特別研究報告

題目

Rubygem におけるバージョンの不一致に起因する問題の調査

指導教員
肥後 芳樹 教授

報告者
橋本 二千翔

令和 6 年 2 月 7 日

大阪大学 基礎工学部 情報科学科

令和 5 年度 特別研究報告

Rubygem におけるバージョンの不一致に起因する問題の調査

橋本 二千翔

内容梗概

ソフトウェアのライブラリとは，再利用可能なコード群のことである．ライブラリを利用して開発効率を向上させる取り組みはソフトウェア開発一般に見られ，ソフトウェアの再利用は，小〜中規模のソフトウェアプロジェクトにおいても慣例となっている．さらにパッケージマネージャーの台頭によりライブラリの共有コストが削減され，プログラムを手早く実装するために多くの開発者がライブラリを利用するようになった．

プログラミング言語 Ruby は，シンプルかつ柔軟な文法を特徴としたオープンソースのオブジェクト指向スクリプト言語であり，Web 開発などに広く使用されている．Ruby のライブラリは Gem と呼ばれ，複数の Gem を用いた Ruby プログラム開発では，Gem の依存関係として，依存 Gem の名前とそのバージョンを宣言した Gemfile が必要となる．この Gem のバージョン制約が不整合を起こした場合，必要な Gem を利用することができない問題が発生してしまう．これを Gem における依存関係問題と呼ぶ．現在，依存関係問題を解決するための様々な手法が提案されているが，Ruby に適したものは見られない．

そこで本研究では，Ruby に適した依存関係解析手法の実現支援を見据え，現段階で発生している問題の実態把握を目指す．実態を明らかにすることで，依存関係解析手法の研究の手助けや，Gem 開発の補助となるデータの提供に繋がることが期待される．目的達成のアプローチとして，GitHub における Gem の開発リポジトリに投稿された Issue を対象とし，依存関係問題の調査を行った．具体的には GitHub の Issue から依存関係問題に関連する可能性があるものを抽出し，その内容を精査し記録した．483 件の Issue を精査した結果，192 件の Issue が実際に依存関係問題に関係しているもので，そのうち 181 件が Bot による自動報告，残り 11 件が人の手による報告であった．それぞれについて，問題の種類・解決状況・競合 Gem の種類の項目を収集したところ，互換性のある Gem のバージョンが見つからないという問題が最も多く，競合 Gem の種類としては bundler が最多であった．最後に結果に対する考察を行い，Ruby の開発者・利用者に有用であると考えられる情報を示した．

主な用語

Ruby
Gem
依存関係
競合

目 次

1　はじめに

Ruby は，シンプルかつ柔軟な文法を特徴としたオープンソースのオブジェクト指向スクリプト言語であり，Web 開発などに広く使用されている．Ruby のサードパーティライブラリ（Gem）は 170,000 件以上[1]存在し，公式リポジトリである RubyGems によってホスティングされている．Ruby ソフトウェアの開発者は，Gem を用いることで開発時間の短縮や開発コストの削減を実現している．

ライブラリを利用して開発効率を向上させる取り組みはソフトウェア開発一般に見られ，例えば「効率的なソフトウェアエンジニアリングと大規模なソフトウェアシステムの構想は，再利用可能なコードに依存している」との主張がある [6]．またソフトウェアの再利用は，小〜中規模のソフトウェアプロジェクトにおいても慣行となっていることが知られている．7 つの主要なソフトウェアパッケージエコシステムを対象とした調査では，そのすべてにおいて大多数のパッケージが他のパッケージに依存していることが判明した [3]．さらにパッケージマネージャーの台頭によりライブラリの共有コストが削減され，プログラムを手早く実装するために多くの開発者がライブラリを利用するようになっている．例えば 1 つの関数のみを含む単純なライブラリであっても，多くのユーザーが利用している事例がある [1]．

複数のライブラリを利用したソフトウェア開発では，ライブラリの依存関係として，依存ライブラリの名前とそのバージョン制約を指定する必要がある．この際，バージョン制約の不整合を原因とした依存関係問題がしばしば発生し，ソフトウェアの開発者と利用者を悩ませている [7]．2024 年現在，依存関係問題を解決するための様々な手法が提案されており，C#開発者による依存関係問題解決プロセスの模倣 [5] や，Python を対象とした独自のヒューリスティックアルゴリズムの提案 [2] はその一例である．しかしこれらは言語依存のアプローチであり，Ruby に適した依存関係解析手法は現状として見られない．

Ruby に適した依存関係解析手法の実現に向け，まず現段階で発生している問題を把握することは重要であると考える．そこで本研究では，Ruby における依存関係問題の実態を明らかにするため，Gem の開発・利用において実際に生じている依存関係問題の調査を行った．調査対象は，GitHub の Issue である．これはソフトウェア開発プラットフォームにおける問題の管理システムであり，多くのユーザが利用している．例えば文献 [5] でも，依存関係問題の調査対象として GitHub の Issue が採用されている．

調査の流れは次の通りである．まず，Gem の GitHub リポジトリにおける Isuue を自動で収集し,「dependen」,「conflict」のキーワードでフィルタリングを行った．その後，フィルタリングされた Issue に対して手動で調査を行い，調査結果を基にした考察を行った．

GitHub ページを有する 87,217 種の Gem から得られた Issue のうち 483 件を精査した結

[1]本研究における調査過程（2023 年 10 月 25 日時点）で得られた数である．調査の詳細は 3 節で述べる．

果，依存関係問題に関連する Issue が 192 件得られた．それらの内容を読み取り考察したところ，以下に示す傾向が見られた．

- Bot による自動報告 181 件
  - 多くが「bundler」と呼ばれる Gem により引き起こされる問題
- 人の手による手動報告 11 件
  - 報告に挙がる Gem は様々
  - 多く見られる実際の問題は「ライブラリのインストール自体ができない」
  - 解決していた Issue は 7 件
  - 解決手法として問題となっていた Gem のバージョンを下げるものが多く見られた

以降，2 節では本研究の背景として，Ruby・パッケージマネージャ・依存関係問題・GitHub の Issue について説明する．続いて 3 節で調査の設計と手順について述べたあと，4 節にその結果を記し，5 節で考察を行う．その後 6 節で妥当性への脅威に関して述べたうえで，最後に 7 節で本論文をまとめ，今後の課題について議論する．

2　背景

本節では，本研究の背景として，プログラミング言語 Ruby，パッケージマネージャ，依存関係問題，および GitHub の Issue について説明する．

2.1　Ruby

Ruby[2]とは，オープンソースのオブジェクト指向スクリプト言語である．その特徴として，柔軟で簡潔な文法やクラスベースのオブジェクト指向機能が挙げられ，開発効率の良いプログラミング言語として，Web 開発をはじめに広く使用されている．

Ruby には Gem と呼ばれる数多くのサードパーティライブラリが存在している．Ruby ソフトウェアの開発者は，Gem の導入により，開発時間の短縮や開発コストの削減を実現している．Gem を用いたソフトウェアの開発・実行の流れは，以下の通りである．

1. ソフトウェア開発者が，利用する Gem の依存関係（ライブラリの名前とそれに対するバージョン制約）を Gemfile に書き込む．
2. ソフトウェア利用者が，Gemfile に基づいて依存関係のある Gem をインストールする．これによりソフトウェア開発者のローカル環境が再現され，ソフトウェアの実行が可能となる．

ただし，Gemfile に記載された依存関係に問題がある場合，

- Gem のインストール時
- Gem のビルド時
- ソフトウェアの実行時

のいずれかにエラーが起こる可能性がある．この問題はたびたび発生し，ソフトウェアの開発者や利用者を悩ませている．

2.2　パッケージマネージャ

パッケージマネージャは，ソフトウェアライブラリやパッケージの管理を行うためのソフトウェアである．特に Ruby のライブラリにおいては，RubyGems[3]というパッケージマネージャが存在する．パッケージマネージャを利用することで，コンピュータ言語におけるサードパーティライブラリの利用を効率化できる．

[2] https://www.ruby-lang.org
[3] https://github.com/rubygems/rubygems

パッケージマネージャは通常，パッケージ管理や依存関係解決といった機能を持つ．以下にその詳細を述べる．

パッケージ管理

パッケージ管理は，パッケージマネージャがユーザに提供する，サードパーティライブラリのインストールやアンインストール，アップグレードに関する機能である．サードパーティライブラリのソースコードや利用可能なバージョン情報，依存関係に関する情報は，インターネット上に公開されたリポジトリ（ホスティングサービス）において公開されている．

パッケージ管理の主な流れは次の通りである．

1. ユーザがパッケージ管理に関する操作を行う．
2. パッケージマネージャが，特定のライブラリに関するソースコードやその他の情報をホスティングサービスへ問い合わせる．
3. 問い合わせで得られたデータを基に，パッケージマネージャがインストールやアンインストール，アップグレードを実行する．

依存関係解決

依存関係解決は，ソフトウェアの実行に必要となるライブラリの特定やバージョンの解決を実現する機能であり，ライブラリのインストール時にパッケージマネージャにより実行される．

依存関係解決の主な流れは次の通りである．

1. ユーザがライブラリのインストール操作を行う．
2. パッケージマネージャがホスティングサービスへ依存関係情報を問い合わせ，以下の情報を取得する．
   - インストール対象のライブラリ（依存元ライブラリ）が依存しているライブラリ（依存先ライブラリ）の名前
   - 依存先ライブラリのバージョンに対する制約（バージョン制約）
3. パッケージマネージャが，バージョン制約を満たすような依存先ライブラリをインストールする．

2.3 依存関係問題

ソフトウェアの依存関係とは，あるソフトウェアと（その実行に必要となる）別のソフトウェアの間に生じる関係である．ソフトウェアの依存関係は，一般的に以下を用いて表現さ

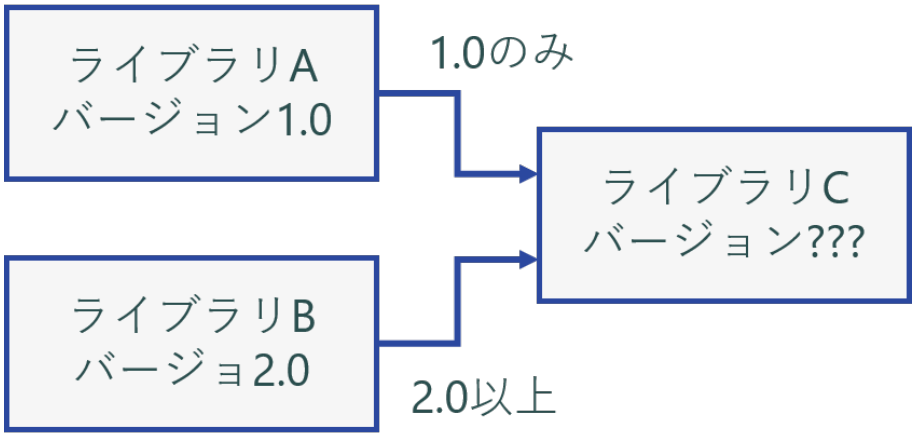

図 1: 依存関係問題の例

れる．

- 自身のソフトウェアの名前とバージョン
- 依存関係のあるソフトウェアのバージョン指定

複数のライブラリを用いたソフトウェアにおいては，依存関係により指定されたバージョンを満たすライブラリをインストールできない場合がある．これを依存関係問題と呼ぶ．

2.3.1 具体例

本節では，ごく簡単な例を用いて依存関係問題を説明する．図 1 はライブラリの依存関係をグラフ状の図で表しており，以降これを依存関係図と呼ぶ．依存関係図は，ソフトウェアを表すノードと依存関係を表すエッジで構成される．

ノード ソフトウェアの名前とバージョンを表す．

エッジ ソフトウェアの依存関係を表す．エッジの向きは依存関係の導入方向に，エッジのラベルはバージョン制約に対応している．

図 1 は，3 つのライブラリ A・B・C における依存関係問題の例である．まずノードを見ると，ライブラリ A のバージョン 1.0 とライブラリ B のバージョン 2.0 がすでにインストールされている．続いてエッジを見ると，以下 2 つの依存関係が読み取れる．

- ライブラリ A のバージョン 1.0 からライブラリ C に対して依存関係があり，バージョン制約として「バージョン 1.0 のみ」が指定されている．

- ライブラリ B のバージョン 2.0 からライブラリ C に対して依存関係があり，バージョン制約として「バージョン 2.0 以上」が指定されている．

この例では,「ライブラリ A から C へのバージョン制約」と「ライブラリ B から C へのバージョン制約」が矛盾しているため，ライブラリ C のインストールができず，依存関係問題が発生している．

2.3.2 既存研究

本節では，ソフトウェアの依存関係問題に関する 2 つの研究事例を紹介する．

Nufix

Nufix[5] は，C#（.NET）のライブラリにおける依存関係問題を取り扱った研究である．.NET には複数のプラットフォームが存在しており，1 つのプロジェクト内であってもプラットフォームごとに別々の依存関係問題が発生する場合があるため，開発者はその対応に頭を悩ませている．この研究では.NET の依存関係問題に対し，開発者の依存関係問題解決プロセスを模倣する手法を開発することにより，解決を試みている．

具体的な手法は次の通りである．まず，実際の依存関係問題事例に対して調査を行い，開発者の依存関係問題解決における共通点を発見する．続いて，その共通点に基づいて解決プロセスを 0-1 線形計画法として定式化を行う．その後，線形計画法のパラメータに対してチューニングを行うことで，より開発者の解決プロセスに近い手法を実現している．

提案手法を既存の依存関係問題に適用した結果，その 73.3%においてテストケースを通過した．

PyCRE

PyCRE[2] は，Python のライブラリにおける依存関係問題を取り扱った研究である．Python のサードパーティライブラリは数多く存在し，それを利用することで開発者は効率的なソフトウェア開発を行うことができる．Python のコードは様々なウェブプラットフォームで公開されているが，依存関係の指定が適切に行われていないため，利用者はランタイム環境の再現に苦労している．

この問題に対して，Python の依存関係を自動推論するツールの提案 [4] が見られるが，これは Python の依存関係を推論するための情報を十分に持っておらず，推論の成功率には限界がある．そこで，この研究では事前の知識収集と独自のヒューリスティックアルゴリズムを用いたツールの実装を行った．

具体的な手法は次の通りである．まず，PyPI でホスティングされている 10,000 以上のライブラリに対する事前調査を行い，名前・バージョン・ライブラリの持つモジュール名を収集する．続いて，収集したデータに基づいたデータベースの構築を行い，それを用いてソースコードが依存するライブラリの推論を行う．最後に，推論されたライブラリの推移的依存関係を独自のヒューリスティックアルゴリズムで解決することで，利用するライブラリのバージョンを特定する．

この手法を，10,250 にのぼる既存の依存関係問題例に適用した結果，既存手法よりも 447 件多くインポートエラーを解決し，また 337 件多く実行可能な解決を達成した．

上記の既存研究はそれぞれ C#および Python に特化した手法であり，そのまま Ruby に適用することはできない．すなわち，Ruby に適した依存関係解析手法が求められている．

2.4 ソフトウェア開発における問題の共有と管理

ライブラリのようなソフトウェアを複数人で開発する際は，発生した問題の共有や管理が必要となる．本節では，ソフトウェア開発のプラットフォームであり Gem の開発にも広く活用されている GitHub と，その機能の 1 つである Issue について説明する．

2.4.1 GitHub

GitHub[4]とは，ソフトウェア開発を支援するプラットフォームである．開発者は「リポジトリ」と呼ばれる単位でソフトウェアの管理を行い，以下の機能を活用しながら複数人で開発を進める．

- バージョン管理機能
  - ソースコードのコミット履歴
- コラボレーション機能
  - Issue（バグ報告や機能追加に関する議論）
  - Pull Request（コード変更の提案とそのレビュー）
- ドキュメンテーション機能
  - README ドキュメント（プロジェクトの目的・概要の説明）
  - Wiki ページ（プロジェクトに関する様々な情報・ノウハウの共有）

[4] https://github.com

上記の機能により

- プロジェクトの進行状況の追跡
- チームメンバー間のコミュニケーションの促進

が可能となり，開発プロセス全体の効率性向上につながっている．GitHub を用いた開発は世界中で行われており，2024 年 2 月 7 日時点で存在するリポジトリは計 3.3 億件以上にのぼる[5]．その中には Gem の開発リポジトリも多く含まれる．

2.4.2 Issue

本研究では，GitHub の提供する機能の中でも Issue に着目する．Issue とは，発生したバグの報告や新規機能の要求などに関する議論を行うチャットスペースであり，開発における課題管理を目的として広く利用されている．以下に，Issue の利用の典型的な流れを示す．

1. ソフトウェアのユーザ（あるいは開発者）が Issue を Open し，問題の報告や機能追加の要求に関する詳細を記述する．
2. Open された Issue に対し，開発者がコメントを投稿する．
3. 必要に応じて議論を続行する．例えば，問題が発生した環境の詳細を尋ねたり，他に試した方法は無いか確認したりする．この過程は第三者も参加可能である．

議論の末に問題が解決した場合や，これ以上の議論の続行が不可能であると判断した場合は，Issue のステータスを明示的に Closed へ変更するケースが多い．ただし，ステータスの変更は Issue を Open したユーザや開発者の判断により実施されるため，問題の解決状況と Issue のステータスは必ずしも一致しないことに注意する．

図 2 は，Gem の 1 つである chef の開発リポジトリ[6]における Issue の実例である．ここでは，以下の流れで議論が進められている．

1. ソフトウェアの開発者が，発生した問題の詳細記述とともに Issue を Open する（図 2 (1) のコメント）．
2. Open された Issue に対し，別の開発者が対応案を投稿する（図 2 (2) のコメント）．
3. 問題が解決したため，Issue を Open した開発者がステータスを Close に変更する（図 2 (3) の操作）．

このように，GitHub の Issue には実際のソフトウェア開発で発生している問題が数多く記載されている．その中には依存関係問題に関する投稿も多く含まれると考えられる．

[5] https://github.co.jp/
[6] https://github.com/chef/chef

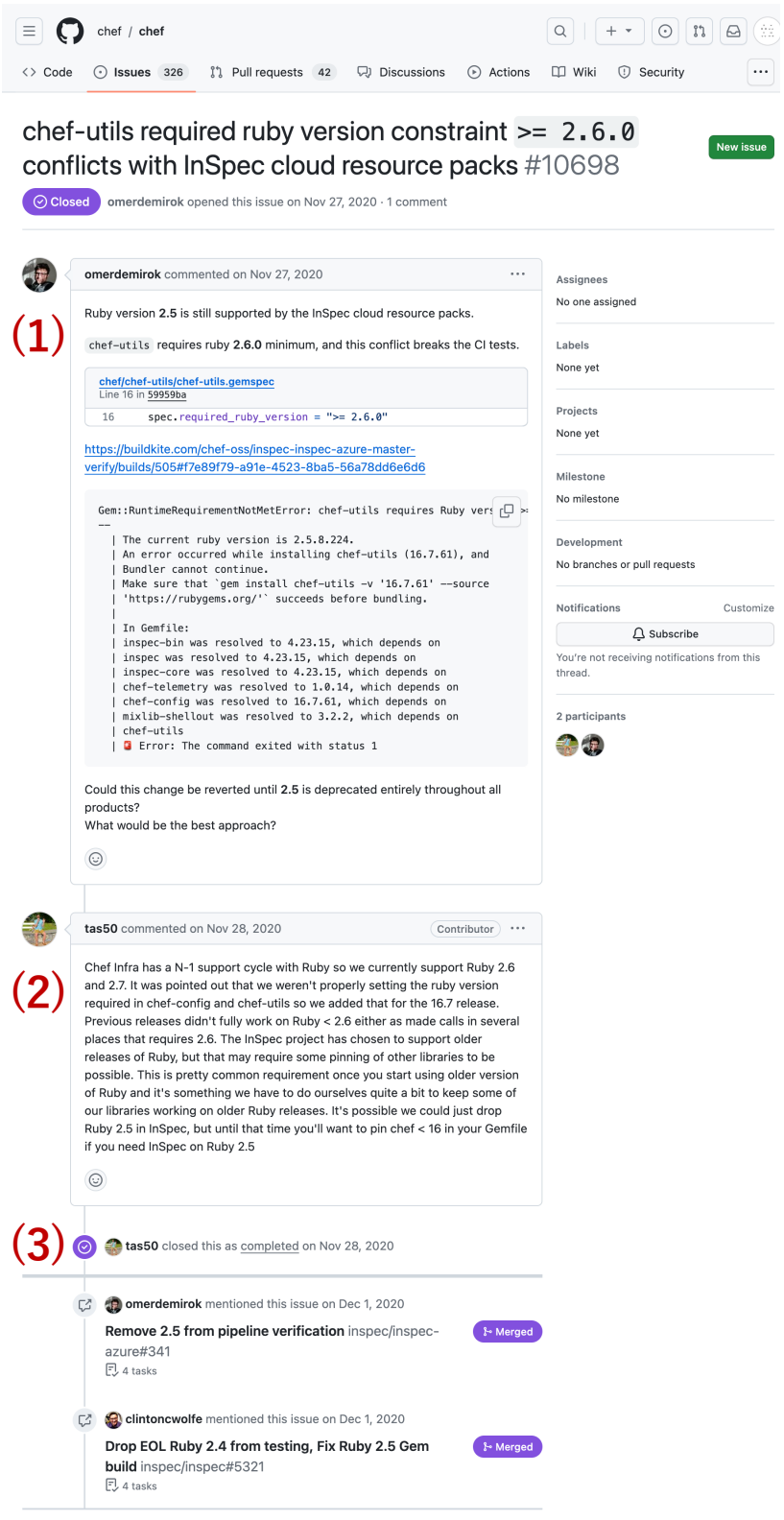

図 2: GitHub における Issue の例

3　調査方法

本研究では，Ruby における依存関係問題の実態を明らかにするため，Gem の開発・利用において実際に生じている依存関係問題の調査を行う．本節では，調査の設計と手順の詳細を述べる．

3.1　設計

3.1.1　調査項目

依存関係問題とそれを取り巻く状況の実態を把握するためには，以下の情報が有用だと考えられる．

- **どのような Gem が競合を起こしているか．**
  特定の Gem，あるいはそのバージョンが頻繁に競合に関わるならば，その情報の重要度は高いと推測できる．
- **競合の結果，どういった実害が発生しうるのか．**
  実害の傾向や競合を起こした Gem との関連性は，有用な情報になると考えられる．
- **実際に発生した問題を解決できているか．**
  解決できている場合は，その解決の過程を調べることで，どういった対処が頻繁になされているのかが分かる．

上記に従い，以下の 3 点を調査項目として採用した．

1. 競合に関係する Gem とそのバージョン
2. 発生した問題の種類
3. 問題の解決状況（解決した場合は解決方法）

3.1.2　調査対象

実際に生じている問題を調査するためには，できるだけ多くの情報を効率的に集める必要がある．そこで本研究では，依存関係問題の調査対象として GitHub の Issue を選択した．2.4 節で紹介したように，GitHub はソフトウェア開発のメジャーなプラットフォームであり，プログラムの実行による情報の自動収集も可能であるため，調査対象として適切であると考えられる．例えば文献 [5] でも，依存関係問題の調査対象として GitHub の Issue が採用されている．

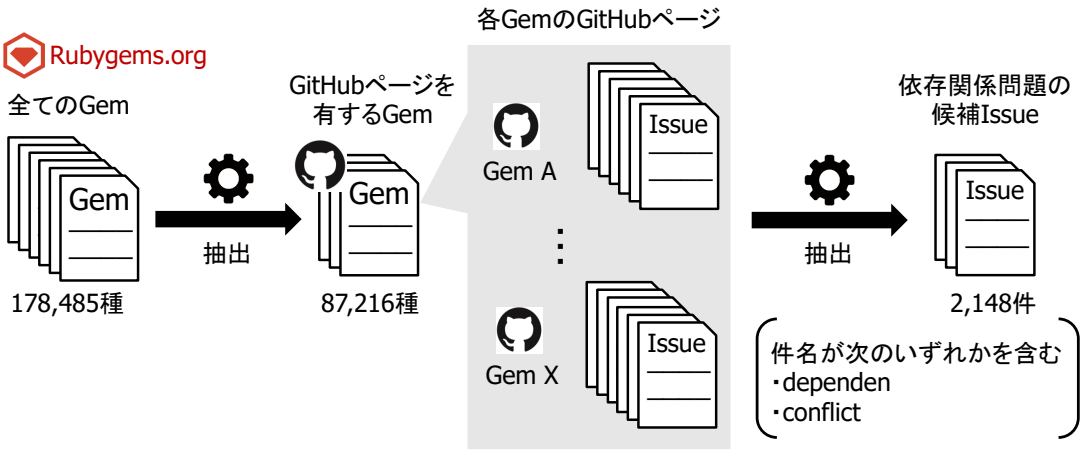

図 3: 【STEP1】候補 Issue の取得（自動）

3.2 手順

調査は大きく，GitHub からの Issue の取得（STEP1）と，その内容の精査（STEP2）に分けられる．STEP1 は自動で，STEP2 は手作業で行う．以降 3.2.1 節および 3.2.2 節で，その詳細を述べる．

3.2.1 STEP1：候補 Issue の取得

STEP1 ではまず，GitHub から Issue を取り出す作業を行う．その概要は図 3 に示す通りである．具体的には，下記の手順でスクレイピング作業を行う Python プログラムを作成し，実行した．

1. RubyGems.org[7]に登録されているすべての Gem を対象に，次の 3 情報を取得して CSV ファイルに記録する．
   - Gem の名前
   - Gem のダウンロード数
   - Gem のホームページの URL
2. 上記 CSV ファイルから，ホームページが GitHub である Gem のみを抽出する．
3. GitHub の URL をもとに Issues ページへアクセスし，件名に「dependen」,「conflict」を含む Issue のみを抽出する．

[7] https://rubygems.org

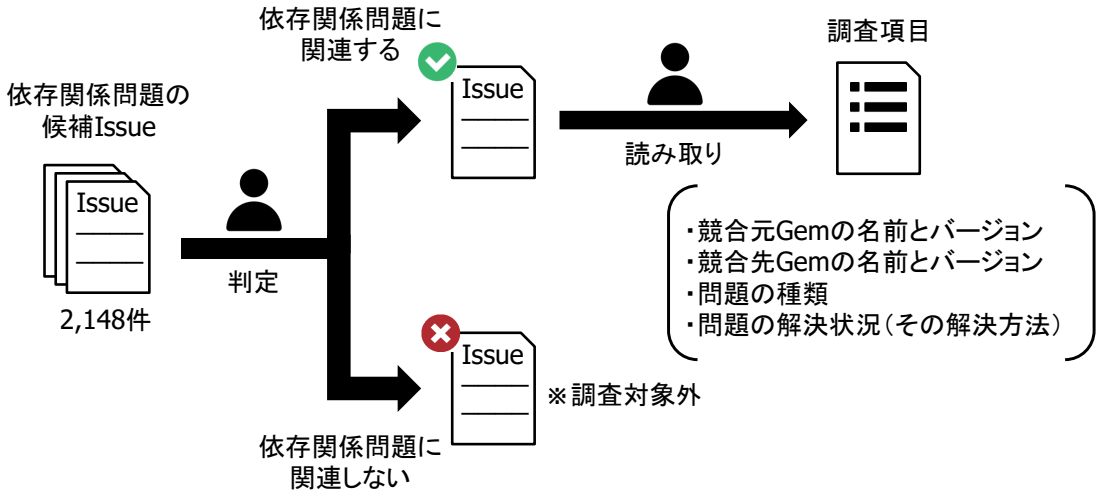

図 4: 【STEP2】候補 Issue の精査（手動）

4. 抽出した Issue を対象に，次の 4 情報を取得して別途 CSV ファイルに記録する．

    - Gem の名前
    - Gem のダウンロード数
    - Issue の件名
    - Issue の URL

なお，手順 3 における抽出キーワードは,「conflict」に加えて「dependen」を用いた理由は,「dependent」,「dependency」のどちらも抽出できるようにするためである．これらのキーワードを含む Issue が，依存関係問題の調査対象として適していると考えた．

また手順 3 においては，ステータスが「Open」である Issue のみを対象としている．これは，筆者の作業効率を考慮したうえで，特に解決が難しい問題に特化して情報を収集するためである．

3.2.2 STEP2：候補 Issue の精査

STEP2 では，STEP1 で得られた候補 Issue を手作業で一件一件精査し，調査対象として適切であると判定した場合にその内容を読み取り記録する．その概要は図 4 に示す通りである．

ある Issue が本研究の調査対象として適切であるかの判定基準を以下に示す．この基準は，筆者自身の作業効率を考慮し，可能な限り情報収集を行いやすくするために設けている．

- 依存関係に関連する Issue

  - エラーログがある
  - 依存関係に関する具体的な問題の記述がある
- 依存関係問題に関連しない Issue
  - Issue ページの文章部分が空である
  - 具体的な Gem の名前が記載されていない
  - 有意な情報が読み取れない

いずれかの基準を満たしている Issue を，適している（あるいは適していない）ものとして判定を行った.

4　調査結果

4.1　Gem および Issue の総数

まず，3.2.1 節の調査手順 STEP1 により得られた Gem および Issue の件数を，表 1 に示す．この結果は全て，2023 年 10 月 25 日時点で得られたものである．最終的に依存関係問題に関連する可能性のある Issue を 2,148 件取り出せた．これは，実態調査のために十分な量の標本であると判断した．

表 1: STEP1 で取得した Gem と Issue の件数

Gem または Issue の種別	件数
RubyGems.org に登録されている Gem	178,486
GitHub ページが確認できた Gem	87,217
各 Gem の GitHub ページから抽出した Issue	2,148

次に，3.2.1 節の調査手順 STEP2 により得られた Issue の件数について述べる．本研究では，2,418 件のうち 483 件を精査対象とし，判定を行った．その結果，実際に依存関係問題に関連する Issue が 192 件得られた．

精査の過程で，GitHub の機能として備わっている Bot（Dependabot）によって自動で報告された Issue が存在することが判明した．こうした Bot による自動報告の内容は，人間側で実際に問題になっているとは限らないため，以降では Bot による自動報告と人間の手による報告を区別する．その内訳は表 2 の通りである．

表 2: 報告方法別に見た Issue の内訳

報告方法	件数
Bot による自動報告	181
人の手による手動報告	11

4.2　問題の種類

4.2.1　Bot による報告

まず，精査対象の大多数を占めた Bot による報告に関して，発生した問題の種類別に見た件数を述べる．全 181 件は表 3 のように分類できた．

表 3 におけるそれぞれの問題の種類について説明する．

- **互換性のある Gem のバージョンがない**

表 3: 問題の種類別に見た Bot による報告の内訳

問題の種類	件数
互換性のある Gem のバージョンがない	162
互換性のある Ruby のバージョンがない	9
それに依存する Gem の互換性のあるバージョンが見つからない	6
当該 Gem が見つからない	4

その Gem が持つ Gemfile に記載されている，ある Gem の適切なバージョンが bundler によって見つけられなかったという問題である．bundler とはパッケージマネージャの役割を果たす Ruby の Gem である．

- **互換性のある Ruby のバージョンがない**

  その Gem が持つ Gemfile に記載されている，ある Gem が要求している Ruby 本体のバージョンと Gemfile に書かれている Ruby 本体のバージョンが食い違うなどで，適切な Ruby のバージョンが存在しないという問題である．

- **それに依存する Gem の互換性のあるバージョンが見つからない**

  これは「互換性のある Gem のバージョンがない」の問題に類似しているが，その Gem の Gemfile に記載されているある Gem について,「互換性のある Gem のバージョンがない」の問題が発生しているものである．

- **当該 Gem が見つからない**

  その Gemfile に記載されている特定の Gem が bundler によって見つけられないという問題である．

4.2.2 人の手による報告

続いて，人の手による報告の Issue 全 11 件について，問題の種類別に見た内訳を表 4 に示す．

表 4 におけるそれぞれの問題の種類について説明する．

- **インストールができない**

  インストールの段階で,

  - ConflictError
  - NoMethodError

表 4: 問題の種類別に見た人の手による報告の内訳

問題の種類	件数
インストールができない	6
アプリケーションの実行エラー	2
互換性のある Gem のバージョンがない	2
ビルドができない	1

  - DependencyError
  - ExtensionBuildError

  といったエラーが発生する問題である．エラーの種類は，各 Issue に記載のエラーログより確認した．

- **アプリケーションの実行エラー**

  Gem のインストール，およびコンパイルは通ったがアプリケーションの実行段階でエラーあるいはバグが発生してしまう問題である．

- **互換性のある Gem のバージョンがない**

  Bot の報告に見られた問題と同様である．

- **ビルドができない**

  Gem のインストールはできても，ビルドを行う段階でエラーが発生してしまうという問題である．

4.3 解決状況

精査できた Issue のうち，解決したと確認できた問題は 7 件であった．その内訳を表 5 に示す．

表 5 におけるそれぞれの問題の種類について説明する．

- **新しいバージョンで修正**

  問題を修正した新しいバージョンをリリースすることで，依存関係問題を解消する解決法である．Issue を立てた側ではなく，開発者自身が対応する形式を取る．

- **自身のバージョンを下げた**

  図 1 でも示されているとおり，自身のバージョンを下げることで，依存関係を維持する解決法である．

表 5: 解決した Issue における解決方法の内訳

解決方法	件数
新しいバージョンで修正	2
自身のバージョンを下げた	1
競合先のバージョンを下げた	1
Gemfile の宣言するバージョンの範囲を広げた	1
不明	2

- **競合先のバージョンを下げた**

  自身のバージョンではなく相手側のバージョンを下げることで，依存関係を維持する解決法である．

- **Gemfile の宣言するバージョンの範囲を広げた**

  Gemfile に宣言された依存関係のバージョン制約の範囲を広げることで，競合していなかったことにする解決法である．

4.4 競合 Gem

競合先の Gem として Issue に挙げられていたすべての Gem を，表 6 に示す．挙げられた Issue の件数と，Bot による報告であるかで区別しグルーピングを行った．

表 6: Issue の件数別に見た競合先の Gem の種類

競合先の Gem	Issue の件数
bundler	160（すべて Bot）
codecov	4（すべて Bot）
rubocop	4（すべて Bot）
berkshelf	2（すべて Bot）
rubustats, Faraday, rspec, reel, reel-rack, celluloid, autotest-fsevent, activesupport, nokogiri, uniform_notifier, carrierwave, aws-sdk-core, chef-provisioning	各 1
pry-byebug, embulk, google_drive, jekyll, rdoc, mutant, windows-pr x86-mingw32, steep, english, rainbow, solidus, activerecord, stealth	各 1（すべて Bot）

5　考察

5.1　問題の種類

本節では，表 3 および表 4 に記載された各問題がなぜ発生したかを考察する．

5.1.1　Bot による報告

まず,「互換性のある Gem のバージョンがない」ケースのほとんどは,「bundler」のバージョンが適切でないというものであった．この bundler はパッケージマネージャであり，自身をアップデートすることができないために問題が発生したと考えられる．bundler については，Bot による報告が非常に多く，すべてが開発者に無視された Issue であったため，情報としての価値は低い可能性がある．また，この「互換性のある Gem のバージョンがない」ケースに分類された Issue のうち，競合先の Gem が bundler 以外のものは 2 件のみであったことから，標本の少なさによりケースが偏っている可能性がある．したがって今回は，本ケースが発生した原因の推定は困難であると判断した．

続いて,「互換性のある Ruby のバージョンがない」ケースは 9 件と，無視できない数を確認できた．このタイプのケースでは rubocop という Gem が 4 件を占めていたため，rubocop は Ruby のバージョンの食い違いが発生しやすい Gem である可能性が考えられる．

また,「それに依存する Gem の互換性のあるバージョンが見つからない」ケースも 6 件あった．こちらのケースでは codecov という Gem が 4 件で登場している．codecov の Gemfile と Issue が投稿された側の Gemfile 間にバージョン制約の宣言の食い違いが発生しやすいということが考えられる．

「当該 Gem が見つからない」ケースについては，情報量が少なく Gem の種類による問題発生の傾向が読み取れないため，本稿での考察は行わない．

5.1.2　人の手による報告

「インストールができない」ケースが最も多かった．依存関係問題が発生した場合には，インストールの段階で Gemfile の照合からエラーが報告される例が多いと推測される．

「アプリケーションの実行エラー」が発生するケースも見られたが，これは依存関係問題とは別に，ソースコード中で名前空間を分けていないための競合（Conflict）によるエラーの Issue を拾った可能性も考えられる．このようなケースは依存関係問題とは本質的には異なるために，本来は除外すべきではないかという議論については，ここでは複雑になるため避けることとする．

5.2 解決状況

本節では，表 5 に示す結果が得られた理由を考察する．

まず，解決できたと確認された Issue の件数は，7 件と非常に少ないものであった．6 節でも後述するとおり，精査できた Issue の数が少ない上，Close された Issue が調査の対象外となってしまったことに起因していると考えられる．

次に，解決方法に関して考察する．解決方法としては「新しいバージョンで修正」が多く見られた．これは，依存関係問題への対応手段として根本的なものといえる．しかし開発者が対応しなくてはならないため，依存関係問題が多発した場合，開発者への負担が大きくなってしまう可能性がある．

他の解決方法としては，関係する Gem の最新バージョンを使うことを諦め，バージョンを下げることで競合を回避するものであった．この場合だと，最新バージョンでは修正されているバグが残っているなどの状況も想定されるため，別の不具合の原因になりうる．だが，その場での依存関係問題解消が可能なことから，解決方法としては十分有効と考えられる．今後，ツールなどで依存関係問題を解消する際に現実的な手法は，この解決法であると考えられる．

5.3 競合 Gem

本節では，表 6 の結果が得られた原因について考える．

まず，bundler が最多となったのは，自己アップデートが仕様上できないという問題から，競合が発生しやすいためであると考えられる．

bundler に次いで Issue 件数の多い codecov はコードカバレッジに関わる機能の Gem であり，広く用いられている．また rubocop はコードスタイルのフォーマッタの Gem であり，こちらも広く用いられている．これらはメジャーな Gem といえ，広く用いられていることから，問題が発生しやすくなったと考えられる．

6　妥当性への脅威

6.1　内的妥当性

今回は，各 Issue の精査に想定外の時間がかかってしまったために，取り出せた Issue の約 20%にあたる 483 件しか精査できなかった．また，精査が簡単に終わる Bot の自動報告を先にまとめて精査したため，人の手による報告を精査できた数が非常に少ない．そのため調査結果に偏りが生じている可能性がある．加えて，今回抽出した Issue はすべて Open のものだったため，そのほとんどは未解決の Issue であった．Close された Issue も抽出すれば，依存関係問題を解決した例がより多く得られる可能性がある．

問題の種類の分類についても，基準が正当でない可能性がある．調査手順の STEP2 には主観が入らざるを得ない形になっていたため，より明確な基準を決めて調査を行えば，客観性の高いデータが得られると考えられる．

6.2　外的妥当性

今回の調査では，Gem の選定を RubyGems.org で行った．登録されている Gem の総数が 178,486 個であったことから，このサイトで Ruby の Gem を調査することは妥当であると考えられる．

また GitHub ページを持つ Gem に限定して調査を行ったが，その条件を満たす Gem の総数は 87,217 個と，RubyGems.org 全体の約 49%に留まった．半分以上の Gem が調査対象から外れているため，より広く依存関係問題の実例を取得できるよう，スクレイピングの範囲を広げることが望ましい．

なお，GitHub の Issue を取り上げたことに関しては，同様の調査が行われていた例があった [5]．そのため，妥当性はある程度保証されていると考えてよい．

7　おわりに

本研究では，Gem の GitHub リポジトリにおける Issue を対象として，Ruby に関する依存関係問題の実態調査を行った．その結果として次の情報が得られた．

問題になりやすい Gem の種類

Bot による自動報告では，bundler が最も多く，次いで codecov と rubocop が見られた．これらはダウンロード数も多いものであった．人の手による報告では特定の Gem が多く見られるといったことはなかった．

実際に起こっている問題の種類

Bot による自動報告では，互換性のある Gem のバージョンがないという問題が多く見られた．人の手による報告では，インストール自体ができないという問題が最も多く見られた．

本研究における今後の課題として，以下が挙げられる．

Close された Issue の調査

本研究では，ステータスが「Close」である Issue を調査の対象外とした．それらの Issue も対象としてスクレイピングと精査を行うことで，依存関係問題の実態をより詳細に解明できる余地がある．

アプリケーション間で発生する依存関係問題の調査

本研究では調査対象をライブラリに絞っていたが，依存関係問題はライブラリだけでなくアプリケーション間でも発生しうる．その発生傾向についての追加調査を行い，結果を本研究と比較することで，新たな知見が得られる可能性がある．

依存関係解析手法の提案

依存関係問題の実態をより詳細に調査し得られたデータを利用して，Ruby を対象とした汎用性の高い依存関係解析手法の提案なども行いたい．

謝辞

肥後 芳樹 教授は，研究活動についてや中間報告会などにおける御助言だけでなく，私が体調を崩したときに多大な配慮を賜りました．研究発表についても鋭い指摘によって発表の穴などを明確にしていただき，発表の質を向上させる手助けとなりました．

松下 誠 准教授は，本研究にあたっての指導教員として，多くの研究方針やするべきタスクの御助言をたくさん賜りました．私の体調や研究の進捗に合わせての軌道修正のために，多大な配慮の元，相談を重ねて研究の質向上に尽力していただきました．

神田 哲也 助教は，本論文の執筆にあたって，論文の添削や相談などに御協力を賜りました．

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 豊永 民哉 氏は，本研究プロジェクトのメンバーとして，松下 誠 准教授と共に多くの相談や御助言を賜りました．また本論文の執筆にあたって関連論文の提案や多大な御協力を賜りました．また，研究発表についても多くの御協力を賜りました．

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 田畑 彰洋 氏は，私の体調についても御心配いただき，相談にも乗っていただきました．特に論文執筆と発表準備にあたって多大な御協力をいただきました．それだけでなく，研究や論文執筆にあたって今後の糧となる助言も多く賜りました．

最後に，その他様々な御指導，御助言等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深く感謝いたします．

参考文献

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 385–395, 2017.

[2] Wei Cheng, Xiangrong Zhu, and Wei Hu. Conflict-aware inference of python compatible runtime environments with domain knowledge graph. In *Proceedings of the 44th International Conference on Software Engineering*, pp. 451–461, 2022.

[3] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, Vol. 24, pp. 381–416, 2019.

[4] Eric Horton and Chris Parnin. V2: fast detection of configuration drift in python. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, pp. 477–488, 2020.

[5] Zhenming Li, Ying Wang, Zeqi Lin, Shing-Chi Cheung, and Jian-Guang Lou. Nufix: escape from nuget dependency maze. In *Proceedings of the 44th International Conference on Software Engineering*, pp. 1545–1557, 2022.

[6] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.

[7] Hongjie Ye, Wei Chen, Wensheng Dou, Guoquan Wu, and Jun Wei. Knowledge-based environment dependency inference for python programs. In *Proceedings of the 44th International Conference on Software Engineering*, pp. 1245–1256, 2022.