社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

R 言語に対する欠陥データセット構築の試み
—dplyr プロジェクトを題材として—

石野　太一† 　松本　真佑† 　楠本　真二†

† 大阪大学大学院情報科学研究科
E-mail: †{t-ishino,shinsuke,kusumoto}@ist.osaka-u.ac.jp

あらまし　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，欠陥データセットの構築を試みる．その結果，dplyr プロジェクトから欠陥の規模が多様な 176 個の欠陥を収集した．

キーワード　R 言語，欠陥データセット

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，欠陥を動的に再現可能である．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．2023 年 9 月時点で Defects4J の被引用数は 1,100 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]．2023 年 9 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 7 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の促進である．そのために，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，欠陥データセットの構築を試みる．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が保証されている欠陥を収集する．また，欠陥収集の工程をすべて自動化したため，将来的な欠陥データの拡充にも対応可能である．本研究では実際に R プロジェクトから欠陥が収集可能であることを確認するため，dplyr[1] プロジェクトから欠陥の収集を試みた．その結果，欠陥の規模が多様な 176 個の欠陥を収集した．

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

(注1)：https://github.com/tidyverse/dplyr

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[2]である．CRAN は R 言語の公式リポジトリであり，2023 年 9 月時点で 19,000 以上のパッケージが公開されている．CRAN の他には Bioconductor[3]，R-Forge[4], 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，依存パッケージの後方互換性のない変更により R パッケージが壊れてしまうことがある [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．

3. 欠陥の収集方法

本研究では，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．

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修正コミット（C_{fix}）候補と欠陥含有コミット（C_{bug}）候補のペアを抽出する．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．

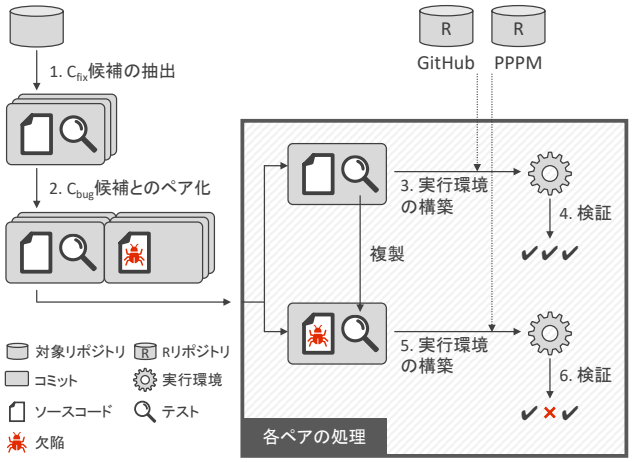

図 1: 欠陥収集の流れ

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．C_{fix} 候補で欠陥が修正された場合，C_{fix} 候補の親コミットである C_{bug} 候補は欠陥を含んでいるはずである．ただし，C_{bug} 候補はそのコミットで欠陥が混入したとは限らない．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）[5] 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．

（注2）：https://cran.r-project.org/
（注3）：https://www.bioconductor.org/
（注4）：https://r-forge.r-project.org/
（注5）：https://packagemanager.posit.co/client/#/

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[6]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 5 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 5 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} のペアとなり，欠陥データとしてデータセットに保存される．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．

4. 欠陥収集手法の試行

4.1 dplyr

本手法で欠陥が収集可能であることを確認するため，dplyr プロジェクトから欠陥収集を試みる．dplyr はデータフレームの操作に特化した R パッケージであり，使用率の高いパッケージの 1 つである．2023 年 9 月時点でスター数が 4,500，コミット数が 7,700，issue 数が 4,800 を超えており，人気があるプロジェクトである．最新コミットではテストケース数が 3,400 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 2023 年 6 月から過去 5 年間の計 2,762 コミットである．

4.2 欠陥収集過程の分析

本手法を dplyr に適用した結果，176 ペアがすべてのステップを通過し，欠陥データとして収集された．ステップごとのフィルター通過数を表 1 に示し，各ステップごとに分析する．

ステップ 1 ではコミットの件数が 2,762 コミットから 926 コミットに減少しており，コミットの探索範囲を全体の約 34% に削減できた．しかし，全コミットのうち約 34% 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 dplyr が成熟したプロジェクトであり，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 2 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 91% であり，大半を占めていた．これは dplyr 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．problem，solve，repair はコミットメッセージ中にほとんど現れなかった．

表 1: ステップごとのフィルター通過数

ステップ	件数
過去 5 年間の総コミット数	2,762 コミット
ステップ 1：C_{fix} 候補の抽出	926 コミット
ステップ 2：C_{bug} 候補と C_{fix} 候補のペア作成	1,048 ペア
ステップ 3：C_{fix} 候補の実行環境の構築	676 ペア
ステップ 4：C_{fix} 候補の検証	383 ペア
ステップ 5：C_{bug} 候補の実行環境の構築	376 ペア
ステップ 6：C_{bug} 候補の検証	176 ペア

表 2: 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	件数
issue 番号	846
fix	274
error	122
remove	81
issue	24
problem	8
solve	8
repair	6

（注6）：https://github.com/r-lib/rig

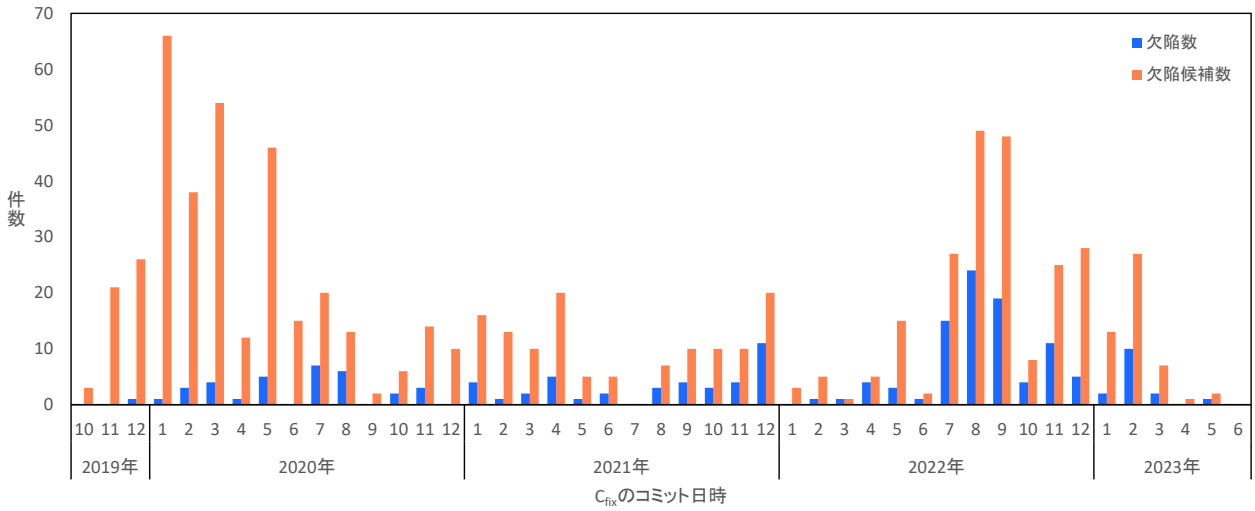

図 2: 期間ごとの欠陥数と欠陥候補数

ステップ 2 では 926 件のコミットから 1,048 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 65% 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 R のクラッシュが最も多かった．その他の理由としては依存パッケージのコンパイル失敗，依存関係の解決の失敗が多かった．依存パッケージのインストール時に R のクラッシュや依存パッケージのコンパイル失敗が発生した原因として，R の実行環境に問題があると考える．R 本体だけでなく，R を実行する環境も過去に戻すとこれらのエラーを解決できると予想する．依存パッケージのインストール時に依存関係の解決の失敗が発生した原因として，依存パッケージのインストール方法に問題があると考える．本手法では開発日時を元に依存パッケージをインストールしたが，インストールされ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．依存関係の解決に失敗した場合は，別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ことで依存関係の解決に成功する可能性がある．

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 57% の C_{fix} 候補に対してテストがすべて成功した．ステップ 4 で排除された理由としてはテスト実行時間のタイムアウトが最も多く，排除理由の約半数を占めていた．その他の理由としてはテストが 1 つ以上失敗，dplyr パッケージのコンパイル失敗が多かった．テスト実行時のタイムアウトは 5 分としたが，時間が短すぎた可能性がある．タイムアウトによって排除されたコミットに対しては，タイムアウトの時間を調整したうえで再度テストを行う必要があると考える．テストが 1 つ以上失敗したコミットもあるが，テストが失敗した原因として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 5 では，ステップ 4 を通過したコミットのうち約 98% 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 47% の C_{bug} 候補に対してテストが 1 つ以上失敗した．ステップ 6 で排除された理由としてはすべてのテスト成功が最も多く，排除理由の約 7 割を占めていた．その他の理由としては dplyr パッケージのコンパイル失敗，テスト実行時間のタイムアウトが多かった．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ケースが 2 件だけあった．排除理由のうちすべてのテスト成功が 7 割を占めたのは，ステップ 1 のフィルターの緩さ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．

4.3 期間ごとの欠陥数の分析

過去 5 年間のコミットから欠陥収集を試みた結果として，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と欠陥候補数を示す．欠陥数は全ステップを通過した C_{bug} と C_{fix} のペア数であり，欠陥候補数はステップ 2 を通過した C_{bug} 候補と C_{fix} 候補のペア数である．期間は C_{fix} のコミット日時で分類した．2019 年 12 月から 2023 年 5 月までの約 3 年半にかけて欠陥を収集できたが，収集した欠陥数は月によってばらつきがある．日付が古いほどその月の欠陥候補数に対して収集できた欠陥数が少ない傾向にある．特に 2020 年 6 月以前は欠陥候補数と比較して収集した欠陥数は非常に少なくなっ

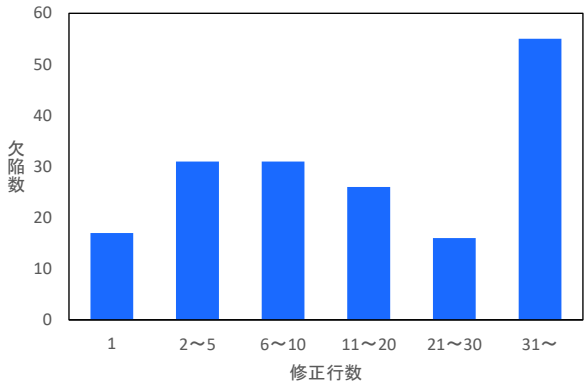

図 3: 修正行数ごとの欠陥数

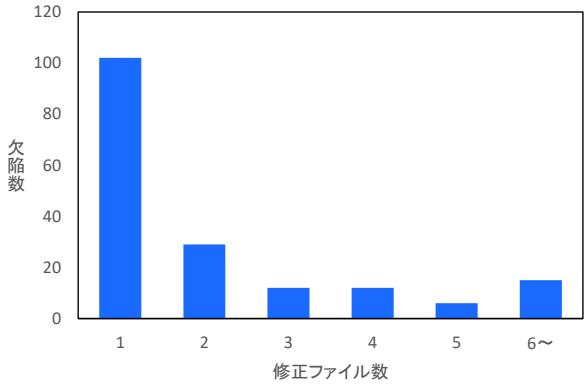

図 4: 修正ファイル数ごとの欠陥数

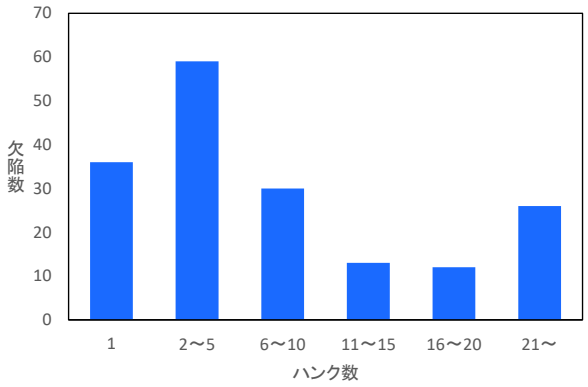

図 5: ハンク数ごとの欠陥数

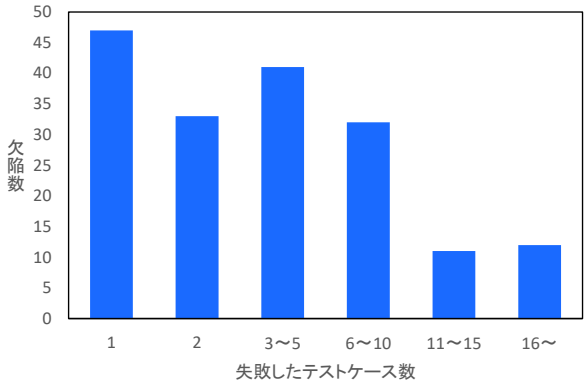

図 6: 失敗したテストケース数ごとの欠陥数

ており，コミット日時が古いほど欠陥の収集が難しいとわかる．

4.4 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 行の欠陥は 17 件，修正行数が 2 行以上 5 行以下の欠陥は 31 件であり，これらの欠陥は修正の大きさが小さい欠陥である．また，修正行数が 31 行以上の欠陥は 55 件であり，これらの欠陥は修正に大きな変更を要する．本データセットに含まれる欠陥は修正の大きさが多様であるとわかる．

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 102 件であり，半数以上を占める．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 36 件，ハンク数が 2 以上 5 以下の欠陥は 59 件であり，これらの欠陥は修正箇所が少ない欠陥である．本データセットに含まれる欠陥は半数以上が修正箇所の数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考えられる．失敗したテストケース数が 1 つの欠陥数は 47 件，失敗したテストケース数が 2 つの欠陥数は 33 であり，約半数の欠陥は 2 つ以下のテストケースで欠陥を検出しているとわかる．失敗したテストケース数が 11 以上の欠陥は少数であった．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. 制　　約

5.1 手法に関する制約

C_{fix} 候補と C_{bug} 候補を決定するにあたり，簡易的な方法を採用している．C_{fix} 候補は issue 番号とキーワードを用いて抽出した．この方法は多数の欠陥修正コミットをカバーできる半面，欠陥修正以外のコミットも C_{fix} 候補に混入してしまう．本研究では C_{fix} 候補を抽出する際に issue の内容は考慮していないが，Gyimesi ら [10] の手法では issue に付与されているラベルを確認している．欠陥修正以外のコミットの混入を防ぐために，C_{fix} 候補の抽出方法を改善する余地がある．また，C_{bug} 候補として C_{fix} 候補の親コミットを採用しており，これ

も簡易的な方法である．この方法は既存の欠陥データセット研究 [7] [10] [11] でも採用されており，C_{bug} と C_{fix} の差分が小さいという点では優れている．しかし，C_{bug} は欠陥が混入したコミットとは限らず，完成した欠陥データセットは欠陥が混入したコミットや欠陥の潜伏期間などを分析する研究には適していないデータセットとなる．

本研究では dplyr から欠陥を収集したが，他のプロジェクトからは欠陥を十分に収集できない可能性がある．コミットメッセージや issue 管理が適切に行われており，テストが十分に行われているプロジェクトであれば本手法の適用は可能である．しかし，他のプロジェクトの過去のコミットに対してテスト実行環境の構築やテスト実行が問題なく行えるかについては不明である．

5.2 収集した欠陥データに関する制約

本研究で収集した欠陥データは，欠陥がソースコードに含まれていない可能性がある．本研究では，テスト失敗の原因はソースコードに含まれる欠陥だと仮定している．しかし，実際にはテスト自体の欠陥やテスト実行環境の不備など，ソースコードに含まれる欠陥以外の理由でテストが失敗する場合がある．ソースコードに含まれる欠陥のみを収集するためには手動検証のステップを追加し，テスト失敗の原因がソースコードに含まれる欠陥であることを確認する必要がある．既存の欠陥データセット研究 [7] [9] [10] では手動検証を行い，欠陥がソースコードに含まれていることを確認している．また，R パッケージは一部を C や C++ による実装が可能であるが，本研究ではそれらのソースファイルの変更を考慮していない．そのため，R のソースコードではなく C や C++ のソースコードに欠陥が含まれている可能性がある．実際，本研究で収集した 176 個の欠陥のうち C_{fix} に C や C++ のソースコードの変更が含まれる欠陥は 15 件存在した．割合としては少ないが，これら 15 件についてはさらなる検証が必要である．

本研究では収集した欠陥データに対して，ペアの差分が欠陥修正のみであると保証していない．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，欠陥データセットはペアの差分が欠陥修正のみである修正パッチを含んでいることが望ましい．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データセットから排除している．

6. おわりに

本研究では R 言語における欠陥研究の促進を目的として，欠陥データセットの構築を試みた．その結果，dplyr プロジェクトから欠陥の規模が多様な 176 個の欠陥を収集した．

今後の課題として以下の 2 つを挙げる．1 つ目は欠陥データセット構築方法の改善である．C_{fix} 候補と C_{bug} 候補の決定方法や収集した欠陥の検証方法など，欠陥収集過程において多くの改善点がある．欠陥データセットの信頼性と価値を高めるために，これらの改善に取り組む予定である．2 つ目は欠陥データの拡充である．多種多様な欠陥をデータセットに含めるためには様々なプロジェクトから欠陥を収集する必要があると考える．本手法を他のプロジェクトへ適用し，多種多様な欠陥の収集を予定している．

謝辞 本研究の一部は，JSPS 科研費（JP21H04877, JP20H04166, JP21K18302, JP21K11829）による助成を受けた．

文献

[1] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. deAlmeida Maia, “Dissection of a bug dataset: Anatomy of 395 patches from defects4j,” Proc. International Conference on Software Analysis, Evolution and Reengineering, pp.130–140, 2018.

[2] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, “A learning-to-rank based fault localization approach using likely invariants,” Proc. International Symposium on Software Testing and Analysis, pp.177–188, 2016.

[3] G. Laghari, A. Murgia, and S. Demeyer, “Fine-tuning spectrum based fault localisation with frequent method item sets,” Proc. International Conference on Automated Software Engineering, pp.274–285, 2016.

[4] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M.D. Ernst, D. Pang, and B. Keller, “Evaluating and improving fault localization,” Proc. International Conference on Software Engineering, pp.609–620, 2017.

[5] X.B.D. Le, D. Lo, and C. Le Goues, “History driven program repair,” Proc. International Conference on Software Analysis, Evolution, and Reengineering, pp.213–224, 2016.

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

[7] R. Just, D. Jalali, and M.D. Ernst, “Defects4j: A database of existing faults to enable controlled testing studies for java programs,” Proc. International Symposium on Software Testing and Analysis, pp.437–440, 2014.

[8] C. Le Goues, N. Holtschulte, E.K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, “The manybugs and introclass benchmarks for automated repair of c programs,” Transactions on Software Engineering, vol.41, no.12, pp.1236–1256, 2015.

[9] R. Widyasari, S.Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J.E. Tan, Y. Yieh, B. Goh, F. Thung, H.J. Kang, T. Hoang, D. Lo, and E.L. Ouh, “Bugsinpy: A database of existing bugs in python programs to enable controlled testing and debugging studies,” Proc. Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp.1556–1560, 2020.

[10] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, c. Beszédes, R. Ferenc, and A. Mesbah, “Bugsjs: a benchmark of javascript bugs,” Proc. International Conference on Software Testing, Validation and Verification, pp.90–101, 2019.

[11] S. Benton, A. Ghanbari, and L. Zhang, “Defexts: A curated dataset of reproducible real-world bugs for modern JVM languages,” Proc. International Conference on Software Engineering, pp.47–50, 2019.

[12] J. Lai, C.J. Lortie, R.A. Muenchen, J. Yang, and K. Ma, “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] M. Vidoni, “Software engineering and r programming: A call for research,” Journal on R J., vol.13, no.2, p.600, 2021.

[15] A. Decan, T. Mens, M. Claes, and P. Grosjean, “When GitHub meets CRAN: An analysis of inter-repository package dependency problems,” Proc. International Conference on Software Analysis, Evolution, and Reengineering, pp.493–504, 2016.