修士学位論文

題目

Redmine プラグイン依存関係問題を解決する手法とシステムの試作

指導教員
松下 誠 准教授 肥後 芳樹 教授

報告者
豊永民哉

令和 6 年 2 月 7 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 ソフトウェア工学講座

令和 5 年度 修士学位論文

Redmine プラグイン依存関係問題を解決する手法とシステムの試作

豊永民哉

内容梗概

Redmine は企業のソフトウェア開発などで広く用いられているプロジェクト管理用オープンソースソフトウェアである．Redmine には標準でタスク管理機能や文書共有機能などが存在し，それを利用することで効果的なプロジェクト管理を行うことができる．さらに，1000 を超えるサードパーティプラグインが存在し，それをインストールすることによって機能の追加やテーマの変更外部サービスとの連携が可能となる．しかし，プラグインは Ruby ライブラリに依存しており．プラグインが複数インストールされることなどによる，プラグイン同士の依存関係による問題が発生し，プラグイン利用を阻んでいる．既存の研究では C#や Python のライブラリに対する依存関係問題の解決手法が提案されている．しかし，既存の手法はその言語に依存した手法であり，Redmine の依存関係や Ruby の依存関係問題に適した手法が求められている．そこで，本研究では Redmine，Ruby の依存関係問題を解決する依存関係解析手法を開発した．

本手法は前処理と本処理から構成される手法を提案する．前処理では，あらかじめプラグインとライブラリの互換性に関して，ソースコードベースの解析を行い，依存関係にあるソフトウェアの各バージョンについて総当たりで評価値を算出しておく．本処理では，前処理で算出された評価値の平均を新たな評価値として設定し，その平均が最大となるバージョン組み合わせを算出する．この時，解の候補となるバージョン組み合わせは膨大であり，単純なアルゴリズムでは求めることが難しい．そのため，整数非線形計画問題と数理計画ソルバーを用いて，互換性に関する評価値の平均が最大となるバージョン組み合わせを求める．また，手法に基づいた依存関係問題解決システムの試作行った．さらに実際の依存関係問題事例に対して手法を適応した．

主な用語

Ruby
Redmine
依存関係問題
互換性問題

目 次

1 まえがき **3**

2 背景 **5**
2.1 Redmine 5
2.2 Redmine における依存関係 5

3 関連研究 **7**
3.1 依存関係 7
3.2 互換性 9

4 提案手法 **12**
4.1 概要 12
4.2 前処理：互換性の評価 12
4.3 本処理：ソルバーによるバージョン探索 15

5 ツールの試作 **18**
5.1 概要 18
5.2 利用するリポジトリ 19
5.3 ライブラリ情報収集モジュール 19
5.4 プラグイン互換性評価モジュール 19
5.5 ライブラリ互換性評価モジュール 21
5.6 互換性情報データベース 23
5.7 ソフトウェアバージョン探索モジュール 29

6 手法の適応例 **30**
6.1 データセット 30
6.2 手順 32
6.3 結果 32
6.4 考察 33

7 まとめ **34**

謝辞 **36**

参考文献 **37**

1　まえがき

ソフトウェアの依存関係 [24] とは特定のソフトウェアを実行する際に必要となる別のソフトウェアとの関係のことである．ソフトウェアの依存関係は一般に依存するソフトウェアの名前とバージョンを用いて静的に指定される．開発者がソフトウェアの依存関係についてメタデータなどで指定することで，利用者は開発者のソフトウェア構成を再現することができる．しかしながら，ソフトウェア依存関係の記述に関して，誤りやメンテンスの不足などがあった場合には依存関係問題が発生することがある [8]．依存関係問題の発生により，ソフトウェアのビルドエラーや実行エラー，想定外の動作が発生するため，ソフトウェアを利用できないことがある [36]．本研究では，ソフトウェアの依存関係問題，その中でも，Redmine [18] における依存関係問題を扱う．Redmine は Ruby [41] で作成されたプロジェクト管理用のオープンソースソフトウェアで，様々な企業において，主にソフトウェア開発プロジェクトなどで採用されている．[42] [5] [43] Redmine にはチケット機能を中心として，ガントチャート機能，Wiki 機能などが備えられており，それを利用することで効果的なプロジェクト管理を行うことができる．また，Redmine にはサードパーティ製のプラグインが多数存在し，それを利用することにより，機能の追加やテーマの変更，外部サービスとの連携が可能となる．しかし，複数のプラグインをインストールした際などに依存関係問題が発生することがあり，プラグイン依存関係問題においても，ビルドエラーや実行エラー，想定外の動作がしばしば発生し，ユーザーを苦しめている．実際，Redmine とそのプラグインが依存するライブラリへの複数のバージョン制約が矛盾を起こすことによって，依存関係問題が発生していた．この問題が実際にユーザによって報告されると，プラグイン開発者はプラグインからライブラリへのバージョン制約のみを変更することで，依存関係問題を解決した．

本研究では，Redmine の依存関係問題に対して整数非線形計画問題を用いた手法とそれに基づいた実装について述べる．あらかじめ，依存関係のあるソフトウェアコンポーネントの互換性について事前に解析を行い，その結果を互換性評価値としてデータベースに保存しておく．この際，プラグインとライブラリの間の互換性については Gemfile を解析することで互換性評価値を算出し，ライブラリ間の互換性についてはソースコードを持ちいて，その API に関する解析を行うことで，互換性評価値を算出する．ここで，Redmine と複数のプラグインがインストールされてプラグイン，ライブラリのバージョンが一つに定まっていたすると，依存関係のあるプラグインとライブラリのペアとライブラリ同士のペアに互換性評価値を割り当てることができる．プラグインとライブラリのバージョン組み合わせごとに互換性評価値の平均を算出することができる．本手法ではこの値が高いほど実行可能性が高いバージョン組み合わせとみなす．ここで，求める解となり得るバージョン組み合わせは，プラグインやライブラリの数の増加とともに急速に増えていくため，単純なアルゴリズムでは求めることが難しい．そこで，互換性評価値の平均を目的とした整数非線形計画問題を作成し，それを数理計画ソルバーで解くことで求めるバージョン組み合わせを得る．また，手法を基に Redmine 依存関係解決システムの試作を行った．さらに，試作したシステムを実際の依存関係問題事例に適応した．

以降，2 節では本研究の背景として，Redmine プラグイン [19] の依存関係問題について紹介する．さらに，3 節では本研究における関連研究についてソフトウェアの依存関係と互換性を扱った事例について，4 節では，提案した手法について，5 節では，実装ツールの詳細についてアルゴリズムを中心に述べる．また，6 節では実装ツールの適応事例について述べる．最後に 7 節では，まとめと今後の課題について述べる．

2 背景

本節では，本研究における背景として Redmine とその依存関係，Redmine プラグインの依存関係問題について説明することで，今回取り扱う課題を明確にする．

2.1 Redmine

Redmine とは，Ruby で作られたプロジェクト管理用のオープンソースソフトウェアである．また，Redmine はウェブアプリケーションフレームワークの Ruby on Rails (以降 Rails と呼ぶ．) [10] を利用して作られている．Redmine は Ruby 本体の開発を含めて様々なプロジェクトで採用されている[1]．

Redmine には中核とされるチケットを利用した課題管理機能をはじめとして，作業の進捗を管理するためのガントチャート機能，文書管理のための Wiki 機能，バージョン管理システムとの連携機能，フォーラム機能などが備えられている．ここで，チケットとは主にソフトウェア開発において用いられる，タスク管理方法の一つで，チケットにはタスクの発生内容や日時，担当者，期日が記載されている．また，ガントチャートとは時系列と共にタスクの進捗状況を棒グラフと共に確認することのできる機能である．Redmine にはサードパーティ製のプラグインが多数存在し，その数は 1000 以上である．ユーザはプラグインを利用することで，Redmine に標準では備えられていない機能の追加，テーマの変更，標準でサポートされない外部サービスとの連携が可能となる．様々な Redmine プラグインが多くのユーザに利用されている．

2.2 Redmine における依存関係

ここでは Redmine におけるソフトウェアの依存関係について紹介する．ソフトウェアの依存関係は依存関係図と呼ばれる図を用いて表される．依存関係図はノード，エッジ，制約の三つの要素で構成される．

ノードはソフトウェアを表し，ノードの中にはソフトウェアの名前とそのバージョンが書かれている．図 1 においては，Redmine のバージョン 4.0(redmine-4.0)，pluginA-1.1，pluginB-2.1，libraryC-1.5，libraryD-2.2 がインストールされていること，また Redmine の依存関係においては Redmine とプラグイン，ライブラリが関わっていることが分かる．

エッジはソフトウェアの依存関係を表す．エッジの出るソフトウェアから向かうソフトウェアに対して依存関係がある．図 1 の redmine-4.0 は pluginA，plguinB，libraryC に対して依存関係があることが分かる．ここで，Redmine の依存関係においては Redmine はプラグインとライブラリに依存しており，プラグインはライブラリに依存していることが分かる．また，ライブラリはライブラリに依存することがある．

制約はソフトウェアの依存関係において，依存元のソフトウェアが依存先のソフトウェアに対して要求するバージョンを表す．依存先のソフトウェアのバージョンが依存元からの制

[1]JAXA スーパーコンピュータ活用課では，チケット管理システムとして Redmine をベースにした CODA`https://www.jss.jaxa.jp/about_coda/` を運用している．

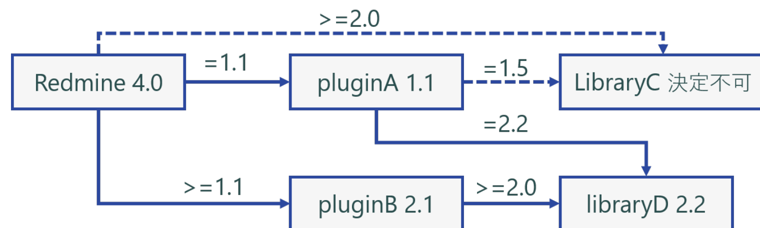

図 1: Redmine における依存関係の例

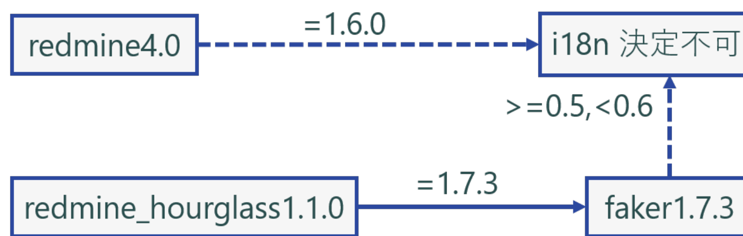

図 2: Redmine プラグイン依存関係問題の発生

約を満たしていない場合は制約の不整合が発生する．制約の不整合が発生している場合は，エッジが破線として表される．

図 1 では redmine-4.0 が libraryC に対してバージョン制約 “>=2.0”，つまりバージョン 2.0 以上を求める．また，liraryA-1.1 は libraryC に対してバージョン制約 “=1.5” つまりバージョン 1.5 のみを求めていることが分かる．このとき，redmine-4.0 と pluginA-1.1 からの libraryC へのバージョン制約は矛盾したものになり依存関係問題が発生する．

ここからは，Redmine の依存関係問題について実際の例 [12]
を用いて紹介する．以下の図 2 は具体的な Redmine プラグイン依存関係問題の例を表したものである．図 2 では redmine-4.1[2] とプラグインである redmine_hourglass-1.1.1 [13] がインストールされており，redmine-4.1 は i18n [33] に対して “=1.6.0” を要求している．対して，redmine_hourglass-1.1.1 は faker [7] に対して “=1.7.3” を要求しており，faker-1.7.3 は i18n に対して “>= 0.5, <0.6” を要求している．この時，Redmine-4.1 から i18n に対する制約と faker-1.7.3 に対する制約は解決不可能となる．実際には，redmine_hourglass の開発者によって，より新しい faker に対応するようにバージョン制約を “>= 2.15.1,<2.6” に変更する修正を行い，それにより対応しているが（図 3）．しかし，利用者からの報告を受けてから実際に変更を行うまでにおよそ 4 か月の時間がかかっている．

[2] `https://www.redmine.org/news/127`

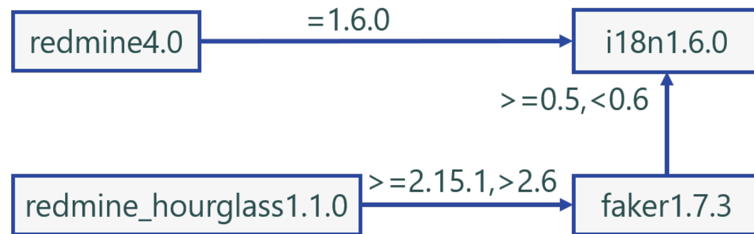

図 3: Redmine プラグイン依存関係問題の解決

3 関連研究

本研究では，Redmine や Ruby の依存関係，互換性を取り扱う．そこで，本節ではソフトウェアの依存関係と互換性を取り扱った関連研究について紹介する．まず，依存関係を取り扱った研究について紹介した後に，互換性を取り扱った関連研究を紹介する．

3.1 依存関係

ここからはソフトウェアの依存関係を取り扱った関連研究５例について紹介する．初めの２例は本研究で扱うような依存関係問題に関する研究として，Nufix と PyCRE における例を紹介する．その後，ソフトウェアにおけるビルド再現性を取り扱った例，依存関係解決における効率性を取り扱った例，依存関係中の脆弱性を取り扱った例について紹介する．

Nufix

Nufix [20] における例は C#エコシステムのライブラリ間の依存関係問題を取り扱った研究である．.NET エコシステムにおいてはプラットフォームが複数あるため，１つのプロジェクトの中で複数の依存関係問題が発生することもあり，開発者の頭を悩ませている．また，ライブラリのバージョン組み合わせ探索空間も非常に大きいため単純なアプローチでは解決が難しい．

この研究では実際の.NET における依存関係問題を調査し，開発者の対処プロセスにおける共通点を発見した．この共通点に基づいて対処プロセスを 0-1 線形計画問題として定式化を行った．また，線形計画法のパラメータを実際の依存関係問題事例に基づいてチューニングすることで，開発者の対処プロセスに近いバージョン組み合わせを出力することを実現している．Nufix を実際の依存関係問題事例 262 件に適用し，その出力を得た．その結果，73.3%においてテストの通過，さらに内 20 件においては実際のプロジェクトにマージされた．

PyCRE

PyCRE [4] における例は Python ライブラリの依存関係問題を取り扱った研究である．Python のオープンソースコードは様々なオンラインプラットフォーム上で公開されている．しかし，ソースコードが利用するライブラリの依存関係が適切に指定されてい

ないため，プログラマはしばしばランタイム環境の再現に苦労する．この問題に対して，Python 依存関係の自動推論ツールが先行研究 [14] において実装された．しかし，このツールは 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，推論の成功率は低くなっている．

そこで，この研究では，事前の知識収集により作成されたグラフ状のデータベースを用いた新しい Python ランライム環境推論ツール PyCRE を実装している．PyCRE は PyPI でホストされている 10000 以上の Python ライブラリに対する事前調査を行い，ライブラリの名前，バージョン，保有しているモジュールの名前を取得する．そのうえで，ソースコードを解析することで依存しているライブラリの名前とバージョンを推論する．その後，ライブラリが依存するライブラリの依存関係を依存関係制約と Python 公式のインストールルールに基づいた独自のヒューリスティックアルゴリズムで解決する．PyCRE を既存の依存関係問題例 10250 例に適応することで，既存手法よりも 447 件多くインポートエラーを解決し，337 件多く実行に成功している．

PyDfix

PyDfix [24] は，Python のビルド再現性を取り扱った研究である．PyPI なのでホストされているライブラリを利用することでコードの再現性が高まる一方で，ソフトウェアのビルド結果は時間共に変化する．特定ライブラリの最新バージョンを常に利用しているアプリケーションにおいてはその影響は顕著である．例えば，依存しているライブラリの一つに破壊的変更が加えられていた場合には，そのソフトウェアのビルドの際にエラーが発生する場合もある．この場合，簡単には過去のビルドを再現できないという問題が発生する．

この研究では，ビルドエラーとなるプロジェクトの直接依存関係指定，推移的依存関係，エラーログを解析することで，ビルドの再現性問題がどのように発生するかを調査している．ここで，直接依存関係，推移的依存関係について説明する．ソフトウェアの依存関係をソフトウェアをノードと依存関係を有効エッジとしたグラフに置き換えて考えると，あるソフトウェアから１つの有効エッジを通じて到達できるソフトウェアを直接依存関係のあるソフトウェア，複数の有効エッジを通じて到達できるソフトウェアを推移的依存関係のあるソフトウェアと呼ぶ．そのうえでビルド再現性修復ツール PyDFix を提案する．PyDfix はビルドのエラーログを解析し，それを基にビルドスクリプトを修正することで Python プロジェクトのビルド再現性を修復した．これを BugSwarm データセット [35] と BugsInPy データセット [38] から取得された 1921 件のビルド再現性のないプロジェクトに適応することで，859 件のプロジェクトに対してビルド再現方法を提供することに成功した．

SmartPIP

SmartPIP [37] は Python おける依存関係解決の時間的，空間的効率を扱った研究である．多くの Python ライブラリは PyPI [9] においてライブラリ管理ツール PIP [31] と共に利用されることで，サードパーティライブラリの再利用性を高め，開発時間の

コストの節約に大きく貢献している．その一方で依存関係の競合が先行する研究で報告され，これに伴い PIP では新たな依存関係解決手法が取り入れられている．また，venv [32] というツールによる仮想環境を用いた手法も様々なプロジェクトで取り入れられている．しかし，この手法ではライブラリを何度もインストールすることによる，時間的，空間的浪費が起こってしまう．

そこで，この研究では事前にライブラリの依存制約を収集し解析を行うことで，より効率的な依存関係解決を行う．また，シンボリックリンクを用いた新たな仮想環境管理手法を用いることで，より空間的効率の高い仮想環境を構築する．この手法を利用することにより，PIP よりも 1.19 倍から 1.60 倍高速なライブラリの依存関係解決と venv よりもストレージ容量を 34.55%から 80.26%削減することに成功した．

DTReme

NDTReme [21] は NPM [25] パッケージの依存関係における脆弱性伝搬について取り扱った研究である．Javascript [17] においてサードパーティライブラリはソフトウェアの開発を促進し，NPM エコシステムの爆発的な成長の一助となった．しかし，外部のライブラリに依存することによる脆弱性の混入は新たなセキュリティ上の脅威となってきている．これに対して，NPM ソフトウェアの依存関係に脆弱なライブラリが利用されているかを推論する手法が提案された．しかし，直接依存関係と推移的依存関係などの考慮しか行われておらず [40]，NPM 固有のインストールプロセスを考慮していないため，誤った脆弱性の推論が起こる可能性がある．

これに対し，この研究では NPM エコシステムのすべてのライブラリに対する脆弱性の調査と，NPM 公式の依存関係解決プロセスを反映した脆弱性推論手法とそれに基づく脆弱性排除ツールを開発した．これを既知の脆弱性を持つプロジェクトに 262 例に適応することで，77 のプロジェクトにおいて公式のツール [26] よりも多くの脆弱性を排除した．

3.2 互換性

ソフトウェアの互換性に関する研究として，Python ライブラリの API に関するものや Java ライブラリの後方互換性に関するもの，JavaScript ライブラリに関するもの，Android アプリケーションの設定ファイルに関するものが行われている．ここではその４例について紹介する．

AexPy

AexPy [6] は Python ライブラリ API の後方互換性を取り扱った研究である．近年の Python の普及に伴い多くの開発者がサードパーティライブラリの開発，保守を行っている．この時，開発者はライブラリを利用するアプリケーションのコードを破壊しないように後方互換性を維持する必要がある．しかし，動的型付け，API エイリアス，デフォルト引数などからライブラリの破壊的変更を検出することは困難である．この研

究に先行する研究 [39] でも，破壊的変更に対する検出手法が提案された，Python の動的な言語特性を考慮していないため，検出は不完全となっている．

そこで，Python の動的特性をライブラリの新たなライブラリ API のモデル化手法を提案する．このモデルを用いてバージョンアップグレード前後の API モデルを比較し，その差分を取得する．取得された差分を 42 種類に分類を行い，その分類結果をもとに API の破壊的変更に関する評価を 4 段階で行う．61 の既知の破壊的変更に対して，この手法を適応したところ，既存手法では 23 の変更を検出できたのに対して，53 の破壊的変更を検出することができた．

DeBBI

DeBBI [3] は Java ライブラリの後方互換性を取り扱った研究である．Java ライブラリは Java のアプリケーションを作成するうえでの労力を軽減し，ソフトウェアの品質を向上させている．その一方で，アプリケーションとライブラリは非同期的にアップグレードされ，そのせいでライブラリの非互換なアップグレードはプリケーションの実行エラーにつながってしまう．

既存のライブラリにおける回帰テスト手法はライブラリの非互換なアップグレードに対して有効であるが，すべての非互換なアップグレードを検出できないでいる．そこで，ライブラリ回帰テストよりも非互換なアップグレードを検知するツール DeBBI を開発する．DeBBI はアプリケーションプロジェクトに付属するテストスイートを利用しそれを非互換性の検出に利用する．さらに，ライブラリのアップグレードされた部分を用いてプロジェクトのテストスイートに対し，検索を行い類似度の高い順にソートすることでより高速な非互換性の検出を行う．実際のサードパーティライブラリに対して，これを適応することで，97 個の非互換性を発見し，またそのうち 19 個は未知の非互換性であった．また，単純にランダムなプロジェクトのテストスイートを用いて，JDK [28] の既知の非互換性を発見するまでにかかった時間と比較して，ライブラリのアップグレードされた部分を用いてプロジェクトのテストスイートに対し，検索を行い類似度の高い順にソートを行ってからテストを行った場合に非互換性を発見するまでにかかった時間は 70.8%短縮された．

CofFix

CofFix [16] はアンドロイドアプリケーションにおける XML 設定ファイルの互換性を扱った研究である．XML 設定ファイルは Android アプリケーションの UI を指定するために利用されるファイルである．このファイルの扱い方は Android フレームワーク [22] のバージョンによって異なるため，XML 設定ファイルの互換性がない場合はソフトウェアのクラッシュや予期せぬ UI の見た目につながる．この問題に対する先行研究 [15] による手法はルールベースであるため，問題の限定的な例のみにしか適応できない．

そのため，この研究では実際の XML 設定ファイルの互換性問題を調査し，その結果を基に互換性問題を自動修正するツール ConfFix を開発した．ConfFix はまず，UI のテ

ストスクリプトを実行し，問題の原因となる XML 要素を特定する．それに基づいて，XML 設定ファイルへのパッチを生成する．このプロセスの繰り返すことで ConfFix は XML 設定ファイルへの適切なパッチの設定を行う．ConfFix を既知の互換性問題 64 例にて適応するとすべての問題例を修正し，ベースラインを上回った．

ConflictJs

ConflictJs [30] は JavaScript ライブラリの互換性を取り扱った研究である．Web アプリケーションでは様々な JavaScript を用いた開発が行われている．これらのライブラリはすべて同じグローバル空間を共有しているため，あるライブラリが他のライブラリの API を誤って変更してしまうことがある．この問題の回避はライブラリが増え続ける中で事実上不可能である．JavaScript ライブラリの解析を行った先行研究 [1] は存在するが，これは複数ライブラリの競合を扱ったものではない．

そこで，この研究では静的解析を用いて，グローバル空間への書き込みを特定し，その書き込みを各ライブラリ間で照合することで潜在的なライブラリ間の競合を検知する．また，テスト生成を用いた動的解析により潜在的な競合が実際に競合しているかを判定する．この手法を 950 のライブラリペアに適応することで，270 の潜在的に競合するライブラリペアと 170 の実際に競合するライブラリペアを特定した．

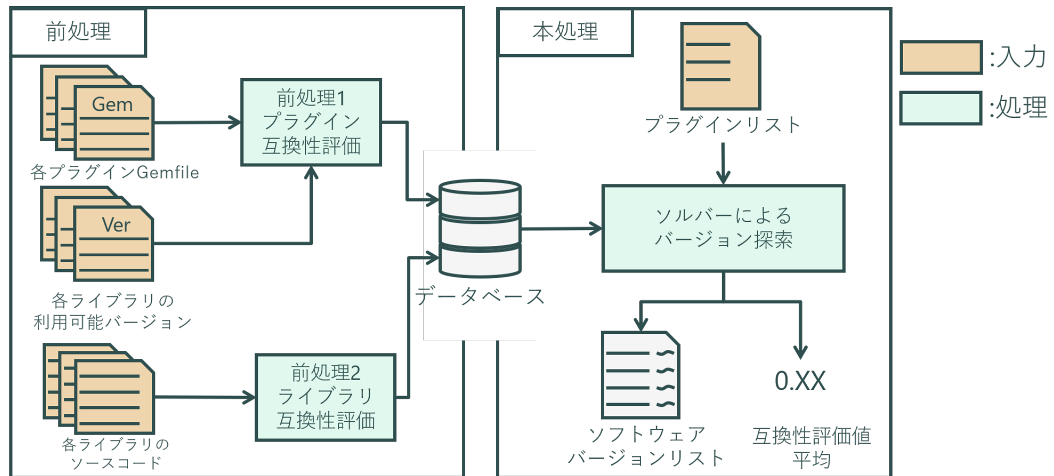

図 4: 手法の概要

4 提案手法

3 説で紹介した，Nufix と PyCRE は依存関係問題を解決する手法が提案されていた．しかし，C# と Python に依存した手法であることがわかる．そこで本節では，2 節で紹介した Redmine プラグインにおける依存関係問題を解決するための新たな手法について紹介する．

4.1 概要

以下の図 4 は本手法の全体像である．本手法は前処理と本処理で構成される．前処理では事前に依存関係にあるプラグイン-ライブラリ，ライブラリ-ライブラリの各バージョンにおける互換性に関する評価を行いスコアを算出し，データベースに保存する．このスコアを互換性評価値と呼ぶ．

本処理では入力されたプラグインリストに基づいて依存するライブラリとその互換性評価値をデータベースから取得する．取得されたスコアを基により実行可能性の高いと考えられるバージョン組み合わせを探索する．バージョン組み合わせの実行可能性は互換性評価値の平均値に基づいて評価される．これを互換性評価値平均と呼ぶ．ここで，解となり得るバージョン組み合わせはライブラリの増加とともに急速に増加する．そのため，互換性評価値平均を目的とした整数非線形計画問題を作成し，それを入力することで求めるバージョン組み合わせを取得する．

4.2 前処理：互換性の評価

前処理では依存関係にあるプラグイン-ライブラリまたはライブラリ-ライブラリの互換性について各バージョンにおける評価を行い，スコアの算出を行う．このスコアを以降では互換性評価値と呼ぶ．また，プラグイン-ライブラリ間の互換性評価値をプラグイン-ライブラ

リ互換性評価値，ライブラリ-ライブラリ間の互換性評価値をライブラリ-ライブラリ互換性評価値と呼ぶ．

ここからは，プラグイン互換性評価とライブラリ互換性評価について紹介する．

前処理 1：プラグイン互換性評価

プラグイン互換性評価とは，依存関係のあるプラグインとライブラリの互換性に関する評価を行うことである．前処理 1 では，プラグイン-ライブラリ間の互換性評価値を依存関係にあるプラグイン-ライブラリの各バージョンについて，プラグインの Gemfile とライブラリの利用可能なバージョン一覧を解析することでプラグイン-ライブラリ互換性評価値の算出を行う．Gemfile とは，Ruby のソフトウェアの依存関係について記述されたファイルで，依存するライブラリの名前とバージョン制約が書かれている．プラグインの特定バージョンに対して互換性のあるライブラリのバージョンとの互換性評価値を 1，互換性のないバージョンとの互換性評価値を 0 とする．

Gemfile の解析について紹介する．図 5 は redmine_hourglass-1.1.0 の Gemfile から coffee-script との依存関係について記述した行を抜き出したものである．ここで “gem coffee-script” はライブラリである coffee-script に依存していることを示し，“ ~>2.4.1” は redmine_hourglass-1.1.0 が coffee-script に対して 2.4.1 以上，2.4.2 未満のバージョンというバージョン制約を導入していることを示している．このとき，coffee-script の利用可能なバージョン一覧が “...,2.2.0,2.3.0,2.4.0,2.4.1” として与えられたとすると，redmine_hourglass-1.1.0 と coffee-script-2.4.1 の互換性評価値は 1，redmine_hourglass と coffee-script-2.4.1 以外との互換性評価値は 0 となる．

前処理 2：ライブラリ互換性評価

ライブラリ互換性評価とは，依存関係にあるライブラリ同士の互換性に関する評価を行うことである．前処理 2 では，ライブラリ-ライブラリの互換性評価値を依存関係にあるライブラリペアの各バージョンについてソースコードへの解析を行うことで算出する．ソースコード解析では依存関係にあるライブラリペアの各バージョンにおける互換性を 0 から 1 の間を取る小数で評価する．スコアを小数で評価することで本来インストール不可能となるソフトウェアの構成であっても可能なかぎり実行可能性の高いバージョン構成を探索することができる．ソースコード解析は提供 API 抽出，呼び出し API 抽出，API 照合で構成される．ここからは架空のライブラリ E のバージョン 1.1 とライブラリ F のバージョン 1.1 を用いた簡単な例を紹介する．

提供 API 抽出

提供 API 抽出では依存先ライブラリが保持している API の完全修飾名をソースコードの AST を解析することで取得する．

```
source 'https://rubygems.org'
gem 'uglifier'
gem 'coffee-script', '~> 2.4.1'
gem 'sass', '~> 3.5.1'
gem 'pundit', '~> 1.1.0'
gem 'therubyracer', :platform => :r
gem 'slim', '~> 3.0.8'
gem 'js-routes', '~> 1.3'
gem 'momentjs-rails', '>= 2.10.7'

gem 'rswag', '<2.0'
gem 'rspec-core'
gem 'rqrcode', '~> 0.10.1'
group :development, :test do
  gem 'rspec-rails', '~> 3.5', '>= 3.5.2'
  gem 'factory_bot_rails', '< 5.0'
  gem 'zonebie'
  gem 'timecop'
  gem 'faker', '1.7.3'
  gem 'database_cleaner'
end
if RUBY_VERSION
```

```
gem 'coffee-script', '~> 2.4.1'
```

図 5: redmine_hourglass-1.1.0 の Gemfile

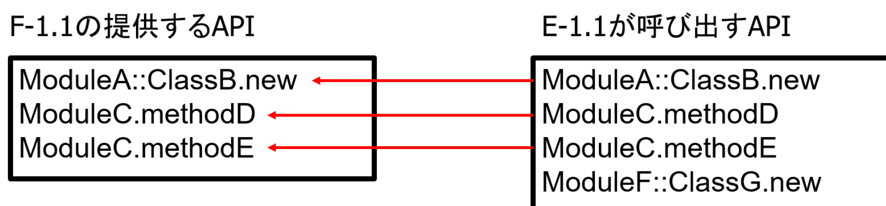

図 6: API 照合

呼び出し API 抽出

呼び出し API 抽出では依存元ライブラリが呼び出す依存先ライブラリの API の完全修飾名をソースコードの AST を解析することで取得する．

API 照合

API 照合では抽出された依存元の API 呼び出しを依存先ライブラリの API をどれだけ満たしているかについて解析を行う．例えば，ライブラリ E-1.1 がライブラリ F-1.1 を呼び出す場合を考える（図 6）．E-1.1 は F-1.1 に対して，“ModuleA::ClassB.new，ModuleC.MethodD，ModuleC.MethodE，ModuleF.ClassG.new” を呼び出したい一方で，F-1.1 が “ModuleA::ClassB.new，ModuleC.MethodD，ModuleC.MethodE” だけを API として持っていたとすると，4 つの API 呼び出しのうち 3 つを F-1.1 が満たしているので，E-1.1 と F-1.1 の互換性評価値は 0.75 となる．このような解析をライブラリ E とライブラリ F の各バージョンについて行うことで，ライブラリ E とライブラリ F 間の互換性評価を行う．

表 1: プラグイン A-ライブラリ B の各バージョンペアに与えられたスコア

A \ B	1.0	1.1	2.0	2.1
1.0	1	0	0	0
1.1	0	1	0	0
2.0	0	0	1	0
2.1	0	0	0	1

4.3 本処理：ソルバーによるバージョン探索

本処理ではまず，利用したいプラグインを入力として与え，データベースから依存するライブラリと互換性評価値を取得する．その後，互換性評価値平均の最大化を目的とする整数非線形計画問題を作成し，それをソルバーに入力する．ソルバーからの出力をソフトウェアバージョンリストと互換性評価値平均に変換することで，最終的な出力を得る．ここからは互換性評価値平均の求め方と整数非線形計画問題，互換性評価値平均の最大化を目的とする整数非線形計画問題の作成，ソルバーによる探索と出力変換について紹介する．

互換性評価値平均の求め方

ここではスコアが与えられた場合の互換性評価値平均算出手順を，ごく簡単な例を用いて紹介する．例えば，プラグイン A がインストールされており，プラグイン A が依存するライブラリ B とライブラリ C がインストールとする．さらに，プラグイン A ーライブラリ B ペアの各バージョンにおけるスコアとプラグイン A ーライブラリ C ペアの各バージョンにおけるスコアが以下の表 1，表 2 ように与えられたとする．ここで，表の 1 行目は依存先ライブラリのバージョン，1 列目は依存元ライブラリのバージョンを表す．

このとき，プラグイン A-1.0，ライブラリ B-1.0，ライブラリ C-1.0 がインストールされていたとすると，互換性評価値平均は 0.88 となる．また，プラグイン A-1.1，ライブラリ B-1.1，ライブラリ C-1.0 がインストールされていたとすると，互換性評価値平均は 0.75 となる．このように，算出される互換性評価値平均を最大化するバージョン組み合わせを見つけることで可能な限り実行可能性の高いと思われるバージョン組み合わせを探索することができる．バージョン組み合わせ探索における解の候補は依存するライブラリの増加とともに急速に増加し単純なアルゴリズムでは解を見つけることが難しい．そのため，依存関係問題を整数非線形計画問題へ変換しソルバーへ入力することで解空間の探索を効率的に行う．

整数非線形計画問題

ここでは，本手法で利用する整数非線形計画問題 [2] について述べる．非線形計画問題とは目的関数もしくは制約条件の中に線形ではない式が含まれているような最適化問題のことであり，資源配分問題や交通流割当問題などの最適化などで用いられる．その中でも，変数

表 2: プラグイン A-ライブラリ B の各バージョンペアに与えられたスコア

A \ C	1.0	1.1	2.0	2.1
1.0	0.75	0.50	0.33	0.10
1.1	0.75	0.80	0.70	0.50
2.0	0.50	0.75	0.85	0.75
2.1	0.50	0.55	0.80	1.00

表 3: 取得された互換性評価値例

B \ A	1.0	1.1
1.0	0.66	0.50
1.1	1.00	0.75

が整数に限られるものを整数非線形計画問題という．計画問題には変数，目的関数，制約式が含まれる．以下に簡単な例を紹介する．

変数が式 1，目的関数が式 2，制約式が式 3 のように定まったとする．この問題を最大化するようにソルバーで最適化を行うと，目的関数の値は 60，x_1 の値は 5，x_2 の値は 2 となる．

$$x_1,\ x_2 \tag{1}$$

$$3x_1^2 + 4x_2^2 + 5 \tag{2}$$

$$-5 < x_1 < 5, -2 < x_2 < 2 \tag{3}$$

整数非線形計画問題の作成　ここでは依存関係問題を互換性評価値平均の最大化を目的とした整数非線形計画問題へ変換する方法について，ごく簡単な例を用いて紹介する．例えばプラグイン A がインストールされており，プラグイン A が依存するライブラリとしてライブラリ B が存在したとする．また，プラグイン A とライブラリ B の互換性評価値が表 3 のように定義されていたとする．このとき，互換性評価値平均の最大化を目的関数としてとる整数非線形計画問題は以下の手順で作成される．

変数の生成: 変数は各プラグインや各ライブラリの各バージョンに割り当てられる．表 3 のような互換性評価値が与えられた場合，生成される変数は以下の式 4 のようになる．ここで，$V_{A-1.0}$ はプラグイン A のバージョン 1.0 が選択されるかどうかを表す．

$$V_{A-1.0},\ V_{A-1.1},\ V_{B-1.0},\ V_{B-1.1} \tag{4}$$

定数の生成: 定数は各プラグインや各ライブラリの互換性評価値に基づいて生成される．生成される定数が以下の式 5 のようになる．ここで $C_{A-1.1B-1.0}$ はプラグイン A-1.1 のライ

ブラリ B-1.0 の互換性評価値を示す.

$$
\begin{aligned}
C_{A-1.0B-1.0} &= 0.66 \\
C_{A-1.1B-1.0} &= 1.00 \\
C_{A-1.0B-1.1} &= 0.50 \\
C_{A-1.1B-1.1} &= 0.75
\end{aligned} \tag{5}
$$

目的関数の生成: 目的関数は生成された変数と定数を基に互換性評価値平均を算出する.

$$
\begin{aligned}
obj =& C_{A-1.0B-1.0} * V_{A-1.0} * V_{B-1.0} + \\
& C_{A-1.1B-1.0} * V_{A-1.1} * V_{B-1.0} + \\
& C_{A-1.0B-1.1} * V_{A-1.0} * V_{B-1.1} + \\
& C_{A-1.1B-1.1} * V_{A-1.1} * V_{B-1.1}
\end{aligned} \tag{6}
$$

制約の生成: 制約は，変数を基に生成され，各ソフトウェアのバージョンのうち 1 つのみが選択可能となるように制限する役割を持つ．ここで，$V = 0\ or\ 1$ は変数が非選択，選択で表されること，$V_{A-1.0} + V_{A-1.1} = 1$ はプラグイン A のうち 1 つのみが選択可能であることを示す.

$$
\begin{aligned}
& V = 0\ or\ 1 \\
& V_{A-1.0} + V_{A-1.1} = 1,\ V_{B-1.0} + V_{B-1.1} = 1
\end{aligned} \tag{7}
$$

ソルバーによる探索と出力変換

整数非線形計画問題を数理計画ソルバーに入力し，その出力結果を得る．整数非線形計画問題の作成で紹介した問題をソルバーへ入力した場合は出力として，目的関数 $obj = 1.0$, 変数 $V_{A-1.0} = 1, V_{A-1.1} = 0, V_{B-1.0} = 0, V_{B-1.1} = 1$ が得られる．これをソフトウェアバージョンリストと互換性評価値平均に変換すると，リストの内容は “A-1.0,B-1.1” になる．また，依存関係の数は 1 であるため，互換性評価値平均は 1.0 となる．これを本手法の出力とする.

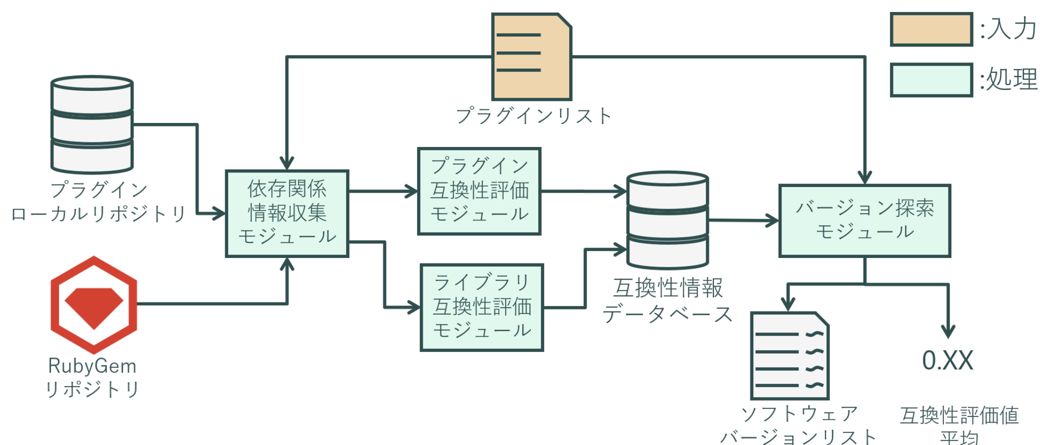

図 7: 試作システムの概要

5　ツールの試作

本節では 4 節で述べた手法を基にして試作した依存関係問題解決システムについて述べる．はじめにシステムの全体像について紹介したのち，その詳細について紹介する．

5.1　概要

以下の図 7 は試作した依存関係システムの全体像である．情報収集モジュールはプラグインリストを入力として受け取り，それに基づいて依存関係情報，ソースコードとプラグインの Gemfile の収集を行う．

その後，取得したプラグイン Gemfile とライブラリの利用可能なバージョン情報をプラグイン互換性評価モジュールへ，依存関係とライブラリのソースコードをライブラリ互換性評価モジュールへ提供する．提供された Gemfile，依存関係情報，ソースコードを基に互換性評価を行い，評価値をデータベースへ保存する．バージョン探索モジュールはプラグインリストを受け取り，それを基にデータベースに問い合わせを行い，ソフトウェア依存関係と互換性評価値の取得を行う．取得した依存関係と互換性評価値を用いて依存関係問題を整数非線形計画問題に変換し，それを解くことで互換性評価値平均が最大となるバージョン構成を得ることができる．

なお，情報収集モジュール，プラグイン互換性評価モジュール，ライブラリ互換性評価モジュールは Ruby を用いて実装し，バージョン探索モジュールは Python を用いて実装し，互換性情報データベースは MySQL [27] を利用して実装した．また，ソルバーとしては Grobi Optimizer [11] を利用する．

以降，利用するリポジトリとシステムの各モジュール，互換性情報データベースについて説明する．

5.2 利用するリポジトリ

RubyGems リポジトリ [34] は Ruby の公式ライブラリリポジトリである．API を利用して依存関係情報の取得，バージョン情報の取得，ソースコードのダウンロードが可能である．本システムではこのリポジトリに対して API を通じて Ruby ライブラリに関する依存関係情報，バージョン情報，ソースコードの取得を行う．

Redmine プラグインリポジトリ [19] は，Redmine が提供している公式のリポジトリであるが，これは API を利用しての情報取得，ソースコードのダウンロードが不可能である．そのため，本実装では Redmine プラグインリポジトリからクローリングによりソースコードを取得して，ローカルプラグインリポジトリを形成した．この際，取得するプラグインは Redmine のバージョン 3.0 以上に対応しており，なおかつ Redmine プラグインリポジトリにおいて１つ以上の評価を得ているものとした．その結果，対象とするプラグインは合計で 364 となった．

5.3 ライブラリ情報収集モジュール

情報取集モジュールではプラグインの名前を基に依存するライブラリの情報を収集する．ここからはその手順を紹介する．プラグインのリスト（$input()$）を入力として受け取り，利用するプラグインの名前（$plugin_name_list$）を特定する．特定したプラグインの名前を基にローカルプラグインリポジトリに対して各バージョンの Gemfile の取得（$getGemfiles$）を行う．取得された Gemfile から，依存しているライブラリの特定を行い（$getDependLibFromGemfile()$），リスト（$library_list$）に格納をする．そのリストを基に RubyGem リポジトリの API に対して問い合わせを行い，利用可能なバージョン一覧を取得する．取得された利用可能なバージョン一覧を基にそのバージョンが依存するライブラリの特定を行う．依存するライブラリに関して，利用可能なバージョンに関する問い合わせ（$getAbailableVersions()$）と依存するライブラリに関する問い合わせ（$getDependLib()$）を再帰的に行う．それにより，プラグインが依存するすべてライブラリの全バージョン ($library_version_list$) を特定する．その後，特定されたすべてのライブラリの全バージョンをダウンロードする．以下にそのアルゴリズムを示す（Algorithm 1）．

5.4 プラグイン互換性評価モジュール

プラグイン互換性評価モジュールでは利用するプラグインの Gemfile と利用可能なバージョンからプラグイン-ライブラリ間の互換性についての評価を行う．その具体的処理について紹介する．まず，各プラグインの全バージョンに関する Gemfile（$plugins_gemfiles$）と各ライブラリの利用可能なバージョン一覧（$library_available_versions$）を入力として受け取る．取得した各 Gemfile からプラグインの名前（$plugin_name$）とバージョン（$plugin_version$），依存するライブラリの名前と制約（$depend_lib_vers_constraints$）の一覧を取得する．取得されたライブラリの名前と制約の一覧から，依存するライブラリ毎に名前（lib_name）と制約（$lib_constraint$）を取得する．さらに，取得されたライブラリの名前を基にライブラリの

Algorithm 1 情報収集

```
plugin_name_list = input()
library_list = list()
library_version_list = list()
dependency_info = dict()
for plugin_name ← plugin_name_list do
    gemfiles = getGemfiles(plugin_name)
    for gemfile ← gemfiles do
        depend_libraries = getDependeLibFromGemfile(gemfile)
        for library ← depend_libraries do
            if library not in library_list then #もし library_list になければ
                library_list ← library
            end if
        end for
    end for
end for
stack = libray_list
while library_list not empty do
    current_library = pop(stack)
    library_version_list ← getAvailableVersions(current_library)
    depend_libraries = getDependLib(currend_library)
    for library ← depend_libraries do
        if library not in library_list then #もし library_list になければ
            library_list ← library
            stack ← library
        end if
    end for
end while
for library_version ← library_version_list do
    downloadLibVersion(library_version)
end for
```

全バージョン（$lib_versions$）を取得する．取得された各バージョン（$lib_version$）について，4.2 節の前処理 1 で説明した手法を用いて，互換性の評価を行い互換性評価値（$score$）を算出（$getScore()$）する．そのスコアを互換性情報データベースに保存する（$storeScore()$）．以下にそのアルゴリズムを示す（Algorithm 2）．

Algorithm 2 プラグイン互換性判定

$plugins_gemfiles = input(0)$
$library_available_verions = input(1)$
for $plugin_gemfiles \leftarrow plugins_gemfiles$ **do**
  for $version_gemfile \leftarrow plugin_gemfiles$ **do**
    $plugin_name = getPluginName(version_gemfile)$
    $plugin_version = getPluginVersio(version_gemfile)$
    $depend_lib_name_constraints = getDependLibInfo(version_gemfile)$
  end for
  for $depend_lib_name_constraint \leftarrow depend_lib_vers_constraints$ **do**
    $lib_name = depend_lib_name_constraint[1]$
    $lib_constraint = depend_lib_name_constraint[2]$
    $lib_versions = getAvailableversions(library_available_verions, name)$
    for $lib_version \leftarrow lib_versions$ **do**
      $score = getScore(lib_version, lib_constraint)$
      $storeScore(plugin_name, plugin_version, lib_name, lib_version, score)$
    end for
  end for
end for

5.5 ライブラリ互換性評価モジュール

ライブラリ互換性評価モジュールでは全ライブラリのソースコードと依存関係情報を与えることでライブラリ間の互換性評価値を各バージョンについて，4.2 節の前処理 2 で紹介した手法で算出し，それを互換性情報データベースへ保存する．具体的には，すべてのライブラリの全バージョンにおけるソースコード（$library_source_codes_list$）と各ライブラリの依存関係情報（$library_dependency_list$）を入力として受け取る．依存関係情報にはライブラリの名前と各バージョン，それぞれのバージョンが依存するライブラリの名前が含まれる．ここで各ライブラリの依存関係情報は以下のリスト 1 のような形式で与えられるとする．ここからは 3 つのステップで説明する．

ステップ 1

まず，各ライブラリの各バージョンのソースコード（$source_code$）を AST に変換し（$getAst()$），それぞれの AST から宣言されたモジュールの名前，宣言されたクラス

の名前，宣言されたメソッドの名前を取得する（$getDeclarations()$）．そのとき，モジュール，クラス，メソッドの名前は以下のようなリスト 2 のような形式で取得される．これを基に，ライブラリの所有するモジュール取得し（$getLibraryModules()$），さらにライブラリ API の各バージョンにおける完全修飾名一覧を得る（$extractedApi()$）．以下にそのアルゴリズムを示す（Algorithm 3）．

ステップ 2

まず，各ライブラリの各バージョンのソースコード
（$source_code$）を AST に変換し（$getAst()$），それぞれの AST からメソッド呼び出し部分（$method_calls$）と変数宣言部分（$variables$）を取得する．変数宣言部とメソッド呼び出し部分のレシーバが一致する場合には，メソッド呼び出しのレシーバを変数の中身に置き換える（$replaceVariable()$）．これを行うことにより，呼び出しているメソッドの完全修飾名（$method_call_fullnames$）を取得する．ここでメソッド呼び出し部分のレシーバについて説明する．"obj.method()"このようなメソッドの呼び出しが行われた場合，Ruby では"obj"の部分をレシーバと呼ぶ．さらに，呼び出しているメソッドの完全修飾名の一覧と依存しているライブラリの持っているモジュール名 ($depend_lib_modules$) を解析する（$exractApiCalls()$）．それにより，依存しているライブラリに対して呼び出す API の完全修飾名（api_calls）を取得する．以下にそのアルゴリズムを示す（Algorithm 4）．

ステップ 3

ライブラリの依存関係情報を基に依存元ライブラリの名前（lib_name）とバージョン（lib_ver），依存しているライブラリのリスト（$depend_libs$）を取得する．依存先ライブラリの名前（$depend_lib$）とバージョン（$depend_lib_ver$）から，依存先ライブラリの各バージョンが提供する API（api）を取得し，依存元ライブラリのバージョンと依存先ライブラリの名前から，依存元ライブラリの各バージョンが呼び出す，依存先ライブラリの API（api_calls）を取得する．依存元の各バージョンが呼び出す API と依存先の各バージョンの API を 4.2 の前処理 2 で示した方法で解析することで，互換性評価値を算出し（$getScore()$），それを互換性情報データベースに保存する（$storeScore()$）．以下にそのアルゴリズムを示す（Algorithm 5）．

Listing 1: ライブラリ依存関係情報

```
{
    "name" = "library1",
    "versions_and_dependencies" = [
        {
            "version" = "1.0",
            "dependencies" = ["library2", … ]
        },
```

```
        {
            "version" = "2.0",
            "dependencies" = ["library2", … ]
        }, …
        ]
}
```

Listing 2: モジュール，クラス，メソッド宣言

```
[
    {
        "type" = "module",
        "name" = "ModuleName1",
        "children = [
            {
                "type" = "class",
                "name" = "ClassName1",
                "children" = [
                    {
                        "type" = "method",
                        "name" = "methodName1"
                    }
                ]
            },
            {
                "type" = "method",
                "name" = "methodName2"
            }
        ]
    }, …

]
```

5.6 互換性情報データベース

互換性情報データベースにはプラグイン互換性評価で求められたプラグイン-ライブラリ互換性評価値と，ライブラリ互換性評価で求められたライブラリ-ライブラリ互換性評価値は互換性情報データベースに保存される．互換性情報データベースは 2 種類のテーブルで構成される．それは，依存関係テーブルと互換性テーブルである．ここからは依存関係テーブ

Algorithm 3 ライブラリ互換性評価：ステップ１

```
library_source_codes_list = input(0)
library_dependency_list = input(1)
extracted_api = list()
library_modules = list()
for library_source_codes ← library_source_codes_list do
    for source_code ← library_source_codes do
        ast = getAst(source_code)
        declarations = getDeclarations(ast)
        library_modules = getLibraryModules(declarations)
        extracted_api = extractedApi(declarations)
    end for
end for
```

Algorithm 4 ライブラリ互換性評価：ステップ２

```
library_source_codes_list = inputFromStep1(0)
library_dependency_list = inputFromStep1(1)
library_modules = inputFromStep1(2)
extracted_api_calls = list()
for library_dependency ← library_dependency_list do
    lib_name = library_dependency["name"]
    for version_dependency ← library_dependency do
        lib_ver = version_dependency["version"]
        depend_libs = getDependencyLib(version_dependency)
        source_code = getLibSourceCode(library_source_codes_list, lib_name, lib_ver)
        ast = getAst(source_code)
        method_calls = getMethodCalls(ast)
        variables = getVariables(ast)
        method_call_fullnames = replaceVariable(method_calls, variables)
        for depend_lib ← depend_libs do
            depend_lib_modules = getModules(library_modules, depend_lib)
            api_calls = exractApiCalls(method_call_fullnames, depend_lib_modules)
            extracted_api_calls ← api_calls
        end for
    end for
end for
```

Algorithm 5 ライブラリ互換性評価：ステップ３

$extracted_api = inputFromStep1(0)$
$library_dependency_list = inputFromStep2(0)$
$extracted_api_calls = inputFromStep2(1)$
for $library_dependency \leftarrow library_dependency_list$ **do**
  $lib_name = library_dependency[''name'']$
  for $version_dependency \leftarrow library_dependency$ **do**
    $lib_ver = version_dependency[''version'']$
    $depend_libs = getDependencyLib(version_dependency)$
    for $depend_lib \leftarrow depend_libs$ **do**
      $depend_lib_vers = getLibVersions(library_dependency_list, depend_lib)$
      for $depend_lib_ver \leftarrow depend_lib_vers$ **do**
        $api_calls = getApiCalls(extracted_api_calls, lib_name, lib_ver, depend_lib_ver)$
        $api = getApi(extracted_api, depend_lib, depend_lib_ver)$
        $score = getScore(api, api_calls)$
        $storeScore(score, lib_name, lib_ver, depend_lib_name, depend_lib_ver)$
      end for
    end for
  end for
end for

ル，互換性テーブルの順で紹介する．

依存関係テーブル

依存関係テーブルには，プラグインとライブラリの依存関係とライブラリ同士の依存関係が保存される．具体的には，以下の表４のような形式で保存される．ここで，依存関係テーブルは dependency_table という名前を持つ．各レコードにはプライマリーキーである id が割り当てられる．dependant_type は依存元ソフトウェアのタイプ（ここではプラグインかライブラリか）を示すカラムである．また，dependant_name は依存元プラグインまたはライブラリの名前で，dependecy_library は依存先ライブラリの名前である．このテーブルを用いてソフトウェア間の依存関係を特定する．

互換性テーブル

互換性テーブルには，プラグイン-ライブラリ互換性評価値とライブラリ-ライブラリ互換性評価値が保存される．具体的には，以下の表５のような形式で保存される．各互換性テーブルは{id}_compatibility_table というテーブル名を持つ．ここで，{id}は dependency_table において保存されているライブラリまたは，プラグインの依存関係情報に関するレコードの id カラムに紐づけられている．また，dependant_version は依存元ライブラリまたは，プラグインのバージョン，dependency_version は依存先ライブラリのバージョン，compatibility はその互換性評価値が保存されている．

表 4: 依存関係テーブル

dependency_table		
PK or FK	カラム名	情報形式
PK	id	整数
	depdendant_type	文字列
	dependant_name	文字列
	dependency_library	文字列

表 5: 互換性テーブル

{id}_compatibility_table		
PK or FK	カラム名	情報形式
	depdendant_version	文字列
	dependency_version	文字列
	compatibilty	小数

互換性評価値は依存元ソフトウェアの名前とバージョン，依存先ソフトウェアの名前とバージョンと共に保存され，ソフトウェアバージョンの探索に利用される．ここからはデータベースへの問い合わせアルゴリズムについて 2 つのステップを用いて紹介する．

ステップ 1

ステップ 1 では依存関係テーブルにたいして問い合わせ（$selectFromDependecyTable()$）を行い，プラグインの依存するライブラリ（$depend_lib$）の特定を行う．さらに，依存するライブラリが依存するライブラリ（$depend_lib$）の特定を再帰的に行うことで，すべての依存関係（$dependency_pairs$）を特定することができる．以下にそのアルゴリズムを示す（Algorithm 6）．

ステップ 2

その後，依存関係にあるプラグインとライブラリ，ライブラリ同士の id（$table_id$）をもとに互換性テーブルへ問い合わせ（$selectFromDependecyTable()$）を行うことで，互換性情報（$compatibilities$）とプラグインの利用可能なバージョン一覧（$plugin_versions$）とライブラリの利用可能なバージョン一覧（$library_versions$）を取得することができる．以下にそのデータベースへの問い合わせアルゴリズムを示す（Algorithm 7）．

ソフトウェアバージョン探索モジュールではに入力として与えられたプラグインリストを基に，以上のアルゴリズムで依存関係情報と互換性情報の取得を行う．

Algorithm 6 互換性情報問い合わせ：ステップ１

```
plugins = input()
libraries = list()
dependency_pairs = list()
for plugin ← plugins do
    results = selectFromDependecyTable(plugin)
    for result ← results do
        depend_lib = result["depend_lib"]
        dependency_pairs ← result
        if depend_lib not inlibraries then
            libraries ← depend_lib
        end if
    end for
end for
stack = libraries()
while stack not empty do
    current_lib = stack.pop()
    results = selectFromDependecyTable(current_lib)
    for result ← results do
        depend_lib = result["depend_lib"]
        dependency_pairs ← result
        if depend_lib not inlibraries then
            libraries ← depend_lib
            stack ← depend_lib
        end if
    end for
end while
```

Algorithm 7 互換性情報問い合わせ：ステップ 2

```
plugins = inputFromStep1()
libraries = inputFromStep1()
dependency_pairs = inputFromStep1()
compatibilities = list()
plugin_versions = hash()
library_versions = hash()
for do dependency_pair ← dependency_pairs
    table_id = dependency_pair["id"]
    dependant_name = dependecy_pair["dependant_name"]
    dependant_type = dependecy_pair["dependant_type"]
    dependency_name = dependecy_pair["dependency_library"]
    results = selectFromCompatibilityTabe(table_id)
    comaptibilities ← results
    for result ← results do
        dependant_version = result["dependant_version"]
        dependency_version = result["dependency_version"]
        if dependant_type == "plugin" then
            if dependant_version not in plugin_versions["dependant_name"] then
                plugin_versions["dependant_name"] ← dependant_version
            end if
        else
            if dependant_version not in library_versions["dependant_name"] then
                library_versions["dependant_name"] ← dependant_version
            end if
        end if
        if dependency_version not in library_versions["dependency_name"] then
            library_versions["dependency_name"] ← dependency_version
        end if
    end for
end for
```

5.7 ソフトウェアバージョン探索モジュール

ソフトウェアバージョン探索モジュールではプラグインのリストを基に，4.3 で紹介した手法で整数非線形計画問題を作成してソルバーに入力する．ソルバーから得られた出力を基にライブラリのバージョンリストと互換性評価平均値を出力する．具体的な処理については，以下に示す３つのステップで行われる．

ステップ１

ステップ１では入力を基に依存関係と互換性評価値の取得を行う．入力としてプラグインリスト（$plugins$）を受け取り，それを基に互換性情報データベースからプラグインの依存するライブラリの名前とバージョンの取得（$getDependLib()$）と互換性評価値の取得（$getCompatibility()$）をする．依存するライブラリ（$libraries$）についてさらに依存するライブラリの名前とバージョン，互換性評価値を再帰的に互換性情報データベースから取得する．以下にそのアルゴリズムを示す（Algorithm 8）．

ステップ２

ステップ２では，ソルバーに入力する整数非線形計画問題クラスのインスタンス（$problem$) を生成し（$Problem.new()$），目的関数，制約を入力し，出力を得る．まず初めに, 変数（$variables$）をデータベースから取得されたライブラリとプラグインのバージョンごとに生成する. 変数は，4.3 節の変数の生成で紹介した手法で作成される（$createVariable()$）．さらに，変数と互換性評価値（$compatibilities$）を用いて目的関数（$objective$）を生成する（$createObjective()$）．目的関数は 4.3 節の目的関数の生成で紹介した手法で作成する．また，制約（$constraints$）についても各プラグイン，ライブラリ毎に 4.3 節の制約の生成で紹介した手法で作成する（$createConstraint()$）．生成された目的関数と制約をソルバーに入力し（$problem.add()$），バージョン組み合わせの探索（$problen.solve()$）を行うことでソルバーからの出力（$result$）を得る．以下にそのアルゴリズムを示す（Algorithm 9）．

ステップ３

ステップ３ではソルバーからの出力をインストールスクリプト（$install_script$）と互換性評価値平均（$compatibility_average$）として出力する．ソルバーからの出力は変数群（$variable_results$）と目的関数の値（$objective_score$）で構成される．各変数（$variable$）に関して０か１かを調査し，変数が１となる場合は該当変数のライブラリの名前とバージョンに関するインストールコマンドの生成（$createInstallCommand()$）とインストールスクリプトへの追記を行う．また，目的関数の値をプラグイン-ライブラリ間とライブラリ-ライブラリ間の依存関係（$compatibilities$）の合計数で割ることで，互換性評価値平均を算出する（$getAvarageScore()$）．その後，得られたインストールスクリプトと互換性評価値平均を出力する．以下にそのアルゴリズムを示す以下にそのアルゴリズムを示す（Algorithm 10）．

Algorithm 8 ソフトウェアバージョン探索：ステップ 1

$plugins = input()$
$compatibilities = list()$
$libraries = list()$
for $plugin \leftarrow plugins$ **do**
  $plugin_name = plugin["name"]$
  $compatibilities \leftarrow getCompatibility(plugin)$
  $libraries \leftarrow getDependLib(plugin)$
end for
$stack = libraries()$
while $stack\ not\ empty$ **do**
  $current_lib = stack.pop()$
  $compatibilities \leftarrow getCompatibility(current_lib)$
  $depend_libs \leftarrow getDependLib(current_lib)$
  if $depend_lids\ not\ in\ libraries$ **then**
    $libraries \leftarrow depend_lids$
    $stack \leftarrow depend_libs$
  end if
end while

6 手法の適応例

本節では試作した依存関係解決システムを依存関係問題へ適応した例について紹介する．ここでは 2.2 節で紹介した事例に対して試作したシステムを適応することで実際に依存関係問題が解決するかを確かめる．ここからはデータセット，実験手順，実験結果，考察の順に紹介する．

6.1 データセット

実験で用いるデータセットについて紹介する．実験では redmine-4.1，redmine_hourglass とそれぞれが依存するライブラリ 174 例を利用する．

依存するライブラリのバージョンに関してはセマンティックバージョニングに基づき，基本的にはメジャーバージョンのうち最新のもののみをデータセットに含めることとする．ただし，メジャーバージョンアップデートが極端に少ないライブラリ（本実験においてはメジャーバージョンアップデートが 4 回未満のライブラリとする.）については，ライブラリの各バージョン文字列を RubyGem リポジトリにおけるバージョン比較方法によってソートした結果に対して，バージョンの取得を行う（Algorithm 11）．ここで，$sorted_versions$ はソートされた各バージョン文字列のリストである．

Algorithm 9 ソフトウェアバージョン探索：ステップ２

```
plugins = inputFromStep1(0)
compatibilities = inputFromStep1(1)
libraries = inputFromStep1(2)
problem = Problem.new()
variables = list()
for plugin ← plugins do
    variables ← createVariable(plugin)
end for
for library ← libraries do
    variables ← cteateVariable(library)
end for
objective = createObjective(variables, comatibilities)
constraints = list()
for plugin ← plugins do
    constraints ← createConstraint(plugin, variables)
end for
for library ← libraries do
    constraints ← createConstraint(library, variables)
end for
problem = problem.add(objective, constraints)
result = problen.solve()
```

Algorithm 10 ソフトウェアバージョン探索：ステップ３

```
result = inputFromStep2(1)
compatibilities = inputFromStep2(2)
install_script = string()
variable_results = result.getVariables()
objective_score = result.getObjScore()
for variable ← variables do
    if variable.value() == 1 then
        install_script ← createInstallCommand(variable)
    end if
end for
compatibility_average = getAvarageScore(objective_score, compatibilities)
```

Algorithm 11 バージョンの選択

$sorted_versions = input()$
$selected_versions = list()$
for $i \leftarrow range(1..5)$ **do**
　$len = sorted_versions.length()$
　$selected_versions \leftarrow sorted_versions[int(len * i/5)]$
end for
$return \quad selected_versions$

図 8: 入力リスト

6.2 手順

まず初めに前処理として，Redmine-4.1 と redmine_hourglass-1.1.0 の依存するライブラリとその依存関係を依存情報収集モジュールへ入力する．依存情報収集モジュールはプラグインの Gemfile と依存するライブラリを収集する．プラグイン互換性評価モジュールは Gemfile と依存するライブラリの情報を基に互換性評価値を互換性情報データベースに保存する．また，ライブラリ互換性評価モジュールは依存関係にあるライブラリのソースコードを基に互換性評価値を互換性情報データベースに保存する．次に，Redmine-4.1 と redmine_hourglass-1.1.0 のリストをバージョン探索モジュールに与えることで，依存関係にあるライブラリのバージョンリストを得る．最後に，Redmine-4.1 と redmine_hourglass をインストールしたのちライブラリ出力されたバージョンリストの通りにライブラリのインストール，redmine と redmine_hourglass が正しく実行できるかテストを行う．

6.3 結果

以下の図 8 がバージョン推薦実行時に入力されたリストであり，図 9 がその出力結果である．出力結果についてはその一部を示す．

データセット，適応手順をもとに手法の検証を行った．その結果，Redmine の起動に必要となる Rails コマンドの実行が，ソースコードのうち依存関係のあるライブラリのバージョンについて静的に指定した部分による例外処理によって停止した．さらに検証を進めるため，静的に指定した部分のコメントアウトを行い，Rails コマンドを再度実行した．すると，Rails コマンドの実行時にエラーが発生した．このエラーは railties ライブラリが依存する activesupport ライブラリの transeform_valuesAPI が不足していたことによるもので

```
listen-0.4.6,
rb-inotify-0.10.1,
rb-fsevent-0.11.2,
deckar01-task_list-3.0.alpha2,
sanitize-5.2.3,
• • •
i18n-0.9.5,
• • •
faker-0.9.5,
• • •
```

図 9: 出力バージョンリスト

あった．

6.4 考察

Redmine 及び Redmine プラグインのテストを実行するには Rails コマンドの実行が必須となるが，インストールスクリプトの実行によって得られたライブラリ構成では，Rails コマンドの実行時にエラーが発生した．その理由としては，依存元ライブラリが必要とする依存先ライブラリの API の不足が考えられる．本手法では，すべての依存元ライブラリが呼び出すすべての API を依存先ライブラリの提供する API が満たされない場合でも，可能な限り API 呼び出しを満たすバージョン構成を出力する．この際，各 API は同等のものとして扱われる．これによって，多くのライブラリやメソッドから呼び出される，API を含まないバージョン構成が出力されたことが考えられる．

7 まとめ

本論文では研究背景として Redmine の依存関係のその問題について，実際の依存関係問題事例を用いて説明し，今回取り扱う依存関係問題のユーザにとっての解決不可能性やプラグイン開発者による解決例を示した．先行研究としては C#における例と Python における例を紹介し，Ruby，Redmine の依存関係問題に対する新たな手法が必要であることを示した．さらに，本研究における手法として，Gemfile の解析を用いたプラグイン-ライブラリ間の互換性評価方法とソースコード解析を用いたライブラリ間の互換性評価方法，数理計画ソルバーと整数非線形計画問題を用いたバージョン組み合わせ探索方法について説明した．手法を踏まえた実装として，試作システムのアルゴリズムを紹介した．また，互換性情報データベースについてその詳細を紹介した．提案手法の適応事例として，適応する依存関係問題とそのデータセットや適応手順について紹介し，その後適応結果を示した．

ここからは今後の課題について述べる．まず初めにライブラリ間の互換性評価方法に関してである．本手法では，4.2 の前処理 2 で紹介した手法でライブラリ同士の互換性について評価を行った．その際，依存元ライブラリが呼び出す API のうち依存先ライブラリの API がいくつ満たしているかを判定することで，評価を行っている．この時，API は同等のものとして扱われており，10 種類の API のうち 5 種類が見たされていれば互換性評価値は 0.5 などといった評価をしている．このままでは，ほとんど外部ライブラリから呼び出されない API と様々な外部ライブラリから呼び出される API が同じものとして評価される．これに対する解決策として，API への重み付けを行い，外部ライブラリから呼び出される API ほど互換性評価時に重視する手法を採用する予定である．具体的にはソフトウェア部品グラフ [23] とページランクアルゴリズム [29] を用いた重みづけ手法である．ここで，ソフトウェア部品とはソフトウェアの構成要素である，モジュールやクラス，メソッドを指し，ソフトウェア部品グラフはソフトウェア部品をノードとしたとき，その利用関係を有効エッジとしたグラフのことである．API の重み付けを行う際には，メソッドをノード，外部ライブラリへの呼び出しをエッジとしたソフトウェア部品グラフを構成する．このソフトウェア部品グラフに対して，ページランクアルゴリズムを用いたノードへのスコア算出を行う．外部ライブラリからの呼び出しが多いほどスコアは高くなる．このスコアを用いて API に重み付けを行うことで，外部ライブラリからの呼び出しを考慮した互換性評価を行うことができる．また，API の完全修飾名のみを用いて互換性の評価を行ったが，API の引数も考慮した互換性判定を行うことで，より正確な互換性の評価も行うことができる．

さらに，プラグイン-ライブラリ間の互換性判定については Gemfile と利用可能なライブラリのバージョンのみを用いた解析を行ったが，プラグインのソースコードも考慮した互換性評価を行うことで 1 と 0 以外の値も取る互換性評価値を割り当てを行う．これを行うことにより，より詳細なプラグイン-ライブラリ間の互換性評価を行うことができる．

さらに，ここからは本研究で紹介した手法のうち，本処理を Ruby，Redmine 以外での言語，ソフトウェアで利用することについて述べる．本手法では Ruby 言語，その中でも Redmine を題材とした依存関係解析をし，ソフトウェア間の依存関係について数値を用いた評価を行い，その評価値をデータベースへ保存した．さらに，保存した評価値を用いて依存

関係問題を整数非線形計画問題へ帰着することで，問題への解答を図った．本研究では依存関係の評価部分を 4.2 節で前処理，整数非線形計画問題への帰着を 4.3 節で本処理として紹介した．ここで前処理では Ruby ライブラリのソースコードの解析，プラグインの Gemfile への解析を行っているため Ruby 言語，Remdine に依存した解析を行っている部分とみなすことができる．これを言語依存部分と呼ぶこととする．これに対して，本処理では算出された互換性評価値，依存関係情報を基に整数非線形計画問題の作成とソルバー探索を行っている．ここで依存関係情報をソフトウェアをノードと依存関係をエッジとしたグラフと考えると Ruby 以外の言語においても，ソフトウェアにおける依存関係情報は同様の形式で表現されることが分かる．ここで，Ruby 以外の言語において，依存関係にあるソフトウェア間のバージョンごとの互換性が数値で表現可能であるとすると，Ruby 以外の言語においても，互換性評価値と依存関係情報は本手法において扱ったものと同様の形式で求められることが分かる．この時，本処理は言語に依存した解析を行っていない部分とみなすことができる．これを非言語依存部分と呼ぶ．たとえば，Python ソフトウェアに関して互換性評価値を算出し，データベースへ保存する言語依存部分の実装を行ったとする．そうすると，本手法における非言語依存部分を流用することで，Ruby ソフトウェアのみならず Python ソフトウェアに関しても本研究で紹介した 4.3 節の本処理を用いた依存関係問題解決を行うことができると考えられる．これを様々な言語に適応することで，複数の言語の依存関係問題を解決するシステムの実装が可能であると考えられる．

謝辞

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 肥後芳樹 教授には，研究活動において貴重な御指導及び御助言を賜りました．肥後 教授に心より深く感謝いたします．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 松下誠 准教授には，研究の方針から本論文の執筆に至るまで，直接の御指導及び御助言を賜りました．松下 准教授に心より深く感謝いたします．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 神田哲也 助教には，研究活動において適切な御助言を賜りました．神田 助教に心より深く感謝いたします．最後に，その他様々な御指導及び御助言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ならびに事務職員 軽部 瑞穂氏に心より深く感謝いたします．

参考文献

[1] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for javascript. *ACM Comput. Surv.*, Vol. 50, No. 5, 2017.

[2] Kurt M Bretthauer and Bala Shetty. The nonlinear knapsack problem – algorithms and applications. *European Journal of Operational Research*, Vol. 138, No. 3, pp. 459–472, 2002.

[3] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. Taming behavioral backward incompatibilities via cross-project testing and analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, p. 112–124, New York, NY, USA, 2020.

[4] 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*, p. 451–461, New York, NY, USA, 2022.

[5] Inc. Chuo Computer Co. 【社内事例】redmine 適用の推進｜貴社ビジネスの it 共創者、中央コンピューター株式会社.

[6] Xingliang Du and Jun Ma. Aexpy: Detecting api breaking changes in python packages. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, pp. 470–481, 2022.

[7] faker ruby. Faker.

[8] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. Escaping dependency hell: finding build dependency errors with the unified dependency graph. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, p. 463–474, New York, NY, USA, 2020.

[9] Python Software Foundation. Pypi · the python package index.

[10] The Rails Foundation. Ruby on rails — a web-app framework that includes everything needed to create database-backed web applications according to the model-view-controller (mvc) pattern.

[11] LLC GUROBI OPTIMIZATION. Gurobi optimizer - gurobi optimization.

[12] hicknhack software. Redmine 4.1.1 and ruby 2.6.6 dependency errors.

[13] hicknhack software. Redmine hourglass.

[14] 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*, p. 477–488. IEEE Press, 2020.

[15] Huaxun Huang, Ming Wen, Lili Wei, Yepang Liu, and Shing-Chi Cheung. Characterizing and detecting configuration compatibility issues in android apps. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, p. 517–528, 2022.

[16] Huaxun Huang, Chi Xu, Ming Wen, Yepang Liu, and Shing-Chi Cheung. Conffix: Repairing configuration compatibility issues in android apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, p. 514–525, New York, NY, USA, 2023.

[17] Ecma International. Ecma-262 - ecma international.

[18] Jean-Philippe Lang. Overview - redmine.

[19] Jean-Philippe Lang. Plugins - redmine.

[20] 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*, p. 1545–1557, New York, NY, USA, 2022.

[21] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*, p. 672–684, New York, NY, USA, 2022.

[22] Google LLC. Android リリース | android デベロッパー | android developers.

[23] 市井誠, 松下誠, 井上克郎. Java ソフトウェアの部品グラフにおけるべき乗則の調査. 電子情報通信学会論文誌 D, Vol. J90-D, No. 7, pp. 1733–1743, 2007.

[24] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. Fixing dependency errors for python build reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, p. 439–451, New York, NY, USA, 2021.

[25] Inc. npm. npm | home.

[26] Inc. npm. npm-audit | npm docs.

[27] Oracle. Consolidated jdk 21 release notes.

[28] Oracle. Mysql.

[29] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking : Bringing order to the web. In *The Web Conference*, 1999.

[30] Jibesh Patra, Pooja N. Dixit, and Michael Pradel. Conflictjs: finding and understanding conflicts between javascript libraries. In *Proceedings of the 40th International Conference on Software Engineering*, p. 741–751, New York, NY, USA, 2018.

[31] The pip developers. pip documentation v24.0.

[32] PyPA. virtualenv.

[33] ruby i18n. Ruby i18n.

[34] RubyGems team. Rubygems basics - rubygems guides.

[35] David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes. In *Proceedings of the 41st International Conference on Software Engineering*, p. 339–349, 2019.

[36] Chao Wang, Rongxin Wu, Haohao Song, Jiwu Shu, and Guoqing Li. smartpip: A smart approach to resolving python dependency conflict issues. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA, 2023.

[37] Chao Wang, Rongxin Wu, Haohao Song, Jiwu Shu, and Guoqing Li. smartpip: A smart approach to resolving python dependency conflict issues. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA, 2023.

[38] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, p. 1556–1560, New York, NY, USA, 2020.

[39] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, p. 54–65, New York, NY, USA, 2005.

[40] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Smallworld with high risks: a study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium*, p. 995–1010, USA, 2019.

[41] Ruby コミュニティ. オブジェクト指向スクリプト言語 ruby.

[42] セキュリティ・情報化推進部スーパーコンピュータ活用課. Coda チケット管理システム | jss@jaxa.

[43] 数値予報課. 数値予報モデル開発のための基盤整備および開発管理.