一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

依存関係の解決におけるバージョン上限の継続的更新に向けて —Python エコシステムのシミュレーションによる評価—

三倉　孝太† 　柗本　真佑† 　楠本　真二†

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

あらまし　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，宣言時点では予知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コンセプトを提案する．互換性破壊を生み出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．本稿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に必要な要件を洗い出す．

キーワード　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

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．2025 年 10 月現在では pandas の最新バージョンは 2.3.3 であるため，“pandas==2.3.3” が依存関係の解決の結果となる．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．コード非互換な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削減できる．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 エコシステム全体でのコード互換問題を低減する依存宣言の実現である．そのために，本稿ではコード互換検証によって発見されるバージョン互換かつコード互換な要求の表現方法と更新方法について考える．解の表現方法としてはバージョン上限というアイデアを考える．バージョン互換かつコード互換な最新のバージョン点は互換性を満たす上限とみなせる．また解の更新方法としては継続的更新

というコンセプトを提案する．これは一種の Observer パターンであり，Python エコシステム全体でのパッケージ更新と要求更新を繰り返す．更に本稿では，提案手法の効果を確かめるためのシミュレーションの方針について議論する．

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，2025 年 10 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]．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調査やその影響を調査した研究が複数存在する [4]〜[6]．一般に依存関係の解決プロセスが保証するのは，あくまでバージョン要求を満たすバージョン互換のみであり，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については一切保証しない．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確認している [7]．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 パラメー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

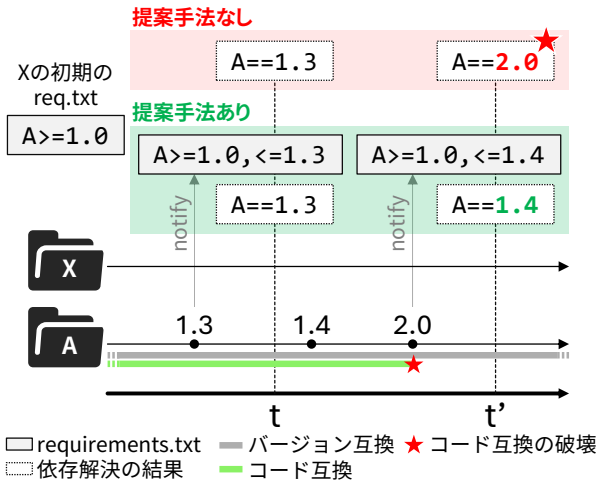

図 1: 推移的要求宣言の適用例

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．

3. 推移的要求宣言

3.1 概　　要

本研究の目的は，Python エコシステム全体で潜在的な互換性破壊を低減する依存要求宣言の実現である．そのために，時間的な推移に伴って要求を更新する，推移的要求宣言というコンセプトを提案する． 図 1 にその適用例を示す．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 t では A の破壊的な更新は含まれておらず，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 t' 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，推移的要求宣言を適用した場合では 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 の破壊的な更新は含まれていないため，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 t 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,<=1.4” とする．そのため，時刻 t' 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==1.4” が解として発見される．以降では，推移的要求宣言を実現するための 2 つのサブコンセプト，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する範囲として表現し直す．例えば，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

バージョン上限は固定的な情報ではなくパッケージ更新に伴う時間的な推移に伴って変化する情報である．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保つ場合は依存するパッケージの上限をそのバージョンに更新し，互換性を破壊する場合は依存するパッケージの上限を更新しない．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推移的要求宣言を適

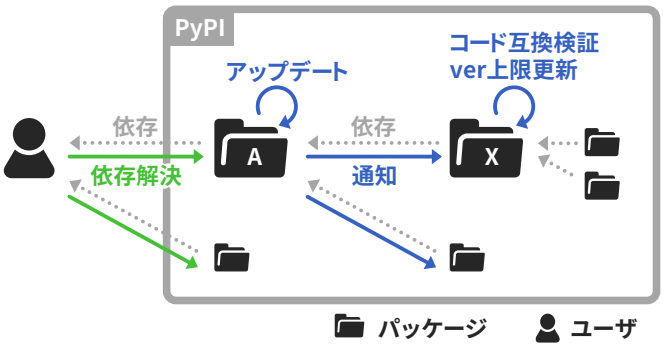

図 2: アクターとイベントの関係

```
{
   "updated-at" : "0",
   "provide-apis" : ["A", "A", "B"],
   "dependencies" : [{
     "name" : "dependency-package",
     "specifier" : ">=2.0",
     "used-apis" : ["A", "B", ""]
  }]
}
```

図 3: パッケージのデータモデル

```
{
   "declared-at" : "10",
   "dependencies" : [{
     "name" : "dependency-package",
     "specifier" : ">=1.0",
     "used-apis" : ["A", "", "A"]
    }]
}
```

図 4: ユーザのデータモデル

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．

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 節で提案した手法を Python エコシステム全体に適用した場合のコード互換性破壊の低減，及びそれに伴う計算コストの削減の確認である．コード互換性破壊は時間的推移に伴う問題であり，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，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し，評価の観点を説明する．続いて 4.3 節で，具体的なシミュレーションの実行手順を説明する．最後に 4.4 節で，シミュレーションに用いるパラメータの決定方法について述べる．

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，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，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．そのバージョンが持つメタデータを持つ．このメタデータは 図 3 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を表し，provide-apis はそのパッケージが提供する API を表し，dependencies はそのパッケージの依存関係を表す．また，このメタデータはある瞬間におけるバージョンの状態を表しており，提案手法を適用したパッケージの場合，バージョン上限の更新に伴い時間的に変化する．

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 4 に示す JSON 形式で表現する．declared-at は依存関係を宣言した時刻を表し，dependencies は宣言された依存関係を表す．時刻は日付ではなく，シミュレーションの実行ステップ数を表している． 図 4 の例は，シミュレーション開始から 10 ステップ目の状態を表している．また，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）から構成される．4.2.3 節で後述するが，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．

4.2.2 イベントの定義

本モデルでは，エコシステム内で発生する主要な出来事として，パッケージの更新と依存関係の解決の 2 種類のイベントを定義する．

図 2 の青矢印で示すパッケージの更新は，あるパッケージ A が新しいバージョンをリリースするイベントである．このイベントが発生すると，まずパッケージ A に直接依存するパッケージ X に更新を通知する．次に，通知を受け取ったパッケージ X はその新しいバージョンがコード互換を満たすかどうかを検証する．最後に，コード互換を満たす場合はその新しいバージョンを上限として更新し，満たさない場合は上限を更新しない．

図 2 の緑矢印で示す依存関係の解決は，あるユーザが自プ

ロジェクトの依存関係を解決するイベントである．このイベントが発生すると，自プロジェクトが宣言する依存関係のバージョン要求を満たす最新のバージョンが選択される．この処理は pip が行う依存関係の解決と同様である．

4.2.3 コード互換検証の定義

コード互換検証は，3.2 節で述べたように多くの計算コストを要する．そこで，本シミュレーションでは，コード互換検証を単純化する．

具体的には，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する．パッケージが提供する複数の API は ["A", "A"] のようにリストとして表現する．リストの各インデックスは個々の API に対応している．パッケージの更新によりシグネチャが変更された API は"B"や"C"のように異なる文字に置き換える．また，API が新しく追加された場合は新しいインデックスに文字を割り当て，削除された場合はその API に対応するインデックスの文字を空文字（""）に置き換える．パッケージやユーザが持つプロジェクトで利用する API も同様にリストで表現する．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．パッケージやプロジェクトで利用する API には提供する API リスト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．ここで，2 つ目の API は利用していない（空文字）ため，シグネチャは一致しないがコード互換検証には影響しない．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", ""] 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 は削除されており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．

4.2.4 評価の観点

本シミュレーションは，以下の評価指標と比較シナリオを用いて実施する．

本シミュレーションにおける評価指標は，4.1 節で述べたコード互換性破壊の低減率と計算コストの 2 点である．

コード互換性破壊の低減率：ユーザが実行した依存関係の解決で発見された解がコード非互換であった割合で評価する．

計算コスト：シミュレーション全体で発生したコード互換検証の総回数で評価する．ただし，潜在的な互換性破壊率を評価するために必要なコード互換検証は除外し，コード互換検証 1 回は 1 つの依存パッケージの 1 つのバージョンに対して API の利用関係を検証することと定義する．

また，以下の 3 つのシナリオを設定し評価指標を比較する．

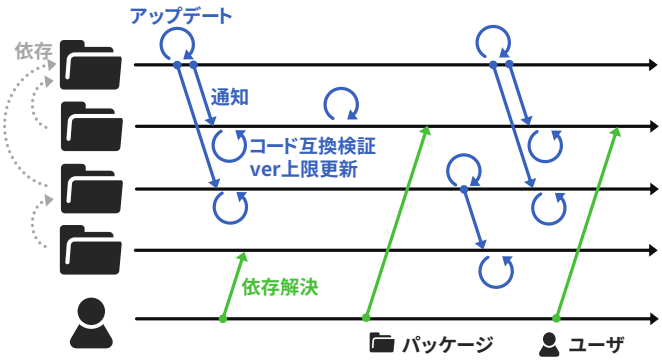

図 5: シミュレーションの流れ

シナリオ 1：パッケージは提案手法を適用せず，バージョン上限の指定や更新を行わない．ユーザは依存関係の解決時にバージョン互換のみを考慮し，コード互換を検証しない．

シナリオ 2：パッケージは提案手法を適用せず，バージョン上限の指定や更新を行わない．ユーザは依存関係の解決時にバージョン互換かつコード互換な解を見つけるためコード互換を検証する．このシナリオでは，コード互換を検証するユーザの割合をパラメータとして変化させ，適用率が評価指標に与える影響を分析する．

シナリオ 3：パッケージは提案手法を適用し，バージョン上限を指定し，継続的に更新する．ユーザは依存関係の解決時にバージョン互換のみを考慮し，コード互換を検証しない．このシナリオでは，提案手法を適用するパッケージの割合をパラメータとして変化させ，適用率が評価指標に与える影響を分析する．

4.2.5 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，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：

- Activity（活動性）：パッケージを更新する確率である．この値が高いほど短いステップで新バージョンを公開していることを表す．
- Stability（安定性）：新バージョンが後方互換性を保つ確率である．この値が高いほど互換性破壊を伴わない安定した更新を行っていることを表す．
- Dependencies（依存数）：依存するパッケージの数である．
- Scale（規模）：提供する API の数である．
- Dependants（被依存数）：依存されるパッケージの数である．

ユーザの属性：

- Activity（活動性）：依存関係の解決を行う確率である．この値が高いほど短いステップでプロジェクトの依存関係を解決していることを表す．
- Dependencies（依存数）：依存するパッケージの数である．

4.3 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初期化ステップとメインループから構成される． 図 5 にシミュレーション全体の流れを示す．左に示すアクターの生成が初期化ステップであり，右矢印で示す部分がメインループである．

シミュレーションはまず初期化ステップから開始される．このステップでは，シミュレーションに登場するすべてのパッケージとユーザを生成し，それぞれに属性値を割り当てる．アクターは初期化以降に生成されることはなく，新規パッケージや新規ユーザの参入は考慮しないものとする．また，属性値は現実の分布に基づき割り当てられる．これによって多種多様なパッケージとユーザの存在を表現する．

初期化が完了すると，シミュレーションはメインループに移行する．このループは規定のステップ数に達するまで繰り返され，各ステップではアクターの Activity に基づきイベントを確率的に発生させる．ここでの 1 ステップは秒や分といった単位ではなく，エコシステム内で何らかのイベントが発生するのに十分な数日といった比較的大きな時間間隔を表している．パッケージは毎ステップ自身の Activity に基づきパッケージの更新を行うかを決定する．新バージョンを公開する場合，更に Stability に基づきその更新が後方互換性を保つかを決定する．シナリオ 3 では更新したパッケージに直接依存するパッケージは，更新通知を受け取るとその更新がコード互換性を保つかを検証し，必要に応じてバージョン上限を更新する．シナリオ 3 では，依存パッケージが行うコード互換検証の回数を計算コストとしてカウントする．ユーザは毎ステップ自身の Activity に基づき依存関係の解決を行うかを決定する．シナリオ 1，3 では解決時にバージョン互換のみを考慮し，コード互換を検証しない．シナリオ 2 では解決時にバージョン互換かつコード互換な解を見つけるためコード互換を検証する．依存関係の解決で得られたバージョンの組み合わせがコード非互換であった場合，コード互換性破壊としてカウントし，シナリオ 2 ではユーザによるコード互換検証の回数を計算コストとしてカウントする．

4.4 パラメータの決定方法

本シミュレーションの妥当性は，アクターの総数や属性値の分布などのパラメータ設定に強く依存する．これらのパラメータは，Python エコシステムの実態を可能な限り反映するため，PyPI [8] と deps.dev [9] が提供する公開データセットを BigQuery 上で分析し，決定する方針である．

アクターの総数や Activity（活動性），Dependencies（依存数），Dependants（被依存数）といったパラメータの多くは，リリース履歴や依存関係のメタデータの分析から導出可能である．一方で，Stability（安定性）と Scale（規模）については公開データからの直接的な決定が困難であり，PyPI を対象とした実証研究やデータセットは本稿執筆時点では見当たらない．

Stability はパッケージの更新のうち，後方互換性を保つ更新の割合を表すパラメータである．Maven や Go 言語のエコシステムを対象とした実証研究では，マイナーバージョンアップデートなどの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]．しかし，PyPI ではセマンティックバージョニングだけでなくカレンダーバージョニングも広く採用されており [10]，これらの値を直接適用することの妥当性は不明である．事前調査を独自に行う場合にも，Python の動的な性質上静的解析による意味的な変更の検出は困難である．

また，Scale はパッケージが提供する API の数を表すパラメータである．Maven を対象とした実証研究では静的解析により API 総数を計測していたが，Python はアクセス修飾子が存在せず C/C++ 拡張も多いため，同様の解析は困難である．パッケージのファイル数やコード行数から API 数を推定する方法も考えられるが，厳密な値を得ることは難しい．

5. おわりに

本稿では，Python エコシステムにおける潜在的な互換性破壊の問題に対し，その解決策としてバージョン上限の宣言と継続的更新を提案した．更に，提案手法の有効性を評価するため，シミュレーションの設計と実験計画を立案した．

今後の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実装と評価である．本稿で設計したシミュレーションを実行し，提案手法が 2 つの評価指標に与える影響を分析する．2 つ目はシミュレーションに用いるパラメータ調査と決定である．特に，Stability と Scale の 2 つのパラメータは妥当性の高い値を決定するためにさらなる調査が必要である．

謝辞 本研究の一部は，JSPS 科研費（JP25K15056, JP25K03102, JP24H00692）による助成を受けた．

文　献

[1] D. Jayasuriya, V. Terragni, J. Dietrich, and K. Blincoe, “Understanding the impact of APIs behavioral breaking changes on client applications,” In Proceedings of the ACM on Software Engineering, vol.1, no.FSE, pp.1238–1261, 2024.

[2] Y. Peng, R. Hu, R. Wang, C. Gao, S. Li, and M.R. Lyu, “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] H. Lei, G. Xiao, Y. Liu, and Z. Zheng, “PCREQ: Automated inference of compatible requirements for Python third-party library upgrades,” 2025.

[4] L. Ochoa, T. Degueule, J.-R. Falleri, and J. Vinju, “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] D. Venturini, F.R. Cogo, I. Polato, M.A. Gerosa, and I.S. Wiese, “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] W. Li, F. Wu, C. Fu, and F. Zhou, “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] J. Dietrich, D. Pearce, J. Stringer, A. Tahir, and K. Blincoe, “Dependency versioning in the wild,” In Proceedings of International Conference on Mining Software Repositories, pp.349–359, 2019.

[8] Python Packaging Authority, “BigQuery Datasets,” `https://docs.pypi.org/api/bigquery/`, (Accessed at 2025-11-12).

[9] Google, “deps.dev BigQuery dataset,” `https://docs.deps.dev/bigquery/v1/`, (Accessed at 2025-11-12).

[10] Python Packaging Authority, “versioning,” `https://packaging.python.org/en/latest/discussions/versioning/`, (Accessed at 2025-11-12).