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[1,a)]　松下 誠[1,b)]　肥後 芳樹[1,c)]

概要：Redmine は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 には多数のプラグインが存在し，それ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．本報告では，整数非線形計画問題を用いて，Redmine プラグイン依存関係問題を解決する手法について述べる．本手法では，事前にソースコードを解析し，ソフトウェア同士の互換性について評価を行って，その結果をデータベースに保存する．そのデータに対して整数非線形計画問題を適応することで依存関係問題の解決を行う．また，手法に基づいた依存関係問題解決システムの試作について述べる．

キーワード：Ruby，Redmine，依存関係問題，互換性問題

1. まえがき

ソフトウェアの依存関係 [9] 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 [3]．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 [11]．本報告では，ソフトウェアの依存関係問題，その中でも，Redmine [5] における依存関係問題を扱う．

Redmine は Ruby [12] 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，様々な企業のプロジェクトで採用されている．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．

本報告では，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ソースコードベースの解析を行い，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．保存された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帰着させる．整数非線形計画問題をソルバーに入力することで依存関係のあるソフトウェアコンポーネントの実行可能性の高いバージョン構成を取得することができる．

以降，2 節では本報告の背景として，Redmine プラグイン [6] の依存関係問題について紹介する．さらに，3 節では本報告における関連研究について，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述べる．また，6 節では評価実験について述べる．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，本報告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．

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Suita, 565–0871, Japan
[a)] tyngtmyn@ist.osaka-u.ac.jp
[b)] matusita@ist.osaka-u.ac.jp
[c)] higo@ist.osaka-u.ac.jp

2.1 Redmine

Redmine とは，Ruby で作られたプロジェクト管理用のウェブアプリケーションフレームワークであり，Ruby on Rails (Rails)[*1] を利用して作られているオープンソースソフトウェアである．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[*2]．

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．

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が依存元からの制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．

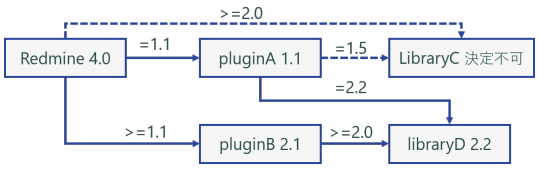

図 1 Redmine における依存関係の例

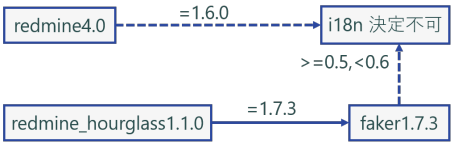

図 2 Redmine プラグイン依存関係問題の発生

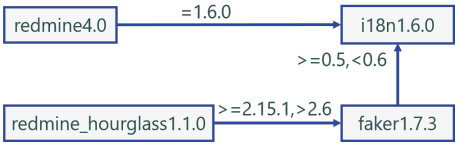

図 3 Redmine プラグイン依存関係問題の解決

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 以上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[*3] 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[*4] とプラグインである redmine_hourglass-1.1.1[*5] がインストールされており，redmine-4.1 は i18n[*6]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 に対応するように制約を “>= 2.15.1,<2.6” に変更する修正を行うことにより対応しているが（図 3），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．

[*1] https://rubyonrails.org/
[*2] 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 CODAhttps://www.jss.jaxa.jp/about_coda/ を運用している．
[*3] https://github.com/hicknhack-software/redmine_hourglass/issues/139
[*4] https://www.redmine.org/news/127
[*5] https://github.com/hicknhack-software/redmine_hourglass/
[*6] https://github.com/ruby-i18n/i18
[*7] https://github.com/faker-ruby/faker

3. 関連研究

本節では C#言語と Python 言語に関する依存関係問題を扱った研究について紹介する．

3.1 Nufix

Nufix [7] 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 線形計画問題に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を基に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ができる．Nufix を実際の依存関係問題事例 262 件に適応し，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．

3.2 PyCRE

PyCRE [2] 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 [4] 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．

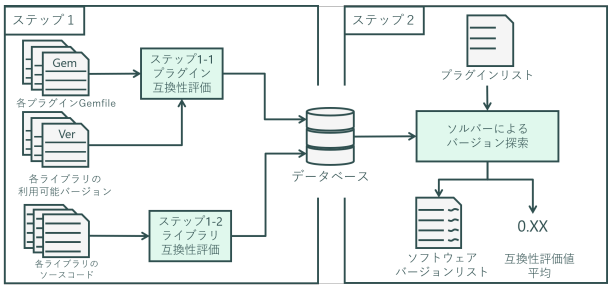

図 4 手法の概要

4. 提案手法

本節では，前節で紹介した 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 2 つのステップで構成される．ステップ 1 では前処理として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．

ステップ 2 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 ステップ 1：互換性の評価

ステップ 1 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．

ステップ 1-1：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．ステップ 1-1 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 “...,2.2.0,2.3.0,2.4.0,2.4.1,2.4.2” 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．

ステップ 1-2：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．ステップ 1-2 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．

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．ライブラリーライブラリの方で実例を持ってくると大きくなりすぎるので，架空の例を

図 5 redmine_hourglass-1.1.0 の Gemfile

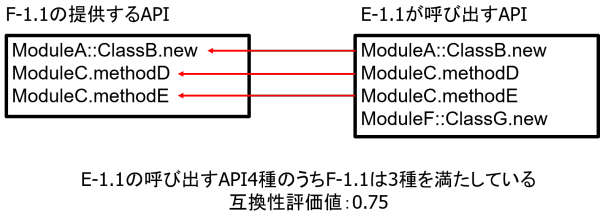

図 6 API 照合

使ってます．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．

4.3 ステップ 2：ソルバーによるバージョン探索

ステップ 2 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

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

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

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 [1] 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．

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

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

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

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

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．

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

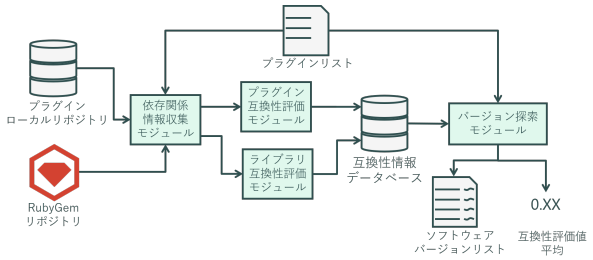

図 7 試作システムの概要

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．

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．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．

情報収集モジュールは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ーライブラリ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ー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基に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 を用いて実装した．

以降，利用するリポジトリとシステムの各モジュールについて説明する．

Algorithm 1 情報収集

```
plugins = input()
libraries = list()
for plugin ← plugins do
    getDepencyInfo(plugin) #依存関係情報の取得
end for
for library ← libraries do
    downloadLibrary(library) #ライブラリのダウンロード
end for
```

Algorithm 2 プラグイン互換性判定

```
gemfiles = input(1) #各プラグインの Gemfile
vers = input(2) #各ライブラリの利用可能なバージョン
for gemfile ← gemfiles do
    compatilities ← procGemfile(gemfile, vers) #プラグイン-ライブラリ互換性評価値の算出
    storeCompatibility(compatibility) #互換性評価値の保存
end for
```

5.2 利用するリポジトリ

RubyGem リポジトリは 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-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

情報収集モジュールでは入力として利用するプラグインの名前を入力することで依存するライブラリの依存関係情報とソースコードを RubyGem リポジトリから取得する（Algorithm 1）．

5.4 プラグイン互換性判定モジュール

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 と利用可能なライブラリのバージョンを与えることで，プラグインとライブラリの各バージョンにおける互換性評価値を 4.2 節のステップ 1-1 で説明した手法で算出し，互換性情報データベースに保存する（Algorithm 2）．

Algorithm 3 ライブラリ互換性判定

```
source_codes_pair_list = input() #依存関係にあるライブラリのソースコードのペアをリストにしたもの
for source_codes_pair ← source_codes_pair_list do
    dependency_codes = source_codes_pair[0] #依存先の各バージョンのソースコード
    dependant_codes = source_codes_pair[1] #依存元の各バージョンのソースコード
    for dep_code ← dependency_codes do
        for dant_code ← dependant_codes do
            score = getCompatibility(dep_code, dant_code) #互換性評価値の算出
            storeCompatibility(score) #互換性評価値の保存
        end for
    end for
end for
```

Algorithm 4 ソフトウェアバージョン探索

```
plugins = input()
compatibilities = getCompatibilities(plugins) #互換性評価値の取得
problem = createProblen(comaptibilities) #整数非線形計画問題の作成
results = Solve(problem) #ソルバーによる出力の取得
OutPut(results) #ソルバー出力の加工
```

5.5 ライブラリ互換性判定モジュール

ライブラリ互換性評価モジュールでは依存関係にあるライブラリのソースコード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 節のステップ 1-2 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（Algorithm 3）．

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互換性判定で解析されたプラグイン-ライブラリ，ライブラリ-ライブラリの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．

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入力として利用したいプラグインのリストを与えると，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（Algorithm 4）．

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

Algorithm 5 バージョンの選択

```
sorted_versions = input()
selected_versions = list()
for i ← range(1..5) do
    len = sorted_versions.length()
    selected_versions ← sorted_versions[int(len * i/5)]
end for
return selected_versions
```

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 5）．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．

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．

```
redmine-4.1,
redmine_hourglass-1.1.0
```

図 8 入力リスト

```
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

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 を実行するために必要な 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 transe-form_valuesAPI が不足していたことによるものであった．

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を出力することが考えられる．

7. まとめ

本報告では Redmine プラグイン依存関係問題を解決する手法と試作システムについて述べた．また，試作したシステムを用いて既存の依存関係問題事例に対する本手法の適応を行った．

今後は，ライブラリ間の互換性判定において API の照合をする際に依存先ライブラリの各 API に重みを付ける手法により，より実行可能性の高いバージョン構成を出力することができると考えられる．API の重み付けをする手法として，メソッドをノード，メソッド呼び出しをエッジとしたソフトウェア部品グラフ [8] の構成を行い．ページランクアルゴリズム [10] などの各ノードの重要度を表す指標を算出することで API への重みづけを行うことが考えられる．

謝辞 本研究は JSPS 科研費 22K11985 の助成を受けたものです．

参考文献

[1] Bretthauer, K. M. and Shetty, B.: The nonlinear knapsack problem – algorithms and applications, *European Journal of Operational Research*, Vol. 138, No. 3, pp. 459–472 (online), DOI: 10.1016/S0377-2217(01)00179-5 (2002).

[2] Cheng, W., Zhu, X. and Hu, W.: Conflict-aware inference of python compatible runtime environments with domain knowledge graph, *Proceedings of the 44th International Conference on Software Engineering*, New York, NY, USA, p. 451–461 (online), DOI: 10.1145/3510003.3510078 (2022).

[3] Fan, G., Wang, C., Wu, R., Xiao, X., Shi, Q. and Zhang, C.: Escaping dependency hell: finding build dependency errors with the unified dependency graph, *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA, p. 463–474 (online), DOI: 10.1145/3395363.3397388 (2020).

[4] Horton, E. and Parnin, C.: V2: fast detection of configuration drift in Python, *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, IEEE Press, p. 477–488 (online), DOI: 10.1109/ASE.2019.00052 (2020).

[5] Lang, J.-P.: Overview - Redmine, Jean-Philippe Lang (online), available from ⟨https://www.redmine.org/⟩ (accessed 2024-01-26).

[6] Lang, J.-P.: Plugins - Redmine, Jean-Philippe Lang (online), available from ⟨https://www.redmine.org/plugins⟩ (accessed 2024-01-26).

[7] Li, Z., Wang, Y., Lin, Z., Cheung, S.-C. and Lou, J.-G.: Nufix: escape from NuGet dependency maze, *Proceedings of the 44th International Conference on Software Engineering*, New York, NY, USA, p. 1545–1557 (online), DOI: 10.1145/3510003.3510118 (2022).

[8] 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).

[9] Mukherjee, S., Almanza, A. and Rubio-González, C.: Fixing dependency errors for Python build reproducibility, *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, New York, NY, USA, p. 439–451 (online), DOI: 10.1145/3460319.3464797 (2021).

[10] Page, L., Brin, S., Motwani, R. and Winograd, T.: The PageRank Citation Ranking : Bringing Order to the Web, *The Web Conference* (1999).

[11] Wang, C., Wu, R., Song, H., Shu, J. and Li, G.: smartPip: A Smart Approach to Resolving Python Dependency Conflict Issues, *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA, (online), DOI: 10.1145/3551349.3560437 (2023).

[12] Ruby コミュニティ：オブジェクト指向スクリプト言語 Ruby，Ruby コミュニティ（オンライン），入手先 ⟨https://www.ruby-lang.org/ja/⟩ （参照 2024-01-26）．