特別研究報告

題目

Python エコシステムにおけるバージョン上限の継続的更新の提案

指導教員

楠本 真二 教授

報告者

三倉 孝太

令和 8 年 2 月 9 日

大阪大学基礎工学部情報科学科

令和 7 年度 特別研究報告

Python エコシステムにおけるバージョン上限の継続的更新の提案

三倉 孝太

内容梗概

ソフトウェア開発において依存関係の宣言とその解決は必須のプロセスである．しかしながら，宣言された依存関係は潜在的な互換性破壊の可能性を内包している．例えば，ある時点で宣言された依存は未来のある時点で互換性破壊の要因となり得る．この破壊は依存先パッケージの更新に起因しており，宣言時点では予見することは不可能である．本研究の目的は，依存関係の解決における潜在的な互換性破壊の低減と互換性を満たす解発見に要する計算コストの削減である．そのためにバージョン上限の宣言，及びその継続的な更新という手法を提案する．互換性破壊を引き起こす本質的要因は上限の未指定であり，互換性を満たすバージョン上限を明記することでその問題を解決する．バージョン上限は時間的な推移に伴って変化する情報であるため，継続的な更新が必須である．本研究では提案手法の有効性を確かめるために，Python エコシステムのシミュレーションを行った．評価実験の結果，従来手法では検出できない互換性破壊の発生を完全に防止し，先行研究の手法と比較して互換性を満たす解発見に要する計算コストを約 45% 削減したことを確認できた．また，多くのパッケージに利用される主要なパッケージが優先的に提案手法を採用することで，全体の採用率が低い過渡期においても有効であることも確認できた．

主な用語

Python，依存関係の解決，破壊的変更，後方互換性，シミュレーション

目次

図目次

1　はじめに

依存関係の解決とは，ソフトウェアが必要とする他のライブラリやパッケージ（以降パッケージ）を自動的に取得するプロセスである．このプロセスにおいては，開発者が宣言した依存関係の要求に基づき，その要求を満たす各依存パッケージのバージョンを自動的に特定する．Python の場合，requirements.txt などの要求ファイルに `pandas>=1.5` のような依存の要求を宣言する．更に pip などのパッケージマネージャにより依存関係の解決を実行することで，要求を満たす可能な限り新しいバージョンの pandas が取得される．2026 年 2 月現在では pandas の最新バージョンは 3.0.0 であるため，`pandas==3.0.0` が依存関係の解決結果となる．また，pandas 自体も他のパッケージへの依存，すなわち推移的な依存を持つ．よって上記の解決手順を再帰的に処理し，すべての要求に矛盾がない解を最終的に得る．

依存関係の解決においては，時間的な推移に伴う互換性破壊が起きることが知られている [1, 2]．一般的なパッケージマネージャが実施する依存関係の解決処理は，バージョン要求を充足する解の発見に限定されている．よって，発見した解がソースコードの観点での互換性を充足しているかは保証しない．そのため，宣言時点では互換性があったとしても未来のある時点で依存先パッケージの破壊的変更によって互換性が失われる可能性がある．コード互換性の破壊はコンパイルエラーや実行不能などの問題に繋がる．更には実行には成功するものの，その実行結果が変わるといった機能的なバグに繋がることもある．

バージョン互換とコード互換の両方を保つ解の発見方法として，PCREQ[3] が提案されている．PCREQ はソースコードに対する静的解析によりバージョン互換かつコード互換な解を自動推論する．より具体的には，依存元のソースコードと依存先パッケージのソースコードを対象として，メソッドのシグネチャレベルでコード互換の有無を解析する．この解析を直接依存から派生するすべての要求同士に適用する．これによって，各種パッケージ利用関係の中でのインタフェースの矛盾（つまりコード互換破壊）を発見可能である．

ここで，PCREQ のようなコード互換検証によって発見したバージョン互換かつコード互換な解を，Python エコシステム全体で共有できた場合を考える．あるソフトウェア X のバージョン互換かつコード互換な解は，X を利用する別の他者にとっても有益な情報である．破壊的変更を含む更新が行われた時点で X の依存宣言を更新できれば，X を利用するすべてのソフトウェアで発生し得るコード互換の問題を低減できる．また解の共有は，コード互換検証に要する計算コストの削減にも繋がる．コード互換検証は依存関係の解決という充足可能性問題の各ステップに対して，ソースコードの静的解析手順が加えられた仕組みだとみなせる．よって全体的に高コストである．もし Python エコシステム全体で解を共有できれば，この解の発見に要する計算コストの削減も可能となる．

本研究の目的は，Python エコシステム全体における潜在的な互換性破壊の低減とコード互換な解発見に要する計算コストの削減である．そのためにバージョン上限の宣言，及びその継続的な更新という手法を提案する．バージョン互換かつコード互換な最新のバージョン点を互換性を満たす上限とみなし，依存先パッケージの更新に伴って上限を更新する．本研究では，提案手法の有効性を確かめるために，Python エコシステムのシミュレーションを行った．その結果，従来手法では検出できない互換性破壊の発生を完全に防止し，PCREQ のように各開発者が独立に検証を行う手法と比較して互換性を満たす解発見に要するコストを約 45% 削減したことを確認できた．また，多くのパッケージに利用される主要なパッケージが優先的に提案手法を採用することで，全体の採用率が低い過渡期においても有効であることも確認できた．

2 準備

2.1 依存関係の解決

あるソフトウェアが他のパッケージを利用するとき，両者の間には依存関係が存在する．依存関係はパッケージ名とバージョン要求から構成され，ソフトウェアが直接依存するパッケージ（直接依存）とそのパッケージが依存するパッケージ（推移依存）に分類される．依存関係の解決は，推移依存を含めた依存関係全体のバージョン要求を同時に満たす各パッケージのバージョンを取得するプロセスである．Python のパッケージマネージャである pip は，要求を満たす範囲でなるべく最新のバージョンを取得する方針を採用している．例えば，`tensorflow>=2.18` という要求が宣言されている場合，2.18, 2.19, 2.20 など要求を満たすバージョンは複数あるが，2026 年 2 月時点で依存関係の解決を行うと最新バージョンである `tensorflow==2.20` が取得される．

2.2 潜在的な互換性破壊

依存関係の解決には，時間的な推移に伴う互換性破壊が起きることが知られている [1, 2]．実際に，Maven や npm，Go 言語のエコシステムなどを対象に，互換性破壊を含む更新の実態やその影響を調査した研究が複数存在する [4, 5, 6]．こうした問題が生じる根本的な要因は，バージョン要求を満たすかどうかといったバージョン互換のみを検証しており，実際にソフトウェアが正常に動作するために必要な API の整合性といったコード互換は検証していないことである．依存関係の解決や依存衝突の検知において広く研究が行われているが，smartpip[7] や Watchman[8]，LooCo[9] などその多くはバージョン互換にのみ焦点を当てている．一方で，コード互換を検証する研究として PyCRE[10] や PyEGo[11], ReadPyE[12] などが存在するが，これらは直接依存の検証に留まり推移依存の検証は行っていない．

また，Dietrich らは PyPI エコシステム上にあるパッケージの 33% がバージョン要求に下限のみを指定しており，上限を指定していたパッケージは 5〜6% 程度であったことを報告している [13]．このように下限のみを指定し上限を指定しないバージョン要求は，未来にリリースされるすべてのバージョンを許容する．そのため依存先パッケージが破壊的変更を含む更新をした場合，パッケージマネージャはそのコード非互換なバージョンを選択し互換性破壊を引き起こしてしまう．

例えば，あるプロジェクトが `tensorflow>=2.18` という要求を宣言していたとする．宣言時点での最新バージョンが 2.19 であり，プロジェクトはバージョン互換とコード互換を共に満たしていた．しかし，その後一部の API を削除したバージョン 2.20 がリリースされると，パッケージマネージャはこの新しいがコード非互換なバージョンを自動的に選択してしまう．結果として，宣言時点ではバージョン互換とコード互換の両方を満たしていた要求が，時間的な推移に伴いバージョン互換だがコード非互換な状態へと変化し，実行時エラーといった問題を引き起こす．本稿では，このように依存先パッケー

ジの未来の更新に起因する予見不可能な互換性の問題を潜在的な互換性破壊と呼ぶ.

2.3 潜在的な互換性破壊の解決策

潜在的な互換性破壊を解決する方法の 1 つとして，PCREQ[3] が提案されている．PCREQ は依存関係のバージョン要求を解決するだけでなく，ソースコードの静的解析を組み合わせてコード互換性を検証しバージョン互換かつコード互換な解を自動的に推論する手法である．このプロセスは以下の 6 つの主要なモジュールから構成される．

知識獲得：後の分析に必要となる依存パッケージの情報を収集する．まず，各依存パッケージのインストール可能なバージョンリストと各バージョンの依存関係のメタデータを収集する．次に，各バージョンのソースコードを解析し，提供するモジュールと API の情報を収集する．

バージョン互換性の評価：バージョン要求を SMT（Satisfiability Modulo Theories）式として表現し直し，バージョン要求を満たすバージョンの組み合わせを求める．まず，知識獲得で収集したメタデータから推移依存を含むすべてのバージョン要求を 1 つの SMT 式として表現する．次に，元の要求からの変更数を最小限に，変更がある場合は最新バージョンを優先する方針でこの SMT 式を満たす解（バージョン互換な組み合わせ）を求める．

呼び出し API とモジュールの抽出：コード互換性の検証に必要な，プロジェクトが実際に利用している API とモジュールを抽出する．まず，プロジェクトのソースコードを解析し，直接依存のパッケージから利用している API を抽出する．次に，パッケージのソースコードを解析し，抽出した API が利用している API やモジュールを再帰的に抽出する．

コード互換性の評価：バージョン互換性の評価で得られた解がコード互換を満たすかを検証する．モジュールと API 名，API パラメタに互換性があるかを知識獲得で収集した情報と呼び出し API とモジュールの抽出で収集した情報を比較し検証する．

バージョン変更：コード互換性の評価でコード非互換が検出された場合に，原因となったパッケージのバージョンを変更し，プロセスをバージョン互換性の評価に戻す．

不足パッケージの補完：バージョン互換とコード互換の問題を解決する過程で発見した，新しい推移依存のパッケージを補完する．補完するときは新しいバージョンで未知のエラーが発生しないよう，バージョン要求を満たす最も古いバージョンを選択する．

2.4 Python エコシステムでの互換性破壊の低減に向けて

ここで Python エコシステム全体において，潜在的な互換性破壊を低減する方法を考える．PCREQ はあくまで各開発者が自身の環境で実行するツールであり，発見されたバージョン互換かつコード互換な解を他の開発者と共有する仕組みは存在しない．しかし，発見された解は同じソフトウェアの実行を

試みる他の開発者にとっても有益な情報である．もし，Python エコシステム全体でこの解を共有できた場合，各開発者が独立に解析する必要はなくなり全体での互換性破壊を低減できる．

しかし，解の単純な共有は依存関係の解決における解空間の大幅な限定に繋がる．これは PCREQ が発見する解がロックファイルのように特定のバージョンを示す点で表現されているためである．解をバージョン要求としてそれに依存するソフトウェアの依存関係の解決を行うと，他の依存パッケージのバージョン要求と衝突する可能性が高くなってしまう．

したがって，PCREQ のようなコード互換性検証で発見された解をエコシステム全体で効率的に共有しつつ，依存関係の解決における解空間を維持するような新しい仕組みが求められる．

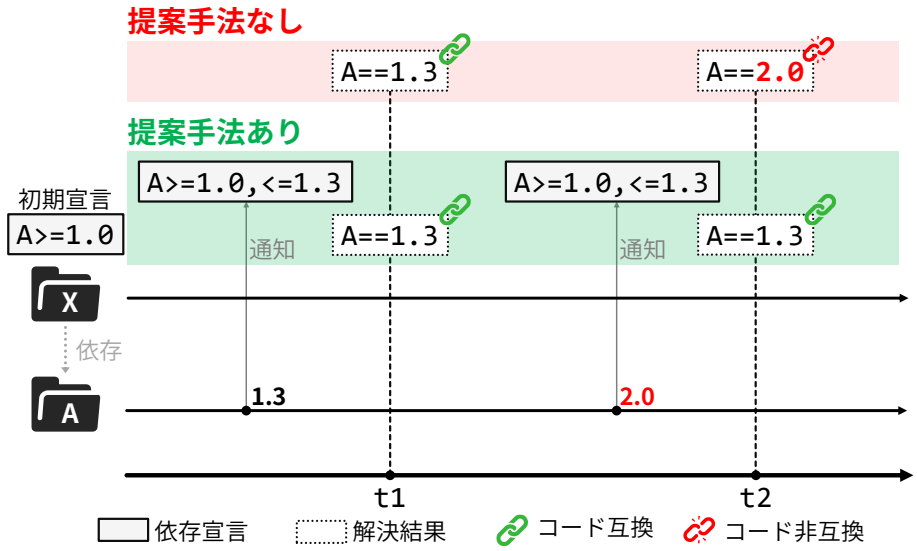


図 1: 提案手法の採用例

3 提案手法

3.1 概要

本研究の目的は，Python エコシステム全体における互換性破壊の低減とコード互換な解発見に要する計算コストの削減である．そのために，コード互換な解をバージョン要求の上限と定め，依存先パッケージの更新に伴って上限を更新するという手法を提案する．図 1 にその採用例を示す．ソフトウェア X はパッケージ A の要求を宣言している．従来の宣言方法では，コード互換性の有無に関係なく A の更新に伴い依存関係の解決で発見される解は変化する．時刻 t1 では A は破壊的変更を行っておらず，発見したバージョン `A==1.3` は互換性を満たしている．その後，A のメジャーアップデートが行われ，廃止メソッドの削除が行われたとする．時刻 t2 では，`A==2.0` が解となるが廃止メソッドの削除によってコード非互換が発生する．一方，提案手法を採用した場合では A の更新に伴い X の要求が更新される．A がバージョン 1.3 をリリースした時点では A は破壊的変更を行っていないため，X の要求は最新バージョンを含むよう `A>=1.0,<=1.3` と更新され，時刻 t1 では変わらず `A==1.3` が解として発見される．その後，バージョン 2.0 をリリースした時点では X と A の互換性が失われているため，X の要求は更新を停止して `A>=1.0,<=1.3` とする．そのため，時刻 t2 では `A==2.0` ではなくコード互換を満たす最新バージョンである `A==1.3` が解として発見される．以降では，提案手法を実現するための 2 つのアイデア，バージョン上限の宣言と継続的更新について説明する．

3.2 バージョン上限の宣言

提案手法では解の表現方法としてコード互換を保証するバージョン上限というアイデアを導入する．バージョン互換とコード互換を共に満たす最新のバージョン点を現時点の上限と定め，解空間が範囲となるよう表現し直す．2.2 節で説明した tensorflow の例を考える．2.19 が最新バージョンの際に宣言された `tensorflow>=2.18` という要求は，2.19 がバージョン互換とコード互換を共に満たす最新のバージョン点であるため，`tensorflow>=2.18,<=2.19` というバージョン上限を宣言する要求に書き換えられる．また，コード非互換な 2.20 がリリースされた場合は上限を更新せず，`tensorflow>=2.18,<=2.19` のままとする．このようにコード互換の範囲を上限を用いて指定することで，互換性破壊を低減しつつ点の指定と比較して依存関係の柔軟性を高く保つことができる．

前述の例では，上限の表現方法として`<=2.19` と`<2.20` の 2 つの選択肢が考えられるが，本研究では `<=2.19` を採用する．これは 2.19.1 のようなメンテナンスリリースが意図せず要求を満たすことを防ぎ，検証済みのバージョンのみを範囲に含めるためである．

また，上限となるバージョンを発見する方法として，コード互換検証といった静的解析やテストを用いた動的解析が考えられる．静的解析はどのソフトウェアでも適用できるが，意味的な互換性破壊を見逃す可能性がある．一方，動的解析は実行時の振る舞いを確認できるため信頼性は高いが，テストコードが必須であり適用できるソフトウェアが限られる．よって，静的解析を基本としつつ，動的解析はより厳密な検証が必要な場合に補助的に用いる構成とする．

3.3 継続的更新

バージョン上限は静的な情報ではなく依存先パッケージの更新に伴って変化する動的な情報である．そのため，従来の宣言方法と異なり継続的な更新が不可欠である．

上限を更新する素朴な方法として，定期的に更新する方法が考えられる．しかしこの方法は，更新のリアルタイム性と計算コストの間にトレードオフの関係が発生する．更新頻度を増やせば最新のバージョン情報がすぐに反映されるが無駄な更新が多く計算コストが増大し，減らせば計算コストは抑えられるが最新のバージョン情報を反映するまでに遅延が発生する．

そこで本研究では，より効率的な方法である Observer パターンに基づく更新を採用する．これはあるパッケージの更新を契機として，そのパッケージに直接依存するパッケージのバージョン上限を更新する方法である．最新のバージョンが互換性を満たす場合はバージョン上限を最新のバージョンに更新し，互換性を満たさない場合はバージョン上限を更新しない．この方法では，上限の更新は必要なタイミングにのみ行われるため，計算コストを抑えつつ最新のバージョン情報を遅延なく反映できる．また，バージョン上限の検証と更新は，更新されたパッケージに直接依存するパッケージのみが対象であり，

推移依存のパッケージまで更新を伝播させる必要はない．これは，推移依存のパッケージはあくまで直接依存のパッケージの API のみを利用しており，ソースコードの変更があったパッケージの API を直接利用していないためである．直接依存のパッケージが提案手法を採用する限り，推移依存のパッケージは元のパッケージ更新による互換性破壊の影響を受けることはない．

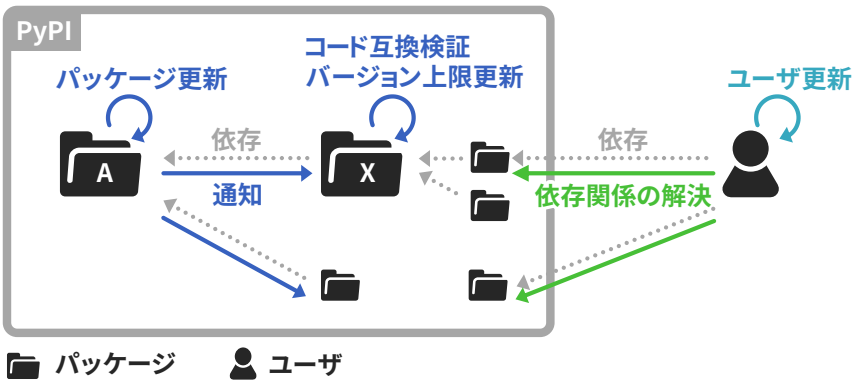


図 2: アクターとイベントの関係

4 Python エコシステムのシミュレーション

4.1 目的

本シミュレーションの目的は，3 節で提案した手法が Python エコシステム全体で採用された場合の互換性破壊の低減とコード互換な解発見に要する計算コストの削減の確認である．あるパッケージの更新が他の様々なプロジェクトへ影響を及ぼすため個別のプロジェクトを分析するだけでは不十分である．そこで提案手法の評価には，Python エコシステム全体を対象としたシミュレーションを用いる．

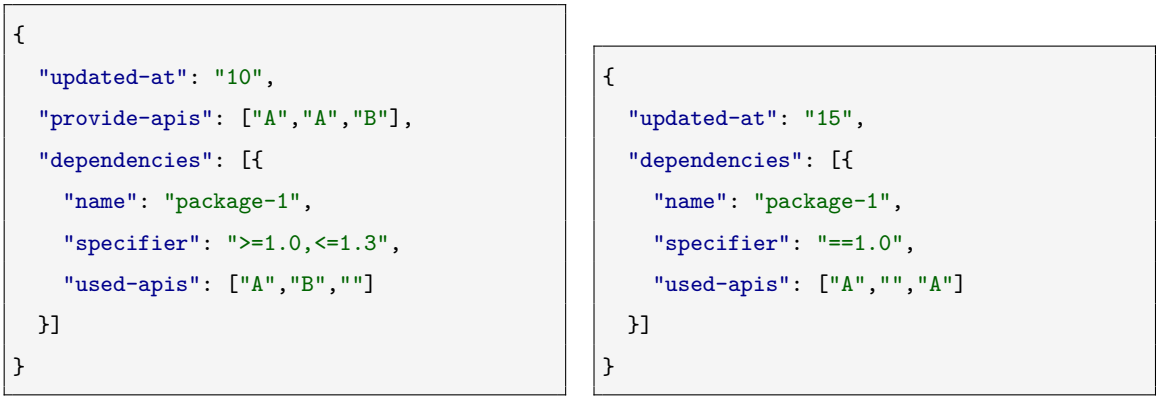
以下本節では，4.2 節でシミュレーションの構成要素であるアクターとイベントのモデルを定義する．続いて 4.3 節で現実の Python エコシステムを反映させるためのパラメタ設定について述べ，最後に 4.4 節で具体的なシミュレーションの実行手順を説明する．

4.2 Python エコシステムのモデリング

Python エコシステムとは，Python の標準パッケージリポジトリである PyPI を中心とした Python ソフトウェアの利用関係の構造を意味する．Python エコシステム内では，PyPI 上に公開されているパッケージ群は互いに依存し，PyPI の外側にいる多くの Python 開発者もまた PyPI 上に公開されているパッケージを利用する．この Python エコシステム全体を完全に模倣するのはその複雑さと規模から困難である．そこで本シミュレーションでは，Python エコシステムを構成する主要な要素とそれらが引き起こす事象を抽象化したデータモデルを設計する．

4.2.1 アクターの定義

本モデルは，Python エコシステムの主要な構成要素としてパッケージとユーザの 2 種類のアクターを定義する．アクターと後述するイベントの関係を 図 2 に示す．パッケージは PyPI 上に公開されて



```
{
  "updated-at": "10",
  "provide-apis": ["A","A","B"],
  "dependencies": [{
    "name": "package-1",
    "specifier": ">=1.0,<=1.3",
    "used-apis": ["A","B",""]
  }]
}
```

(a) パッケージのデータモデル

```
{
  "updated-at": "15",
  "dependencies": [{
    "name": "package-1",
    "specifier": "==1.0",
    "used-apis": ["A","","A"]
  }]
}
```

(b) ユーザのデータモデル

図 3: データモデル

いる配布パッケージを表すアクターである．パッケージは．そのバージョンに関するメタデータを持つ．このメタデータは 図 3(a) に示す JSON 形式で表現する．updated-at はメタデータが更新された時刻，provide-apis はそのパッケージが提供する API，dependencies はそのパッケージの依存関係を表す．なお，時刻は日付ではなく，シミュレーションの実行ターン数として表現する．依存関係はパッケージ名（name）とバージョン要求（specifier），利用する API（used-apis）から構成され，パッケージ開発の一般的な慣習に基づき`>=1.0.0` のような下限指定によって記述されるものとする．4.2.3 節で後述するが，提供する API と利用する API は `["A","","A"]` のようなリストで表現する．各インデックスは個々の API に対応しており，要素は対応する API のシグネチャを表現している．

ユーザは依存関係の解決を行う開発者を表すアクターである．ユーザは自身のプロジェクトで利用するパッケージの依存関係の宣言を持つ．この依存関係の宣言は 図 3(b) に示す JSON 形式で表現する．updated-at は依存関係を更新した時刻を表し，dependencies は宣言された依存関係を表す．ユーザが持つ依存関係は`==1.0.0` のような特定のバージョンに固定されたロックファイルとする．

これらデータモデルはある時刻におけるパッケージやユーザの状態を表している．提案手法を採用したパッケージの場合，バージョン上限の更新に伴って変化する．図 3(a) の例では，シミュレーション開始から 10 ターン目にメタデータが更新されたということを表している．

4.2.2 イベントの定義

本モデルでは，Python エコシステム内で発生する主要な出来事として，パッケージ更新とユーザ更新，依存関係の解決の 3 種類のイベントを定義する．

図 2 の青色矢印で示すパッケージ更新は，あるパッケージが新しいバージョンをリリースするイベントである．このイベントの発生によりリポジトリ上のバージョン情報が更新される．また，このイベ

ントに伴って提案手法を採用したパッケージのバージョン上限の更新を行う．図 2 ではパッケージ X が提案手法を採用している．この場合にパッケージ A がパッケージ更新を行うと，パッケージ X にその通知を行う．パッケージ X が通知を受け取ると，その新しいバージョンが自身と互換性があるかどうかを検証する．その結果，互換性があれば新しいバージョンを上限として更新し，互換性がなければ更新せず通知の受け取りを停止する．

図 2 の水色矢印で示すユーザ更新は，あるユーザがプロジェクトの依存関係を更新するイベントである．このイベントの発生によりユーザの依存関係のバージョン要求がその時点での最新バージョンに更新される．これは開発者の定期的なメンテナンスとして依存関係を更新する状況に相当する．

図 2 の緑色矢印で示す依存関係の解決は，あるユーザがプロジェクトの依存関係を解決し実際に利用するパッケージバージョンを決定するイベントである．本モデルでは以下の 2 種類の解決手法を定義する．1 つ目は pip に相当する一般的な解決手法である．これはバージョン互換のみを検証し解を探索する手法であり，以降これを従来手法と呼ぶ．2 つ目は PCREQ に相当する先行研究の解決手法である．これはバージョン互換に加え API の利用関係に基づいてコード互換を検証し解を探索する手法であり，以降これを既存手法と呼ぶ．

4.2.3 コード互換検証の定義

本シミュレーションでは，コード互換検証を以下のように単純化する．

パッケージとユーザはソースコードの代わりに API リストを持つ．パッケージは提供する API リストと利用する API リストの 2 種類，ユーザは利用する API リストの 1 種類である．各 API のシグネチャ（返り値，引数，API 名の組）を`"A"`などの一文字で表現し，リストの各インデックスは個々の API に対応している．パッケージ更新によりシグネチャが変更された場合，提供する API リストの対応する要素を`"B"`や`"C"`のように異なる文字に置き換える．利用する API リストの各インデックスに対応する API は提供する API リストと同じである．パッケージやユーザが利用する API には提供する API と同じ文字を割り当て，利用しない API には空文字（`""`）を割り当てる．

コード互換検証では，提供する API リストと利用する API リストを比較する．任意の空文字でない利用する API のシグネチャが提供する API のシグネチャと一致する場合に，コード互換であると判定する．例えば，パッケージ A のバージョン 1.0 で提供する API が `["A","A","B"]`，パッケージ A に依存するパッケージ X の利用する API が `["A","","B"]` である場合を考える．パッケージ X がパッケージ A のバージョン 1.0 を利用する場合，利用している 1 つ目と 3 つ目の API のシグネチャがともに提供する API のシグネチャと一致するため，コード互換であると判定される．その後，パッケージ A がバージョン 2.0 に更新され，提供する API が `["A","B","C"]` に変化したとする．この場合，1 つ目の API のシグネチャは一致するが 3 つ目の API はシグネチャが一致しないため，コード非互換であ

ると判定する．ここで，2 つ目の API のシグネチャも 3 つ目の API と同様に変化しているが，利用していない（空文字である）ためコード互換検証には影響しない．

4.2.4　アクター属性とイベント発生モデル

本シミュレーションでは現実に存在する多種多様なアクターを表現するため，各アクターに以下の属性を付与しこれに基づいてイベントを発生させる．

パッケージの属性は以下の 4 つである．

- update interval：パッケージ更新をする間隔
- instability：パッケージ更新が破壊的変更を含む確率
- dependencies：依存するパッケージの数
- dependents：依存されるパッケージの数

update interval の値が小さいほど高頻度で更新を行っていることを表し，instability の値が高いほど互換性破壊を含む更新を高頻度で行っていることを表す．

また，ユーザの属性は以下の 3 つである．

- update interval：ユーザ更新をする間隔
- resolve interval：依存関係の解決を行う間隔
- dependencies：依存するパッケージの数

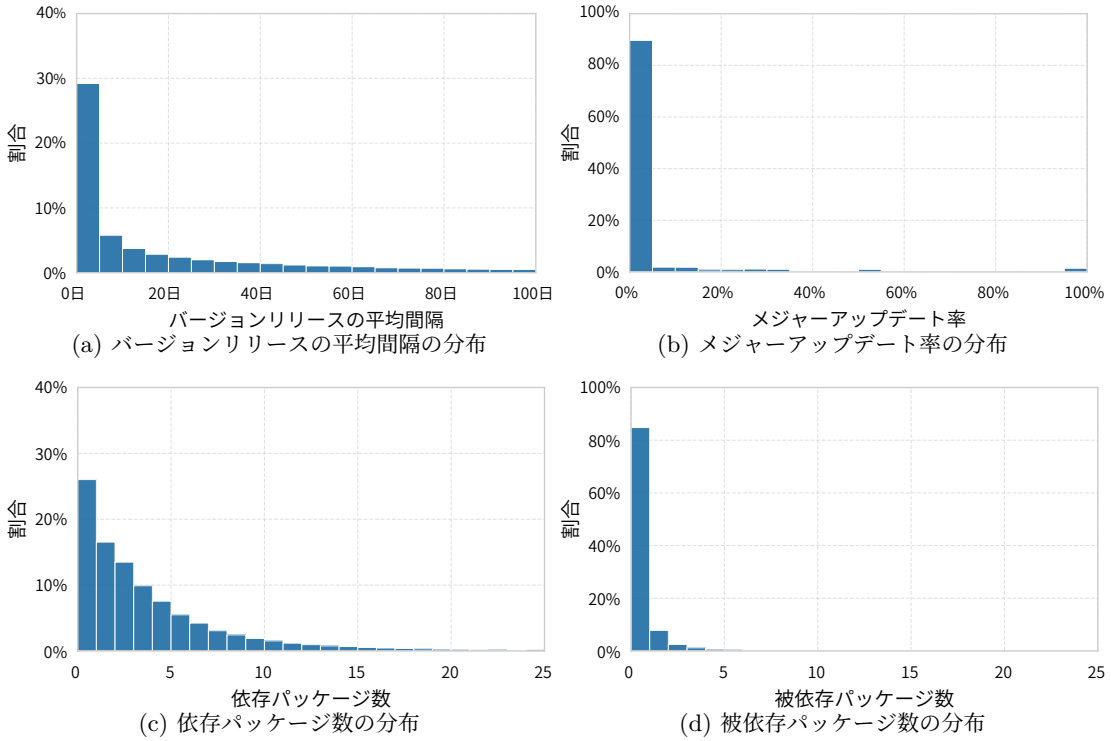
4.3　パラメタの設定

本節では，4.2.4 節で定義したアクターの属性値を決定するために行った，PyPI データセットの調査結果とそれに基づくパラメタ設定について述べる．

4.3.1　データセットと調査手法

本シミュレーションの妥当性を高めるためには，現実の Python エコシステムの実態に即したパラメタ設定が不可欠である．そこで，本研究では BigQuery 上で公開されている PyPI のデータセットを用いて実データの調査を行った．BigQuery とは，Google Cloud が提供する分析データウェアハウスである．調査対象は 2023 年 1 月 1 日から 2026 年 1 月 14 日までの期間にリリースされたパッケージバージョンのうちプレリリースやポストリリースを除外したリリースとし，以下の 4 点を調査した．

- バージョンリリースの平均間隔
- メジャーアップデート率
- 依存パッケージ数
- 被依存パッケージ数



(a) バージョンリリースの平均間隔の分布
(b) メジャーアップデート率の分布
(c) 依存パッケージ数の分布
(d) 被依存パッケージ数の分布

図 4: 属性分布の調査結果

バージョンリリースの平均間隔はパッケージ属性の update interval を決定するための指標である．本調査では，パッケージごとに前回のリリースからの経過日数の平均値を算出した．メジャーアップデート率はパッケージ属性の instability を決定するための指標である．本調査では，バージョン番号の 1 桁目の変更をメジャーアップデートとみなし，パッケージごとに全リリースにおけるメジャーアップデートの割合を算出した．依存パッケージ数は，パッケージ属性 dependencies を決定するための指標である．本調査では，パッケージごとに extra を含まない必須依存パッケージ数の平均値を算出した．被依存パッケージ数は，パッケージ属性 dependents を決定するための指標である．本調査では，調査期間内にリリースされた全パッケージの依存関係をもとに，各パッケージを必須の依存先として宣言している他パッケージの総数を算出した．

4.3.2 調査結果とパラメタ設定

調査の結果を 図 4 に示す．図 4 のグラフはいずれも横軸がそれぞれの指標の階級を表し，縦軸はその階級に属するパッケージの全体に対する割合を示している．図 4(a) はバージョンリリースの平均間隔の分布である．分布が集中している 0 日から 100 日までの範囲を表示している．0 日から 5 日の間隔を表す 1 番左の階級が最も高く，その割合は全体の約 30% を占めている．以降，間隔が大きくなるに

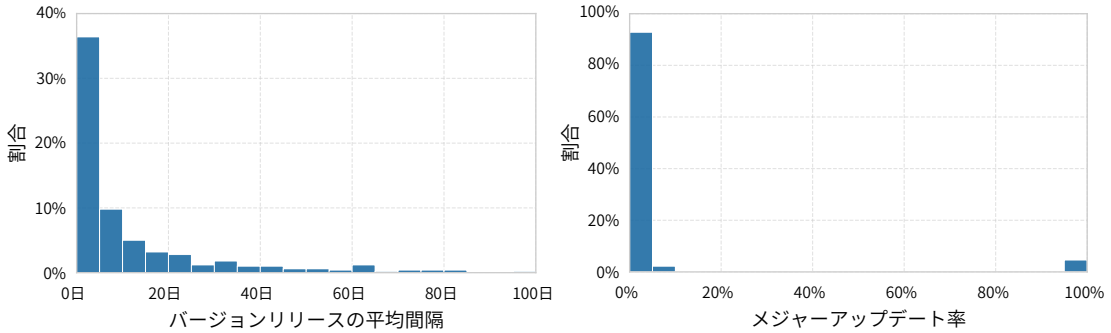
つれ割合は指数関数的に減少している．なお，図中の積算は約 60% であり，図の範囲外の 40% のうち 10% は 100 日以上の間隔を持つパッケージ，30% は調査期間内のリリースが 1 回のみのパッケージであった．

次に，図 4(b) はメジャーアップデート率の分布である．0% から 5% を表す 1 番左の階級が全体の約 90% を占めており，大部分のパッケージにおいてメジャーアップデート率が極めて低いことが読み取れる．一方で，Maven や Go 言語のエコシステムを対象とした実証研究では，マイナーアップデートやパッチアップデートといった本来互換性を保つべき更新において，20〜30% 程度の割合で互換性破壊が発生していたことが報告されている [4][6]．そこで，シミュレーションではメジャーアップデートだけではなく，マイナーアップデートとパッチアップデートであっても 30% の確率で互換性破壊が発生するものとした．

続いて，図 4(c) は依存パッケージ数の分布である．分布が集中している 0 個から 25 個までの範囲を表示している．0 個を表す 1 番左の階級が約 25% と最も高く，10 個以下のパッケージで全体の約 90% を占める結果となった．

最後に，図 4(d) は被依存パッケージ数の分布である．図 4(c) 同様に 0 個から 25 個までの範囲を表示している．0 個を表す 1 番左の階級が最も高く，全体の 80% 以上を占めていた．一方で，100 個以上と非常に多くのパッケージに依存されるパッケージも存在しており，被依存数の最大値は約 70,000 個に達していた．

本シミュレーションにおける各属性値の生成にあたっては，調査結果の分布を再現する確率モデルを採用した．具体的には，各指標がいずれもロングテールな分布であることを踏まえ，対数正規分布に従うと仮定した．パラメタの決定にあたっては，実データをもとに算出した平均値と分散を初期値とし，生成される分布が 図 4 にある調査結果のヒストグラムに適合するよう微調整を行った．パッケージ間の依存関係の構築においては，指定された依存数と被依存数を厳密に満たす依存関係グラフの生成は困難であるため，各パッケージに割り当てられた被依存数を重みとして扱い，依存先をランダムに決定する手法を採用した．生成されたパッケージ群の属性分布の一例を 図 5 に示す．図 4 に近い分布が生成できていることが確認できる．ただし，図 5(d) に示す被依存数は，生成時に割り当てられた重みそのものではなく，構築された依存関係グラフにおいて実際に計測された被依存数である．また，図 5(a) 内の積算は約 65% であり，図の範囲外の 35% のパッケージのバージョンリリースは最初の 1 回のみであった．一方で，ユーザの属性分布に関しては実データの調査を行えなかったため，ユーザの属性分布についても調査結果と同一の分布に従うと仮定して設定を行った．



(a) 生成モデルのバージョンリリース間隔の分布　(b) 生成モデルのメジャーアップデート率の分布

(c) 生成モデルの依存パッケージ数の分布　(d) 生成モデルの被依存パッケージ数の分布

図 5: 生成モデルの属性分布

4.4　シミュレーションの流れ

4.2 節で定義したモデルに基づき，シミュレーションはシナリオ生成とイベント実行の 2 段階から構成される．図 6 にシミュレーション全体の流れを示す．シミュレーションは始めにシナリオ生成を行う．シナリオ生成では，まずアクターを生成する．各アクターには，4.3 節で決定した分布にしたがって属性値を割り当て，dependencies 及び dependents 属性をもとに依存関係を構築する．次に，update interval 及び resolve interval 属性をもとに各アクターがいつ，どのイベントを行うのかを決定する．最後に，生成された各パッケージが提案手法を採用するかどうか，各ユーザがどの解決手法を用いるかを決定する．なお，アクター及びイベントの生成はこれ以降に新たに生成されることはなく，新規アクターの参入は考慮しない．

シナリオ生成が終わると，続いてイベント実行を行う．シナリオ生成で作成されたイベントキューからイベントを 1 つずつ取り出し，その種類に応じた処理を時系列順に実行する．各種類における処理内容は以下の通りである．

- パッケージ更新：図 6 の青色矢印に該当するイベントであり，リポジトリ上のバージョン情報を更新する．更新したパッケージが提案手法を採用している場合は，新しいリリースの依存先パッ

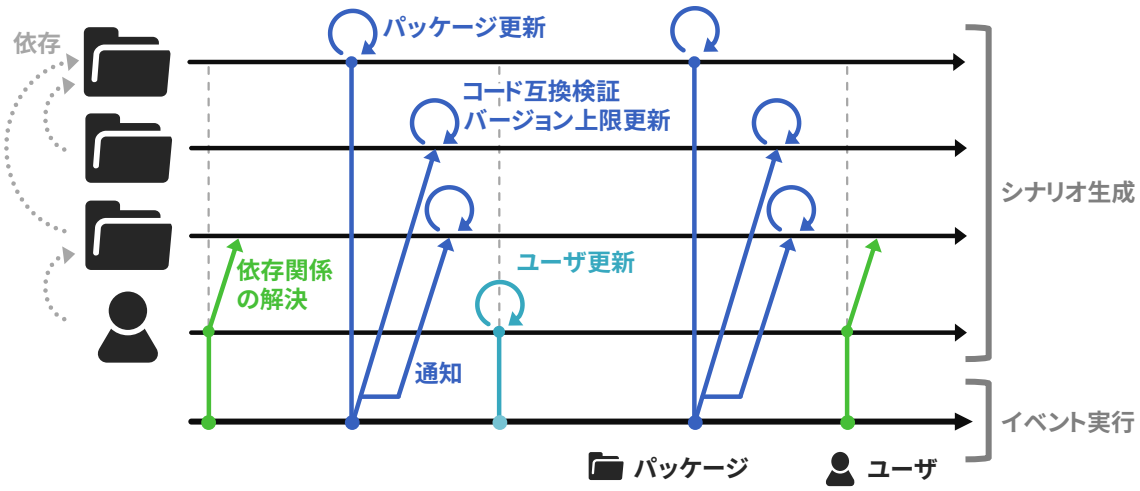


図 6: シミュレーションの流れ

ケージとコード互換検証を行いバージョン上限を設定する．また，更新したパッケージに依存しているパッケージが提案手法を採用している場合は，新しいリリースとのコード互換を検証し結果に応じてバージョン上限を更新する．

- ユーザ更新：図 6 の水色矢印に該当するイベントであり，ユーザが持つ依存関係を更新する．リポジトリ上の最新バージョン情報を参照し，自身の依存パッケージのバージョン要求をその時点での最新バージョンに更新する．
- 依存関係の解決：図 6 の緑色矢印に該当するイベントであり，シナリオ生成で割り当てられた解決手法を用いて解を探索する．

以上の処理をイベントキューが空になるまで繰り返す．

5　シミュレーションによる提案手法の評価実験

5.1　実験の概要

提案手法が互換性破壊の低減と計算コストの削減において有効であるかを確認するために，4 節で設計したシミュレーションモデルを用いて以下の 2 つの実験を行う．

実験 1：既存手法との比較評価

実験 2：提案手法の採用過渡期の評価

実験 1 では，提案手法及び既存手法がそれぞれ Python エコシステム全体で完全に採用された理想的な環境を想定する．ここでは，いずれの手法も一切採用されていない現状の Python エコシステムを模した従来手法を基準と定め，各手法の有効性を検証し比較する．実験 2 では，提案手法の採用が部分的である過渡期を想定する．ここでは，提案手法の採用率による有効性の推移を検証する．

これらの実験において，依存関係の解決結果の割合と計算コストを評価指標として設定した．依存関係の解決結果は得られた解をコード互換検証することで以下の 3 つに分類する．

- 解決成功：依存関係の解決に成功し，その解はコード互換である場合
- 互換性破壊：依存関係の解決には成功したが，その解はコード互換でない場合
- 解決失敗：バージョン要求を満たす解が存在しない，あるいは探索における再帰回数や深さが上限に達し解決が打ち切られた場合

これら 3 つの分類のうち，解決成功と解決失敗はパッケージマネージャが正しい解決結果を出力した場合である．一方で，互換性破壊は実際には互換性がないにも関わらずパッケージマネージャが誤って成功したと出力した場合である．本実験では，この誤った結果である互換性破壊をいかに低減できるかを評価の焦点とする．また，計算コストはパッケージまたはユーザが各手法においてコード互換検証を行った回数で表現する．コード互換検証 1 回に要する計算コストはパッケージ規模によって異なるが，本実験ではこの差異を無視し 1 回あたりの計算コストを均一であると仮定する．

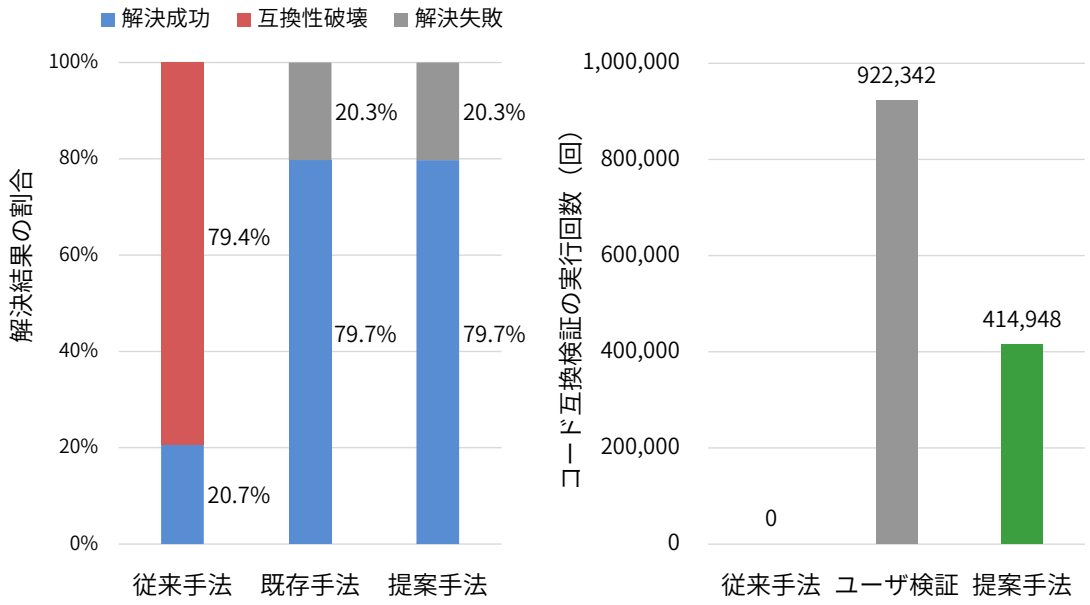
なお，本実験ではシミュレーションに要する時間の観点から，パッケージ数を 500，ユーザ数を 1000，期間を 100 日に設定した Python エコシステムの縮小モデルを用いて評価を行う．

5.2　実験 1：既存手法との比較評価

5.2.1　実験設定

本実験では，提案手法及び既存手法がそれぞれ Python エコシステム全体で完全に採用された理想的な環境を想定する．比較のために設定した各シナリオは以下の 3 つである．

1 つ目は従来手法である．これは提案手法及び既存手法はいずれも一切採用されていない現状の



(a) 各手法における依存関係の解決結果の割合　(b) 各手法におけるコード互換検証の実行回数

図 7: 実験 1 のシミュレーション結果

Python エコシステムを模したシナリオである．本シナリオでは，ユーザは依存関係の解決の際に一般的なパッケージマネージャと同様にバージョン互換のみを検証し解を探索する．この際，コード互換の検証は一切行われない．

2 つ目は既存手法である．これは PCREQ のようなバージョン互換とコード互換を検証する依存関係の解決手法が普及した場合の Python エコシステムを模したシナリオである．本シナリオでは，ユーザは依存関係の解決を行う際，解の候補となるバージョンに対して都度 API 比較を行い，コード互換性を検証しながら解を探索する．

3 つ目は提案手法である．これは本研究で提案するバージョン上限の継続的更新が普及した場合の Python エコシステムを模したシナリオである．本シナリオでは，パッケージは依存先パッケージの更新に伴ってコード互換の検証を行い，結果に応じてバージョン上限を更新する．ユーザは従来手法と同様にバージョン互換のみを検証し解を探索する．この際，ユーザ側でのコード互換の検証は一切行われない．

5.2.2　実験結果

依存関係の解決結果の割合：依存関係の解決結果の割合を 図 7(a) に示す．まず，互換性破壊の発生率に着目する．従来手法では全解決結果のうち 79.4% が互換性破壊に分類された．一方，既存手法と提案手法では互換性破壊の発生は確認されなかった（0.0%）．既存手法は解決時にすべての解の候補に対

```
{
  "name": "user-732",
  "requirements": [{
      "name": "package-47",
      "specifier": "==2.0.0",
      "used-apis": ...
    },
    {
      "name": "package-48",
      "specifier": "==2.2.0",
      "used-apis": ...
  }]
}
```

(a) ユーザの依存関係

```
{
  "name": "package-47",
  "version": "2.0.0",
  "provide-apis": ...,
  "requirements": [{
    "name": "package-48",
    "specifier": "<=1.1.0,>=1.0.0",
    "used-apis": ...
  }]
}
```

(b) 解決時のバージョン情報

図 8: 依存失敗に分類された事例

してコード互換の検証を行うため，互換性破壊が発生しないのは原理的に保証されている．ここで重要な点は，提案手法が既存手法と同等の互換性破壊の低減効果を示した点である．

次に，解決失敗の発生率に着目する．従来手法では解決失敗が 0.0% であるのに対し，既存手法と提案手法では 20.3% と増加する結果となった．解決失敗に分類された事例はいずれも従来手法において互換性破壊に分類されており，その内訳は 93.2% が解が存在しない事例，残り 6.8% が探索が打ち切られた事例であった．解決失敗の大半を占める解が存在しない事例の一例を 図 8 に示す．図 8(a) は解決失敗となったユーザの依存関係，図 8(b) は解決時点における `package-47==2.0.0` のメタデータである．図 8(b) にあるように `package-47` は `package-48` に依存しているが，提案手法によってコード互換な上限が設定されている．一方，図 8(a) にあるようにユーザは `package-47==2.0.0` と `package-48==2.2.0` を要求している．`package-47==2.0.0` は `package-48<=1.1.0,>=1.0.0` を要求しているため，両者を同時に満たすことはできない．その結果，パッケージマネージャは解決失敗と出力した．もし提案手法が採用されておらず上限が存在しなかった場合，コード互換な解は存在しないにも関わらずパッケージマネージャは誤って解決成功と出力してしまっていた．また，探索が打ち切られた事例の一部に対し探索回数の上限を緩和して再度解決を試みたところ，いずれも実際には解が存在しない事例であることが確認できた．提案手法は上限を設定することで互換性のないバージョンを除外し探索範囲を小さくするが，これは依存関係の解決において制約条件が厳格化されたとも捉えられる．その結果，バックトラックの回数が増加し，制限内で探索が終了しなかったと考えられる．以上のことから，既存手法と提案手法による解決失敗の増加は従来手法では検出できない互換性破壊を正しく判断

できた結果であるといえる．しかしその一方で，厳格な制約は依存関係の解決に要する計算コストを増加させ，探索の打ち切りによって一部の有効な解を見逃す可能性も生じさせている．

コード互換検証の実行回数：コード互換検証の実行回数を 図 7(b) に示す．従来手法における検証回数は 0 回と最小であるが，前述の通り互換性破壊の発生率は非常に高く単に検証を行っていないだけである．そこで，互換性破壊の発生率が 0.0% と安全性を確保できている既存手法と提案手法の 2 つを比較する．既存手法では検証回数が約 92 万回であったのに対し，提案手法は約 41 万回となり約 45.0% にまで削減された．提案手法のコード互換な解を共有するという設計によって，検証の重複を防止し計算コストの削減に対して有効であることが確認できた．

5.3 実験 2：提案手法の採用過渡期の評価

5.3.1 実験設定

実験 1 ではすべてのパッケージが提案手法を採用した理想的な環境を想定した．しかし，現実のソフトウェア開発において新しい手法が優れていたとしてもすべての開発者に遵守されることは稀であり，その普及は一部のパッケージに留まることが一般的である．例えば，セマンティックバージョニングはバージョン番号の規則のみで互換性を保証する手法であり，開発者が正しく運用を行えば低コストで互換性破壊を低減できる．しかし，現実のソフトウェアエコシステムにおいてすべてのパッケージ開発者がこれを遵守しているわけではない．そこで本実験では，提案手法の採用が部分的である過渡期を想定し，提案手法の採用率の変化に伴う有効性の推移を検証する．具体的には，提案手法の採用率を 0% から 100% まで 10% 刻みで変化させ，互換性破壊の発生率の推移を確認する．パッケージ開発者の自主性に任せ採用が広まっていく状況を想定し，提案手法を採用するパッケージはランダムに選出する．採用率を増加させる際は前段階で採用したパッケージは引き続き提案手法を採用するものとする．

5.3.2 実験結果

提案手法を採用するパッケージをランダムに選択した場合の採用率の変化に伴う互換性破壊の発生率の推移を 図 9 に示す．本実験では，提案手法を採用するパッケージの組み合わせによる結果の変動を考慮し，同シナリオに対し提案手法を採用するパッケージを変化させ計 5 回シミュレーションを行った． 図 9 の 5 つの折れ線グラフは，それぞれの試行における推移を表している．全体的な傾向として採用率の増加に伴って互換性破壊の発生率は低減していることが確認できる．しかし，提案手法の採用率が 40% 以下の段階では互換性破壊の発生率は 40%〜80% と高く，大きな改善は見られなかった．更に，採用率が 90% と高い水準に達してもなお，大半の試行において互換性破壊の発生率は 40% に留まり，当初の 80% から半分程度しか低減できていない結果となった．

また，一部の試行で採用率が増加したにも関わらず，互換性破壊の発生率が増加する現象が確認さ

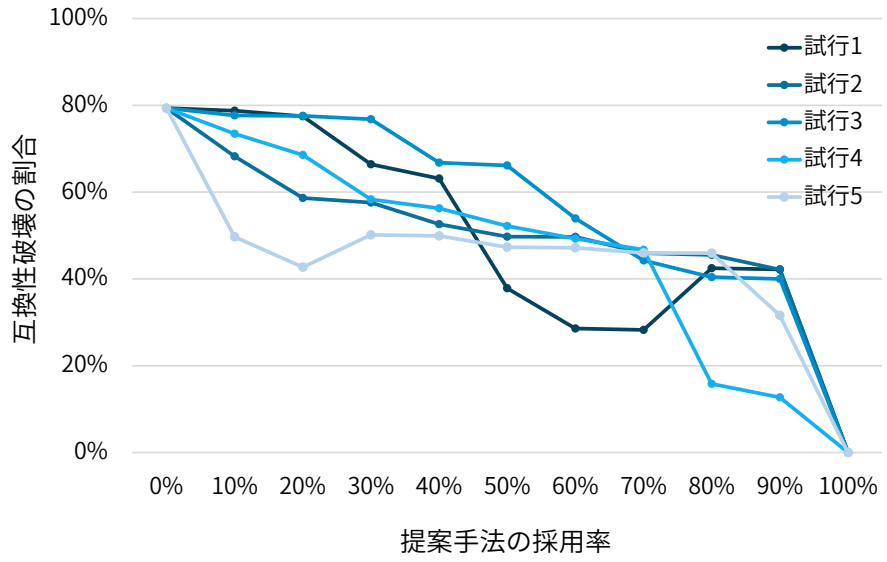


図 9: ランダム採用における互換性破壊の発生率の推移

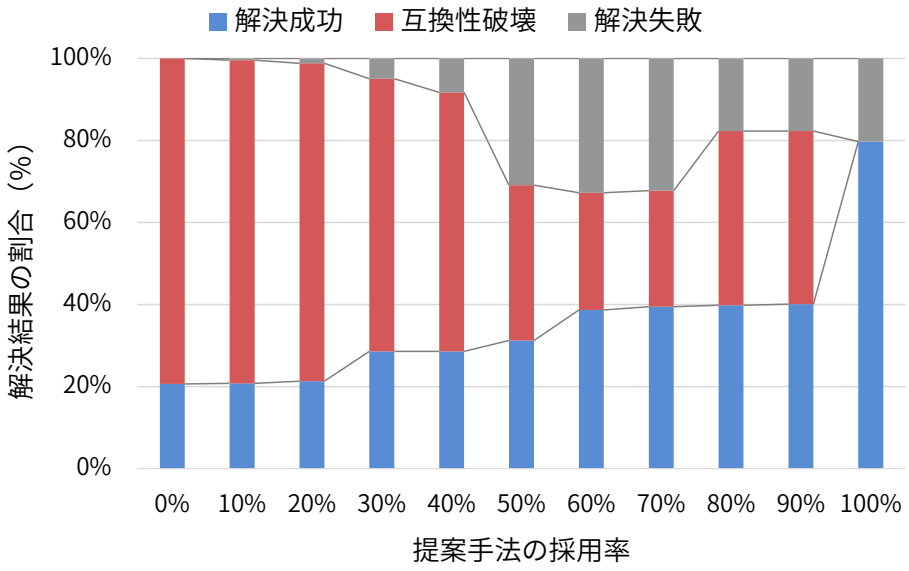


図 10: 試行 1 における解決結果の割合の推移

れた．例えば，試行 1 において採用率を 70% から 80% に増加させた場合に，互換性破壊の発生率が 30% 程度から 40% 程度まで増加していることが確認できる．この試行 1 における解決結果の割合の推移を 図 10 に示す．提案手法の採用率を 40% から 50% に増加する段階では互換性破壊が大きく減少する一方で解決失敗が増加しており，逆に 70% から 80% に増加する段階では互換性破壊が増加する一方で解決失敗が減少していることが確認できる．個別の事例を調査した結果，採用率 50% の段階で互換

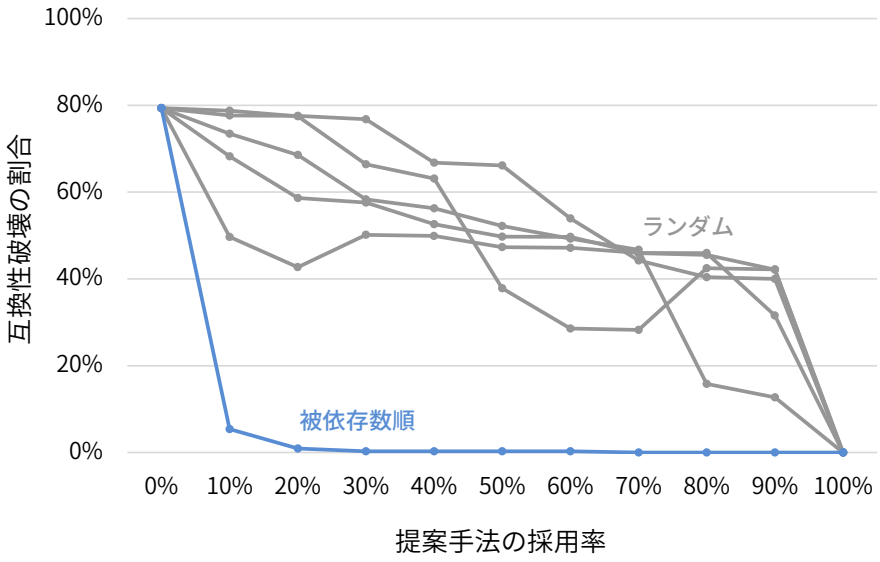


図 11: 被依存数順採用における互換性破壊の発生率の推移

性破壊から探索打ち切りが原因で解決失敗に分類された事例が多く発生しており，またそのすべてが採用率 80% の段階で再度互換性破壊に分類されていた．これは採用率 50〜70% の段階では提案手法による制約条件の厳格化が原因で探索が打ち切られていたが，採用率が高まり 80% の段階では探索空間が適切に縮小されたことで探索が終了するようになり，本来の分類結果である互換性破壊に戻ったためであると考えられる．以上のことから，採用率の増加が直接的に互換性破壊を引き起こしたわけではないといえる．しかしその一方で，ランダムに採用が進む場合その過渡期においては計算コストの増加による探索打ち切りの発生が懸念される．

一方で，試行によって互換性破壊の発生率の推移は大きく異なっていた．例えば，図 9 で互換性破壊の発生率を 50% にまで低減させるために要した採用率を比較すると，試行 5 ではわずか 10% であるのに対し試行 4 では 70% も必要であった．この結果から，提案手法の有効性は単なる採用率ではなく，どのパッケージが採用するのかによって大きく左右されるといえる．そこで，被依存数の多いパッケージほど提案手法を採用した際の互換性破壊の低減効果が高いと推測した．被依存数の多いパッケージはエコシステム内での影響力が強く，その更新が依存関係を通じて広範囲に波及するためである．この推測の妥当性を検証するため，被依存数が多いパッケージから優先的に提案手法を採用させた場合のシミュレーションを行った．なお，採用順序の決定においては 1 つ以上の依存先を持つという条件を前提とした．これは，依存先を持たないパッケージは原理的に他パッケージの更新の影響を受けず，提案手法の恩恵を受けないためである．そのシミュレーション結果を 図 11 に示す．灰色のグラフは 図 9 と同様の 5 つの試行であり，青色のグラフが 1 つ以上の依存先を持ち被依存数が多いパッケージから優先的に提案手法を採用させた場合のグラフである．採用率がわずか 10% の段階で互換性破壊の発生率を 5% 程

度まで低減させており，20% の段階ではほぼ 0% に達していることが確認できる．したがって，提案手法の普及においては単に普及率を高めるだけではなく，被依存数が多くエコシステム内で影響力の大きいパッケージが優先的に採用することで，採用率が低い過渡期においても高い効果を発揮できることが確認できた．

6　今後の課題

6.1　シミュレーションの改善

本研究では，提案手法の評価に Python エコシステムのシミュレーションを用いた．しかし，シミュレーションモデルの構築にあたり現実の Python エコシステムを複数の観点で抽象化しているため，実環境との乖離が課題として残されている．主な改善点として，以下の 3 つが挙げられる．

1 つ目はアクター属性である．本モデルでは，4.2.4 節で述べたように多種多様なアクターを表現するため，パッケージ属性は 4 つ，ユーザ属性は 3 つ定義した．しかし，これらのみで現実の多様なアクターを表現するには限界があり，属性の追加が必要であると考えられる．例えば，パッケージ属性として提供する API 数（パッケージ規模）が挙げられる．現実にはパッケージが提供する API をすべて利用すること稀であり，API 数が多い大規模なパッケージほど破壊的変更が含まれていても自身の利用箇所には影響がないという可能性が高くなる．

2 つ目はイベントの生成ルールである．本モデルでは，すべてのパッケージやユーザが割り当てられた属性値に基づいて一定確率で行動すると仮定した．しかし，現実の開発者は状況に応じた行動を取る．例えば，互換性破壊の問題に直面した開発者は短期間のうちに依存関係やソースコードを修正した更新を行うと考えられる．また，提案手法の採用についても運用コストの観点から採用を中止することも想定できる．

3 つ目は計算コストの算出である．本実験では，計算コストをコード互換検証の回数として定義し，検証に要する計算コストは均一と仮定した．しかし，実際には API が多く複雑なパッケージと小規模なパッケージでは静的解析やテストに要する計算コストは異なると考えられる．また，エンドツーエンドな既存手法とは異なり，提案手法は Python エコシステム全体での動的な連携を前提としている．そのため，検証そのもののコストに加え，パッケージ更新通知の送受信やバージョン上限更新に伴うメタデータの更新といったシステム間の連携に要するコストも考慮する必要があると考える．

6.2　実環境への適用

提案手法を実際に Python エコシステムに適用するために，解決すべき課題として以下の 2 つが挙げられる．1 つ目はパッケージ更新の通知機能とバージョン上限の更新機能の実現である．提案手法では，あるパッケージが更新した場合にそのパッケージに依存するパッケージに対して通知を送信する必要がある．また，通知を受け取ったパッケージがバージョン上限の更新を行う場合にバージョンのメタデータを更新する必要がある．しかし，現在の PyPI にはパッケージ更新の通知機能は存在せず，一度リリースしたバージョンのメタデータを更新することはできない．もし現在の仕様のままバージョン上限を更新しようとすれば，ソースコードに変更がなくても新たなバージョン番号での再リリースが必要

となりバージョン管理が煩雑になる．したがって，提案手法を実現するためには，バージョン更新を伴わずに依存関係情報のみを動的に変更できる仕組みが必要である．

2 つ目は提案手法の普及である．セマンティックバージョニングのような規則の遵守やエンドツーエンドな既存手法とは異なり，提案手法は互換性検証や依存関係更新といった継続的な運用コストをパッケージ開発者に要求する手法である．また，コストを負担する側とその恩恵を受ける側が異なっており，開発者自身は提案手法採用による直接的なメリットを得づらい構造となっている．したがって，提案手法の普及に向けた仕組みや取り組みを検討していく必要がある．

7　おわりに

本研究の目的は，Python エコシステムにおける潜在的な互換性破壊の低減とコード互換な解発見に要する計算コストの削減である．そのために，バージョン上限の宣言及び上限の継続的更新という手法を提案した．シミュレーションによる評価の結果，提案手法は従来手法では検出できない互換性破壊の発生を完全に防止し，既存手法と比較してコード互換検証の実行回数を約 45% 削減しており，提案手法の有効性を確認できた．また，被依存数が多いパッケージが優先的に提案手法を採用することで，過渡期においても有効であることも確認できた．

今後取り組むべき課題として，以下の 2 つを挙げる．1 つ目はシミュレーションの改善である．具体的には，アクター属性やイベント生成ルールの追加を行うことで，より現実に即したシミュレーションモデルの構築が期待できる．2 つ目は実環境への適用である．本稿では提案手法の有効性については評価したが，実際の開発環境や PyPI への導入については十分に議論ができていない．そのため，PyPI 側の機能拡張や普及に向けた仕組みについて検討していく必要がある．

謝辞

本研究を遂行するにあたり，多くの方々にご指導とご支援を賜りました．

楠本真二教授には中間報告会で多くのご助言をいただきました．客観的な視点からご指摘をいただき，今一度自身の考えを整理するきっかけとなりました．深く感謝申し上げます．

柗本真佑准教授には本研究のテーマ決めから論文執筆，そして研究会での発表に至るまで，研究過程のすべてにおいて多くのご指導をいただきました．この 1 年間は私にとって非常に多くの学びを得る機会となりました．心から感謝申し上げます．

事務補佐員の橋本美砂子様には出張手続きや研究室の機材購入など，研究生活を送る中で多岐にわたるご支援をいただきました．心よりお礼申し上げます．

楠本研究室の先輩方には研究活動や研究室での生活に至るまで，多くの面でお世話になりました．また，研究室内でのイベントは良い気分転換になり，研究室では楽しい時間を過ごすことができました．心からお礼申し上げます．

楠本研究室の同期の皆様とは研究の相談や休憩中の雑談など，研究生活での大きな心の支えとなってくれました．本当にありがとうございます．

そしてここまで大学生活を無事に送ることができたのは，経済面のみならず精神面でも支えてくれた家族のおかげです．本当にありがとうございます．

最後に，本研究を支えてくださったすべての方々に改めて感謝申し上げます．

参考文献

[1] Jayasuriya, D., Terragni, V., Dietrich, J. and Blincoe, K.: Understanding the Impact of APIs Behavioral Breaking Changes on Client Applications, *Proceedings of the ACM on Software Engineering*, Vol. 1, No. FSE, pp. 1238–1261 (2024).

[2] Peng, Y., Hu, R., Wang, R., Gao, C., Li, S. and Lyu, M. R.: Less is More? An Empirical Study on Configuration Issues in Python PyPI Ecosystem, in *Proceedings of International Conference on Software Engineering*, pp. 2494–2505 (2024).

[3] Lei, H., Xiao, G., Liu, Y. and Zheng, Z.: PCREQ: Automated Inference of Compatible Requirements for Python Third-party Library Upgrades (2025).

[4] Ochoa, L., Degueule, T., Falleri, J.-R. and Vinju, J.: Breaking bad? semantic versioning and impact of breaking changes in maven central: An external and differentiated replication study, *Journal on Empirical Software Engineering*, Vol. 27, No. 61 (2022).

[5] Venturini, D., Cogo, F. R., Polato, I., Gerosa, M. A. and Wiese, I. S.: I Depended on You and You Broke Me: An Empirical Study of Manifesting Breaking Changes in Client Packages, *Transactions on Software Engineering and Methodology*, Vol. 32, No. 4, pp. 1–26 (2023).

[6] Li, W., Wu, F., Fu, C. and Zhou, F.: A large-scale empirical study on semantic versioning in golang ecosystem, in *Proceedings of International Conference on Automated Software Engineering*, pp. 1604–1614 (2023).

[7] Wang, C., Wu, R., Song, H., Shu, J. and Li, G.: smartpip: A Smart Approach to Resolving Python Dependency Conflict Issues, in *Proceedings of International Conference on Automated Software Engineering*, pp. 1–12 (2023).

[8] Wang, Y., Wen, M., Liu, Y., Wang, Y., Li, Z., Wang, C., Yu, H., Cheung, S.-C., Xu, C. and Zhu, Z.: Watchman: Monitoring Dependency Conflicts for Python Library Ecosystem, in *Proceedings of International Conference on Software Engineering*, pp. 125–135 (2020).

[9] Wang, H., Liu, S., Zhang, L. and Xu, C.: Automatically Resolving Dependency-Conflict Building Failures via Behavior-Consistent Loosening of Library Version Constraints, in *Proceedings of Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 198–210 (2023).

[10] Cheng, W., Zhu, X. and Hu, W.: Conflict-aware inference of python compatible runtime environments with domain knowledge graph, in *Proceedings of International Conference on Software Engineering*, pp. 451–461 (2022).

[11] Ye, H., Chen, W., Dou, W., Wu, G. and Wei, J.: Knowledge-based environment dependency inference for python programs, in *Proceedings of International Conference on Software Engineering*, pp. 1245–1256 (2022).

[12] Cheng, W., Hu, W. and Ma, X.: Revisiting Knowledge-Based Inference of Python Runtime Environments: A Realistic and Adaptive Approach, *Transactions on Software Engineering*, Vol. 50, No. 2, pp. 258–279 (2024).

[13] Dietrich, J., Pearce, D., Stringer, J., Tahir, A. and Blincoe, K.: Dependency Versioning in the Wild, in *Proceedings of International Conference on Mining Software Repositories*, pp. 349–359 (2019).