠陥が多く含まれていることがわかる．また，修正行数が 26 行以上の欠陥は 25 個であり，これらの欠陥

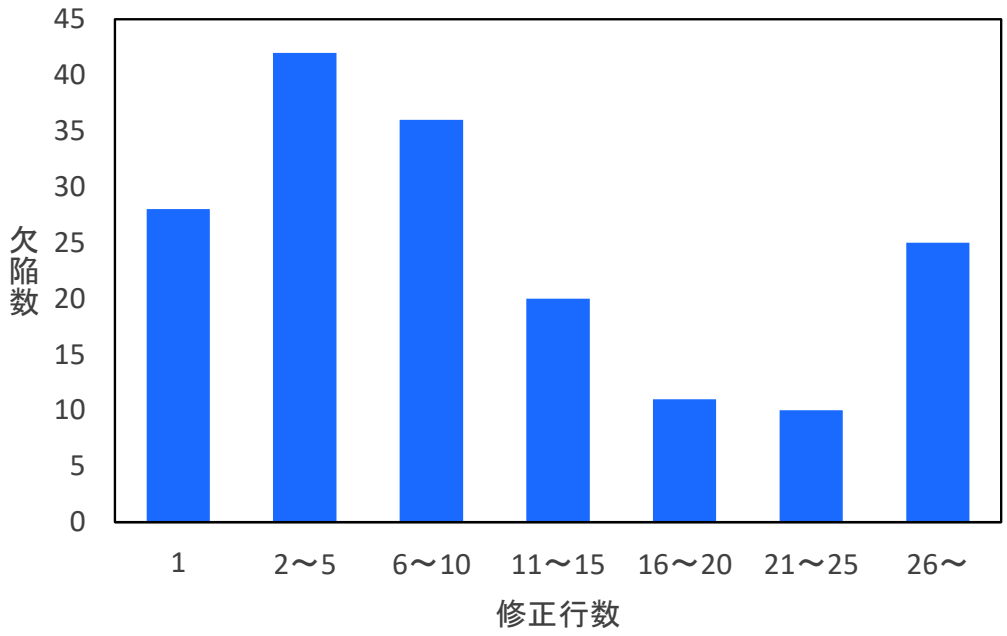

図 3　修正行数ごとの欠陥数

は修正に大きな変更を要する．よって，本データセットには修正の大きさが小さい欠陥から大きい欠陥まで含まれており，修正の大きさが多様であるとわかる．

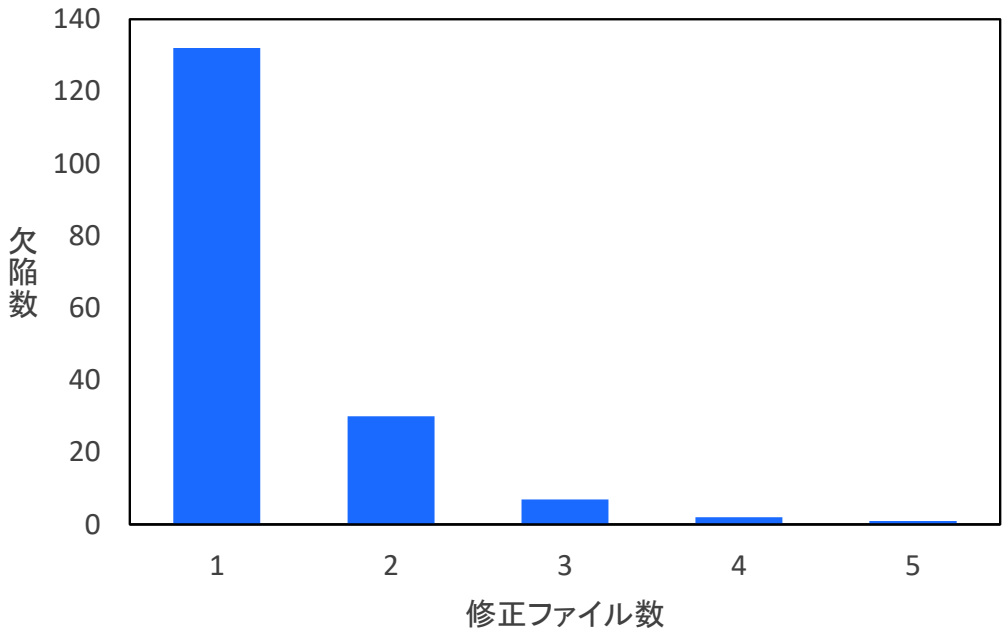

図 4　修正ファイル数ごとの欠陥数

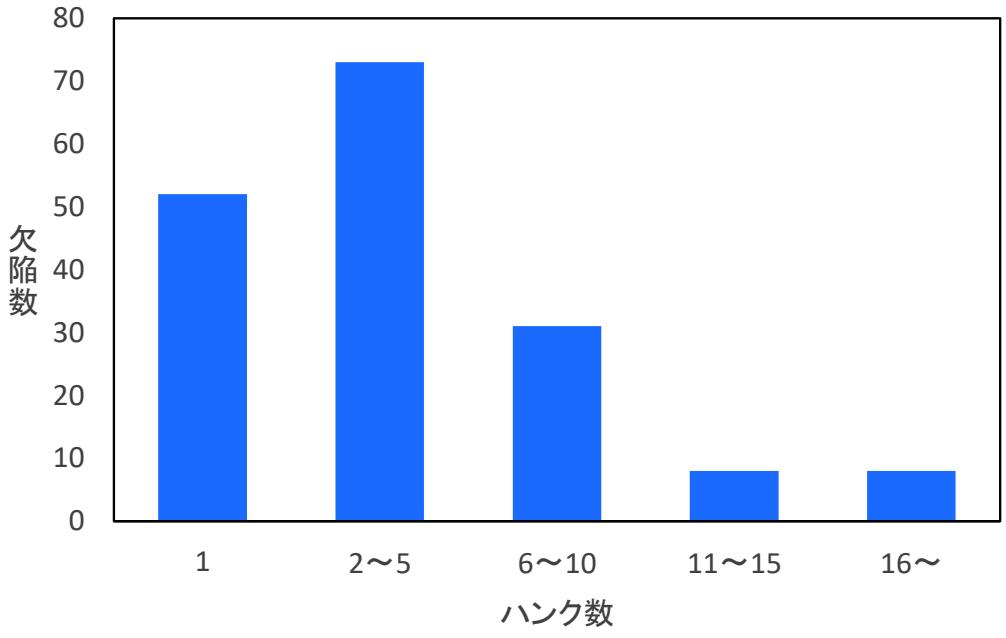

図 5　ハンク数ごとの欠陥数

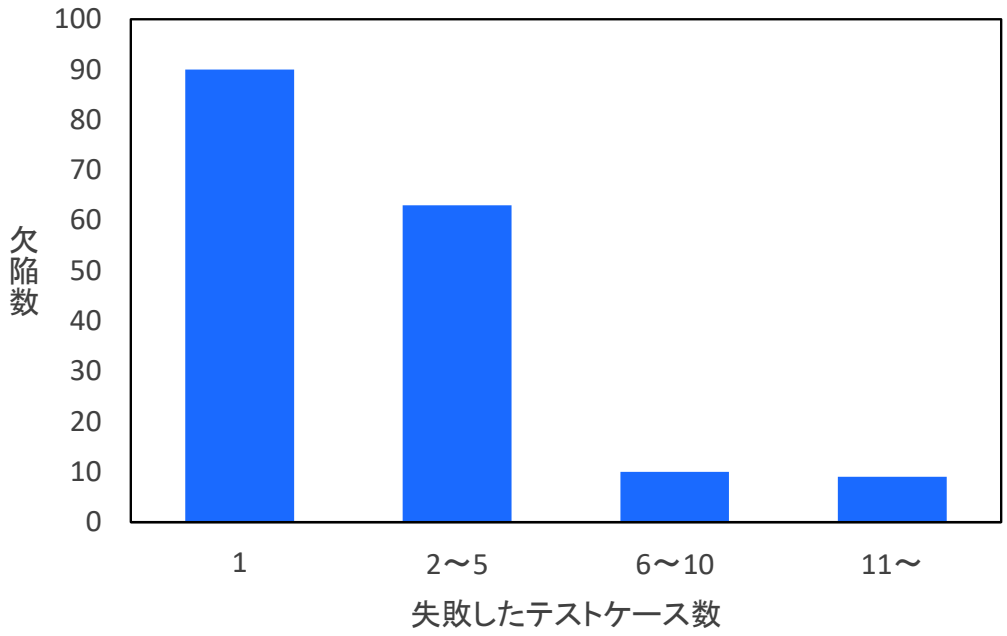

図 6　失敗したテストケース数ごとの欠陥数

次に，修正箇所の数を分析する．図 4 に修正ファイル数ごとの欠陥数を示す．修正ファイル数は R のソースフォルダの中で変更されているファイル数である．修正ファイル数が 1 つの欠陥は 132 個であり，7 割以上を占める．修正ファイル数が 1 つの欠陥は修正範囲が 1 ファイル内に限定されるため，欠陥研究で扱いやすい欠陥である．図 5 にハンク数ごとの欠陥数を示す．ハンク数は連続したコード変更の数であり，修正ファイルが複数ある場合は各ファイルのハンク数を合計した値である．ハンク数が 1 の欠陥は 52 個，ハンク数が 2 以上 5 以下の欠陥は 73 個であり，これらの欠陥は修正箇所が少ない欠陥である．修正ファイル数の分析とハンク数の分析から，本データセットに含まれる欠陥の 7 割以上は修正箇所が少ない欠陥であるとわかる．

最後に，欠陥の複雑さを分析する．図 6 に失敗したテストケース数ごとの欠陥数を示す．失敗したテストケース数は C_{bug} が C_{fix} のテストに失敗したテストケース数である．失敗したテストケース数が多いほどその欠陥は複雑だと予想される．失敗したテストケース数が 1 つの欠陥数は 90 個，失敗したテストケース数が 2 つの欠陥数は 63 個であり，大半の欠陥は 2 つ以下のテストケースで欠陥を検出していることがわかる．失敗したテストケース数が 6 以上の欠陥は 19 個であり，少数であった．欠陥を検出するためのテストケースが少ない欠陥が多いため，本データセットに含まれる欠陥は単純な欠陥が多いと予想される．

5　考察

実験結果より，本手法で実際のプロジェクトから欠陥を収集できると言える．まず，本手法では 3 プロジェクトから 172 個の欠陥を収集できた．これは平均すると 1 プロジェクトから約 57 個の欠陥を収集したことになり，実際のプロジェクトから一定数の欠陥を収集できていることがわかる．次に，期間ごとの欠陥数の分析より，本手法では過去 5 年間にかけて欠陥を収集できた．過去の実行環境の構築は難しいため，過去の欠陥を再現可能な状態で収集することは非常に困難である．しかし，本手法であれば 5 年前の開発履歴から欠陥を再現可能な状態で収集できる．

欠陥データの分析より，本データセットに含まれる欠陥は修正の規模が多様であることがわかる．まず，修正行数が 1 行の欠陥が 28 個ある．1 行の欠陥は SStuBs（simple stupid bugs） [16] と呼ばれ，SStuBs を対象とした研究も行われている [17] [18] [19] [20]．本データセットに含まれる SStuBs の個数は少ないが，SStuBs の研究に役立つ可能性がある．次に，ハンク数が 1 個の欠陥が 52 個ある．欠陥限局や自動プログラム修正などの研究において，最初の題材としてハンク数が 1 個の欠陥が利用される．よって，本データセットは欠陥研究の初期段階として利用できると考える．また，ハンク数が 2 以上 5 以下の欠陥は 73 個含まれている．そのため，研究の初期段階からより複雑な欠陥を対象とした場合にも本データセットを利用できると考える．

6 妥当性への脅威

6.1 内的妥当性への脅威

欠陥収集の過程において，本手法で収集できなかった欠陥も存在すると考える．まず，ステップ 1 で C_{fix} 候補として抽出されなかったペアからは欠陥を収集できない．本研究ではコミットメッセージや issue の管理が適切に行われているプロジェクトを収集対象として選択したため，ステップ 1 による欠陥の漏れは少ないと考える．次に，ステップ 3 やステップ 5 で実行環境の構築に失敗したペアからは欠陥を収集できない．特に，2018 年以前のペアは大半が実行環境の構築に失敗してしまい，欠陥を収集できなかった．依存パッケージのインストール方法や R の実行環境を改善することで収集可能な欠陥が増えると考える．最後に，ステップ 4 やステップ 6 で期待通りのテスト結果を得られず排除したペアに，収集すべき欠陥が混入している可能性がある．ステップ 4 やステップ 6 で期待通りのテスト結果が得られなかった原因の 1 つとして，ステップ 3 やステップ 5 でインストールした依存パッケージのバージョンが適切でなかった可能性がある．依存パッケージをインストールする際に使用した PPPM には全ての日付のアーカイブが保存されているわけではない．そのため，PPPM からインストールしたバージョンと本来インストールすべきバージョンにずれが生じ，期待通りのテスト結果が得られなかった可能性がある．

本来欠陥ではないにも関わらず，欠陥収集の過程においてステップ 6 まで通過したペアもあった．ステップ 6 では C_{bug} 候補が C_{fix} 候補のテストに 1 つ以上失敗することを確認した．しかし，ソースコードに含まれる欠陥以外の要因でテストが失敗する場合があるため，欠陥でないペアもステップ 6 を通過してしまう．このような欠陥でないペアを収集過程で排除したい場合，ステップ 1 の C_{fix} 候補の抽出において条件を厳しくすることが最も有効であると考える．例えば，「コミットメッセージに issue 番号が含まれ，その issue に欠陥ラベルが付与されている」をステップ 1 の C_{fix} 候補の条件とすると，欠陥でないペアを大幅に排除できる．しかし，欠陥ラベルが付与されていないが実際には欠陥を扱っている issue が存在した場合，そこで扱われている欠陥はステップ 1 の条件を満たさず，収集できない．このように，ステップ 1 の C_{fix} 候補の条件を厳しくすると本来 C_{fix} 候補として抽出すべきコミットを取り逃がす可能性がある．本研究ではより多くの欠陥を収集したいと考えたため，ステップ 1 ではより多くの C_{fix} 候補を抽出できるようにフィルターを設定し，手動検証で欠陥以外を排除する方針を選択した．

手動検証は著者 1 名で行った．そのため，欠陥の手動検証において著者の主観や先入観が含まれている可能性がある．手動検証においてはコミットメッセージや issue をよく確認し，欠陥修正以外の変更内容を欠陥修正だと誤分類することを避けた．特に issue を確認する際は，issue で議論している開発者や利用者が報告内容をどのように捉えているかを読み取り，可能な限り主観を排除した．

本研究では収集した欠陥に対して，ペアの差分が欠陥修正のみであることを保証していない．収集し

たペアの差分は欠陥修正以外に，リファクタリングや機能追加などの欠陥修正とは無関係な変更を含んでいる可能性がある．欠陥研究への応用を考えると，ペアの差分は欠陥修正のみであることが望ましい．Just ら [7] の研究ではペアの差分から欠陥修正とは無関係な変更を取り除き，差分を最小化している．また，Widyasari ら [9] の研究では差分の最小化を行わず，ペアの差分に欠陥修正とは無関係な変更が含まれるペアは欠陥候補から排除している．

6.2 外的妥当性への脅威

欠陥収集は 3 プロジェクトを対象として実行した．そのため，収集した欠陥に偏りが生じている可能性がある．本研究では対象プロジェクトとして異なるドメインのプロジェクトを選択し，欠陥の偏りを可能な限り避けた．収集した欠陥の偏りを避けるためには対象プロジェクト数とドメインの種類を増やす必要がある．

欠陥収集における C_{fix} 候補の抽出において，コミットメッセージと issue から欠陥修正コミットの候補を抽出している．そのため，コミットメッセージや issue が適切に管理されていないプロジェクトに対して本手法を適用すると，欠陥を十分に収集できない可能性がある．

7 関連研究

7.1 R 言語を対象とした欠陥研究

Ahmed ら [21] は Stack Overflow と GitHub から R 言語と Python の欠陥を収集し，欠陥の分類と分析を行った．この研究では Stack Overflow から R 言語の欠陥を 559 個と Python の欠陥を 1210 個，GitHub から R 言語の欠陥を 544 個と Python の欠陥を 877 個収集しており，欠陥データセットとして公開している．この研究の目的は欠陥がデータ分析に与える影響を明らかにすることであり，本研究とは目的が大きく異なる．また，Ahmed らが収集した欠陥は必ず欠陥を検出するテストが付属しているとは限らず，欠陥の収集過程においてテスト実行による欠陥の再現を行っていない．それに対し，本研究では欠陥を検出するテストによって欠陥の再現性を保証しており，欠陥の再現が可能である．

7.2 欠陥データセットに関する研究

最初に広く利用された欠陥データセットとして Siemens benchmark suite [22] がある．Siemens benchmark suite には C 言語の欠陥が 130 個含まれている．しかし，これらの欠陥は 7 個のプログラムに対して人工的に生成された欠陥であり，実際のプロジェクトの開発履歴から収集した欠陥ではない．

C 言語を対象とした欠陥データセットとして ManyBugs と IntroClass がある [8]．ManyBugs には 9 個のプロジェクトから収集した 185 個の欠陥が含まれており，IntroClass には 6 個の学生課題から収集した 998 個の欠陥が含まれている．ManyBugs は大規模なプログラムを対象とした欠陥研究，IntroClass は小規模なプログラムを対象とした欠陥研究をサポートしている．

他に C 言語を対象とした欠陥データセットとして Codeflaws [23] がある．Codeflaws はプログラミングコンテストである Codeforces から 3902 個の欠陥を収集した研究である．

Java を対象とした欠陥データセットとして，iBugs [24] がある．iBugs は Java プロジェクトから 369 個の欠陥を収集した研究である．テストケースが付属している欠陥は 369 個中 223 個であり，残りの欠陥に対してはテストケースが付属していない．

Java を対象とした最も有名な欠陥データセットとして Defects4J [7] がある．Defects4J は Java の 5 プロジェクトから 357 個の欠陥を収集した研究である．Defects4J は更新されており，現在では 17 プロジェクトから収集した 835 個の欠陥が公開されている．Defect4J に含まれる全ての欠陥にはテストケースが付属している．また，全ての欠陥に対して，欠陥を含むソースコードと欠陥を修正したソースコードの差分に欠陥修正以外の変更が含まれないことを保証している．

他に Java を対象とした欠陥データセットとして Bugs.jar [25] がある．Bugs.jar は Java の 8 プロジェクトから 1158 個の欠陥を収集した研究である．Bugs.jar は Defects4J とは異なるドメインのプロジェクトを対象としている．

CI を利用して Java の欠陥を収集した研究として BEARS [26] がある．BEARS は Travis CI のログを元に，72 プロジェクトから 251 個の欠陥を収集した研究である．BEARS の手法では欠陥候補を収集する際に issue などの課題管理システムやコミットメッセージを利用しない．そのため，課題管理システムを利用していないプロジェクトからも欠陥を収集できるという利点がある．

Python を対象とした欠陥データセットとして BugsInPy [9] がある．BugsInPy は Python の 17 プロジェクトから 493 個の欠陥を収集した研究である．BugsInPy は Defects4J に大きく影響されており，Defects4J と同品質の欠陥データセットの構築を目指した研究である．

JavaScript を対象とした欠陥データセットとして BugsJS [10] がある．BugsJS は JavaScript の 10 プロジェクトから 453 個の欠陥を収集した研究である．

JVM 言語を対象とした欠陥データセットとして Defexts [11] がある．Defexts は Kotlin のプロジェクトから 225 個の欠陥，Groovy のプロジェクトから 301 個の欠陥を収集した研究である．

8　おわりに

本研究では R 言語における欠陥研究で利用可能な欠陥データセットの構築を目的として，3 プロジェクトを対象に欠陥収集を実行した．その結果，過去 5 年間の開発履歴から 172 個の欠陥を収集した．収集した欠陥に対してメトリクスに関する分析を行い，欠陥の規模が多様であることを確認した．また，欠陥を再現するためのコマンドラインインタフェースを実装し，欠陥と共に公開した．本データセットを利用することで，欠陥研究を行う研究者は欠陥を容易に再現できる．

今後の課題として以下の 3 つを挙げる．1 つ目は，欠陥収集手順における実行環境の構築の改善である．実行環境の構築に失敗して欠陥候補から排除されたペアに，本来収集すべき欠陥が含まれていると考える．実行環境の構築を改善することで，より多くの欠陥を収集できると考える．また，より古いコミットから欠陥を収集するためにも，実行環境の構築の改善が必要である．2 つ目は，欠陥におけるペアの差分の最小化である．欠陥研究への応用を考えると，ペアの差分は欠陥修正のみであることが望ましい．ペアの差分から欠陥修正とは無関係な変更を取り除くことで，欠陥研究でより使いやすいデータセットになると考える．3 つ目は，欠陥の拡充である．欠陥データセットには多種多様な欠陥が含まれていることが望ましい．欠陥収集において対象プロジェクト数とドメインの種類を増やし，欠陥データセットに含まれる欠陥の偏りを可能な限り避けることが必要であると考える．

謝辞

本研究の遂行にあたり，多くの方々にご指導とご支援を賜りました．

本研究において，終始暖かく見守ってくださりました楠本真二教授に感謝申し上げます．中間報告では客観的な目線からご指導及びご助言をいただき，大変参考になりました．研究におけるご指導以外にも様々な面でお世話になりました．

本研究の全過程において，終始丁寧かつ熱心なご指導及びご助言を賜りました柗本真佑助教に深く感謝申し上げます．研究で壁にぶつかった際には個別に時間を設けていただき，常に親身になって私の研究を指導していただきました．研究発表における話し方や論文の書き方など，伝わる表現についても多くを学ばせていただきました．

研究の議論や輪講において，貴重なご指導及びご助言を賜りました肥後芳樹教授に感謝申し上げます．

研究生活を支えていただいた，事務補佐員の橋本美砂子氏に感謝申し上げます．私が体調を崩した際には気にかけていただき，心の支えになりました．

研究生活を盛り上げていただきました，研究室の皆様に感謝いたします．研究の議論から雑談まで，様々な面でお世話になりました．

最後に，様々な面で私の生活を支えていただいた家族にも感謝いたします．

参考文献

[1] Sobreira, V., Durieux, T., Madeiral, F., Monperrus, M. and Almeida Maia, de M.: Dissection of a bug dataset: Anatomy of 395 patches from Defects4J, in *Proc. International Conference on Software Analysis, Evolution and Reengineering*, pp. 130–140 (2018).

[2] B. Le, T.-D., Lo, D., Le Goues, C. and Grunske, L.: A Learning-to-Rank Based Fault Localization Approach Using Likely Invariants, in *Proc. International Symposium on Software Testing and Analysis*, pp. 177–188 (2016).

[3] Laghari, G., Murgia, A. and Demeyer, S.: Fine-Tuning Spectrum Based Fault Localisation with Frequent Method Item Sets, in *Proc. International Conference on Automated Software Engineering*, pp. 274–285 (2016).

[4] Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M. D., Pang, D. and Keller, B.: Evaluating and Improving Fault Localization, in *Proc. International Conference on Software Engineering*, pp. 609–620 (2017).

[5] Le, X. B. D., Lo, D. and Le Goues, C.: History Driven Program Repair, in *Proc. International Conference on Software Analysis, Evolution, and Reengineering*, pp. 213–224 (2016).

[6] Xin, Q. and Reiss, S. P.: Leveraging syntax-related code for automated program repair, in *Proc. International Conference on Automated Software Engineering*, pp. 660–670 (2017).

[7] Just, R., Jalali, D. and Ernst, M. D.: Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs, in *Proc. International Symposium on Software Testing and Analysis*, pp. 437–440 (2014).

[8] Le Goues, C., Holtschulte, N., Smith, E. K., Brun, Y., Devanbu, P., Forrest, S. and Weimer, W.: The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs, *Transactions on Software Engineering*, Vol. 41, No. 12, pp. 1236–1256 (2015).

[9] Widyasari, R., Sim, S. Q., Lok, C., Qi, H., Phan, J., Tay, Q., Tan, C., Wee, F., Tan, J. E., Yieh, Y., Goh, B., Thung, F., Kang, H. J., Hoang, T., Lo, D. and Ouh, E. L.: BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies, in *Proc. Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1556–1560 (2020).

[10] Gyimesi, P., Vancsics, B., Stocco, A., Mazinanian, D., Beszédes, Á., Ferenc, R. and Mesbah, A.: BugsJS: a Benchmark of JavaScript Bugs, in *Proc. International Conference on Software Testing, Validation and Verification*, pp. 90–101 (2019).

[11] Benton, S., Ghanbari, A. and Zhang, L.: Defexts: A Curated Dataset of Reproducible Real-World Bugs for Modern JVM Languages, in *Proc. International Conference on Software Engineering*, pp. 47–50 (2019).

[12] Lai, J., Lortie, C. J., Muenchen, R. A., Yang, J. and Ma, K.: Evaluating the popularity of R in ecology, *Journal on Ecosphere*, Vol. 10, No. 1, p. e02567 (2019).

[13] PYPL PopularitY of Programming Language Index (accessed 2023-09-07), `https://pypl.github.io/PYPL.html`.

[14] Vidoni, M.: Software Engineering and R Programming: A Call for Research, *Journal on R J.*, Vol. 13, No. 2, p. 600 (2021).

[15] Decan, A., Mens, T., Claes, M. and Grosjean, P.: When GitHub Meets CRAN: An Analysis of Inter-Repository Package Dependency Problems, in *Proc. International Conference on Software Analysis, Evolution, and Reengineering*, pp. 493–504 (2016).

[16] Karampatsis, R.-M. and Sutton, C.: How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset, in *Proc. International Conference on Mining Software Repositories*, pp. 573–577 (2020).

[17] Kamienski, A. V., Palechor, L., Bezemer, C.-P. and Hindle, A.: PySStuBs: Characterizing Single-Statement Bugs in Popular Open-Source Python Projects, in *Proc. International Conference on Mining Software Repositories*, pp. 520–524 (2021).

[18] Richter, C. and Wehrheim, H.: TSSB-3M: mining single statement bugs at massive scale, in *Proc. International Conference on Mining Software Repositories*, pp. 418–422 (2022).

[19] Hua, J. and Wang, H.: On the Effectiveness of Deep Vulnerability Detectors to Simple Stupid Bug Detection, in *Proc. International Conference on Mining Software Repositories*, pp. 530–534 (2021).

[20] Mosolygó, B., Vándor, N., Antal, G. and Hegedűs, P.: On the Rise and Fall of Simple Stupid Bugs: a Life-Cycle Analysis of SStuBs, in *Proc. International Conference on Mining Software Repositories*, pp. 495–499 (2021).

[21] Ahmed, S., Wardat, M., Bagheri, H., Cruz, B. D. and Rajan, H.: Characterizing Bugs in Python and R Data Analytics Programs (2023).

[22] Hutchins, M., Foster, H., Goradia, T. and Ostrand, T.: Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria, in *Proc. International Conference on Software Engineering*, pp. 191–200 (1994).

[23] Tan, S. H., Yi, J., Yulis, , Mechtaev, S. and Roychoudhury, A.: Codeflaws: a programming

competition benchmark for evaluating automated program repair tools, in *Proc. International Conference on Software Engineering Companion*, pp. 180–182 (2017).

[24] Dallmeier, V. and Zimmermann, T.: Extraction of bug localization benchmarks from history, in *Proc. International Conference on Automated Software Engineering*, pp. 433–436 (2007).

[25] Saha, R. K., Lyu, Y., Lam, W., Yoshida, H. and Prasad, M. R.: Bugs.jar: a large-scale, diverse dataset of real-world Java bugs, in *Proc. International Conference on Mining Software Repositories*, pp. 10–13 (2018).

[26] Madeiral, F., Urli, S., Maia, M. and Monperrus, M.: BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies, in *Proc. International Conference on Software Analysis, Evolution and Reengineering*, pp. 468–478 (2019).