特別研究報告

題目

Parallel Setsによるライブラリの組み合わせと利用状況の可視化

指導教員
井上 克郎 教授

報告者
矢野裕貴

平成 27 年 2 月 13 日

大阪大学 基礎工学部 情報科学科

平成 26 年度 特別研究報告

Parallel Sets によるライブラリの組み合わせと利用状況の可視化

矢野裕貴

内容梗概ソフトウェア開発におけるコードの再利用手段として，オープンソースライブラリの利用が盛んに行われている．しかし，ライブラリはバージョン間で常に完全な互換性が保証されているわけではない．そのため，開発中のシステムにおいて依存ライブラリのアップデートや，新規ライブラリの導入を行う際に，開発者は正しく動作するようなバージョンの組み合わせを選択する必要がある．

本研究では，開発者の判断を支援することを目的としたライブラリの利用状況の可視化を提案する．約 4000 の Java オープンソースプロジェクトを対象に依存関係の収集を行い，ライブラリの利用状況のデータセットを構築した．そして，開発者が利用したいライブラリの組み合わせを入力すると，使用されているバージョンの組み合わせをデータセットから抽出し，Parallel Sets を用いて可視化するツールを作成した．開発者は既存の様々なシステムがライブラリを組み合わせた利用実績を確認することで，ライブラリの組み合わせによって発生する非互換性の問題のリスクの低減を図ることができる．

利用シナリオとして，HTTP クライアントアプリケーションにおけるライブラリの更新を想定した可視化を実施した．その結果，新しいライブラリを利用する場合のバージョンの選定や，既に使用しているライブラリの更新を行うための情報を，実際に可視化結果から読み取れることを確認した

主な用語

オープンソースライブラリ
Parallel Sets
ソフトウェア間の依存関係

目 次

1　まえがき

ソフトウェア開発におけるコードの再利用手段として，オープンソースの（OSS) ライブラリが広く一般的に利用されている．OSS ライブラリは主に GitHub[1]，Maven Central Repository[2] などのリポジトリから入手することができる．これによって様々な機能に関する高品質なコードが再利用できるようになっており，開発者自身が必要な機能を全て開発する場合に比べて大幅に開発コストを削減することができる．また，オープンソースであるため多くのユーザーによってテストされていて安全性が比較的高いというメリットがある．

ライブラリはそれに依存するソフトウェアとは独立して開発が進められており，バグの修正やリファクタリング，新機能の追加，重大な脆弱性の修正などの要因により新しいバージョンの公開が行われる．ライブラリを利用する開発者はソフトウェアのメンテナンスとして，開発中のプロジェクトが依存するライブラリにおいてバグや脆弱性が修正された場合には，バージョンの更新を行う必要がある．しかし，多くのプロジェクトにおいて，脆弱性を抱えたままのバージョンのライブラリが利用され続けている．[1]

ライブラリの更新が行われない 1 つの要因となっているのがライブラリのバージョン間の非互換性である．公開されているライブラリのあるバージョンとその次のバージョンとの間には，常に完全な互換性が保証されているわけではない．ライブラリのバージョン間での互換性が成り立っていない場合，依存ライブラリの更新を行うことによって開発中のプログラムが正しく動作しなくなってしまう可能性がある．また，OSS ライブラリの中には他の OSS ライブラリを利用して開発されているものも多く，開発中のプログラムはそれらのライブラリにも間接的に依存関係を持つことになる．その結果，開発中のプログラムで使用する 1 つのライブラリを更新した場合にそのライブラリを利用している他のライブラリも同時に更新する必要が生じる場合がある．しかし，依存関係が推移的に発生するため，間接的に数多くのライブラリに依存し，またその関係が非常に複雑なものになる場合がある．例えば，Gradle というビルドツールは直接の依存関係はあまり多く持たないが，プロジェクトのコンパイル時には 100 以上の依存関係がクラスパスに含まれる [2]．このような状況で，1 つのライブラリのバージョン間での非互換性が与える影響を予測することは難しい．

開発者がライブラリのバージョンを選択する際には，互換性の問題に加えて機能的な問題やバグが含まれていないかどうかなど様々なことを考慮する必要がある．全てを考慮し最適な選択をすることは難しいため，既になされている多数の開発者による選択は正しい可能性が高い，という考えのもと，選択の指標として実際の利用状況を用いることが提案されている．既存研究 [3][4] では，主にライブラリの利用数の時間的な変化を可視化することによっ

[1] https://github.com/
[2] http://mvnrepository.com/

て，現在の利用傾向や人気の移り変わりを明らかにしている．しかし，1 つのライブラリに関する可視化だけではそのライブラリのアップデートによる影響が他のライブラリに及ぶ可能性を考慮することができない．

本研究では，ライブラリのバージョンの組み合わせに着目した利用状況の可視化を提案する．依存関係の全てを考慮することは難しいが，自身が直接依存するライブラリのバージョンを選択する際に，既に他のシステムによって利用されている実績のある組み合わせを選択することで，ライブラリ間での互換性の問題が発生するリスクを低減する．そこで実際に，GitHub で公開されているプロジェクトにおけるライブラリの利用状況を抽出し，Parallel Sets[5] という手法を用いて可視化を行うツールを作成した．

以降，2 章では本研究の背景について述べる．3 章では使用した可視化手法 Parallel Sets について述べ，4 章では可視化対象としたデータとデータベースの作成について述べる．5 章でツールの実装について述べ，6 章で実際に可視化を行った例を示す．最後に 7 章でまとめと今後の課題について述べる．

2　背景

2.1　ライブラリの互換性

Java における互換性は以下の 3 つに分けられる [6].

- ソース互換性
  利用元のシステムと一緒に再コンパイルを行うことで動作する
- バイナリ互換性
  ライブラリの古いバージョンを用いてコンパイルされていたプログラムが再度コンパイルすることなく新しいバージョンのライブラリとリンクし動作する
- 動作互換性
  プログラムを実際に動かしたときに同じ結果が得られる

これらの互換性はそれぞれが独立したものであり，どの互換性も他の要素と関係なく損なわれる可能性がある．近年，再ビルドを行うことなく，実行中のシステムにコンポーネントを追加したり差し替えたりする技術が広く使われ始めており，バイナリ互換性の重要度が高まっている．動作互換性に関してはライブラリの内部での動作が変わった場合にも互換性がないと定義されるが，開発者は通常，API 仕様書通りにプログラムが動作することを前提としてライブラリを利用する．そのため動作の非互換性によって利用者が影響を受けることは少ない．

バージョン間でのバイナリ互換性が崩れるような変更としては，最も多いものにメソッド，クラス，フィールドの削除があり，その他にもメソッドのシグネチャの変更やフィールドの型の変更など様々な要因が存在する．このような変更がライブラリに加えられると依存するソフトウェアの開発者はライブラリをアップデートする際に自身のコードに手を加える必要がある．

Kim らのライブラリの互換性とその影響の研究 [7] において，ライブラリの非互換性がシステムに何らかの影響を与えるケースはかなり高い頻度で発生することが示されている．利用者がバージョンの更新を行っても利用者に影響が及ばないようなリリースを，後方互換性があるアップデートという．

2.2　推移的な依存関係とバージョン競合

推移的な依存関係とは，プロジェクトがあるライブラリに依存しているとき，その依存ライブラリがさらに別のライブラリに依存しているような関係のことである．推移的な依存関

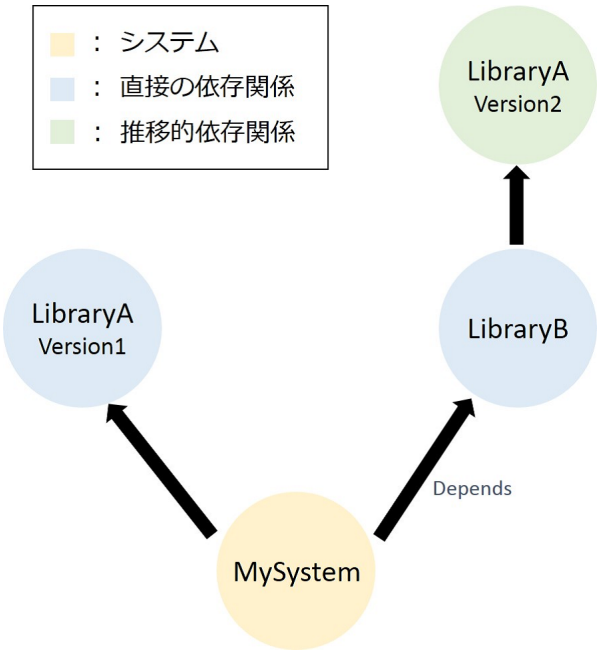

図 1: バージョン競合

係が発生することによって間接的に多くのライブラリに依存することになる．このような複雑な依存関係を管理するために，Maven や Gradle などといったツールが利用されている．

推移的な依存関係によって，例えば図 1 に示したような依存関係のバージョン競合が発生する場合がある．図中の頂点はシステムやライブラリ，矢印は依存関係を示しており，開発中のプロジェクトでライブラリ A と，ライブラリ B を利用しようとしている状況を表している．ここで，自身のプロジェクトがライブラリ A のバージョン 1 を利用していて，ライブラリ B が内部でライブラリ A のバージョン 2 を利用していた場合を考える．JavaVM では，同じライブラリの複数のバージョンを同時に利用する簡単な方法が存在しないため，通常ライブラリ A のどちらかのバージョンを選択し利用することになる．

このような競合は直接依存するライブラリだけではなく，推移的な依存関係同士でも発生する場合があり，これがライブラリのバージョン更新の妨げになる場合がある．図 2 で，依存ライブラリのアップデートの例を示す．この図では，開発中のシステム S_1 を S_2 としてアップデートする際，ライブラリ B の新機能を使うため，S_1 が使用していた B_1 から新しいバージョン $B2$ に更新しようとした．ライブラリ B はバージョン 1 ではライブラリ C のバージョン 1 を使用していたが，バージョン 2 ではライブラリ C のバージョン 2 を使用するように変更されていた．一方で，システム S_2 が使用しているライブラリ A はライブラリ C のバージョン 1 を使用し続けている．これによって，開発中のシステムが直接依存してい

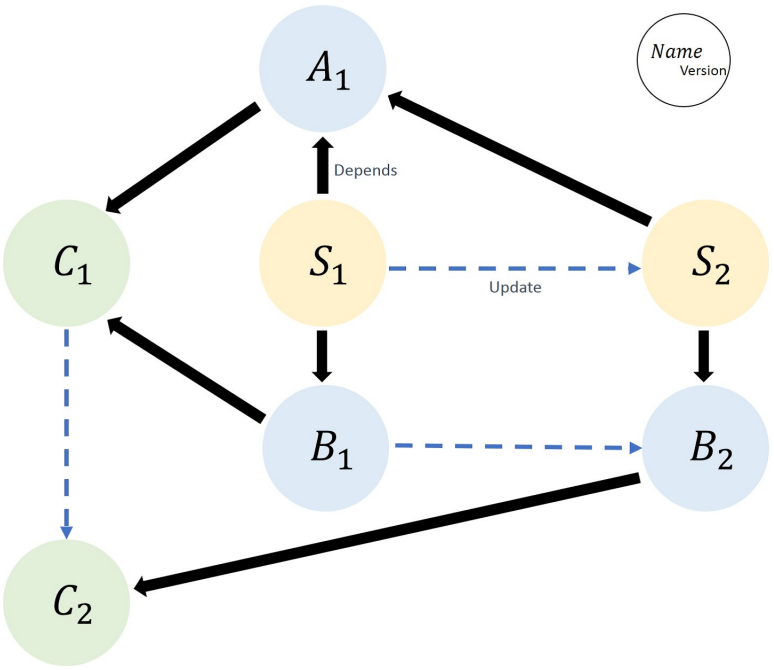

図 2: 依存ライブラリの更新

ないライブラリ C のバージョン競合が発生する．このような場合に, 競合したライブラリ C のバージョン間での後方互換性が成り立っていないとき，ライブラリ A のバージョン 1 とライブラリ B のバージョン 2 を組み合わせて利用することができなくなり，ライブラリ A の別の，ライブラリ C のバージョン 2 に依存するようなバージョンを探すなどの対策が必要になる．本研究では主にこの問題への対策として，組み合わせの可視化を行う．

実際に，ASM[3] というバイトコードを扱うライブラリは，HIBERNATE[4] や Spring[5],, GLASSFISH[6] などの人気の高い OSS ソフトウェアから利用されていたが，バージョンアップの際に後方互換性を失うような変更が度々加えられていた．これにより，例えば ASM の 1.x に依存していた HIBERNATE と，ASM3.x に依存していた Spring を組み合わせての使用を行った際にエラーが発生するなど，多くの問題が報告された [8].

[3]http://asm.ow2.org/
[4]http://hibernate.org/
[5]http://projects.spring.io/spring-framework/
[6]https://glassfish.java.net/

2.3 セマンティックバージョニング

ソフトウェアのバージョン番号のつけ方として，セマンティックバージョニング[7]が推奨されている．これはバージョン番号を

$$MAJOR.MINOR.PATCH$$

の 3 つの数字で構成するスキーマで，

- MAJOR: 互換性のない API の変更を行った場合
- MINOR: 後方互換性が保たれた状態で新しい機能を追加した場合
- PATCH: 後方互換性が保たれた状態でバグフィックスを行った場合

それぞれを上記の条件でインクリメントする．

全てのライブラリがこれに従っているならば，メジャーバージョン間のアップデート以外であれば適用しても問題が起こる可能性は低いはずである．しかし，Raemaekers らの Maven リポジトリにおけるライブラリのセマンティックバージョニングと互換性の関係に関する研究 [9] により，多くのライブラリにおいてこのスキーマが守られておらず，MINOR や PATCH のアップデートにおいても後方互換性が保たれない変更が加えられていることが判明している．そのため，マイナーやパッチのバージョンアップであっても開発者はすぐに適用を決めることが難しくなっている．

[7] http://semver.org/

表 1: タイタニックの乗客データ

Dimension	Values
Class	First, Second, Third, Crew
Sex	Female, Male
Survived	Yes, No

3 可視化手法

この章では，ParallelSets に関する説明と，ライブラリ利用状況の可視化への適用についての説明を述べる

3.1 Parallel Sets

Parallel Sets とは，Kosara らによって考案された，複数の観点からカテゴリーによって分類することができるようなデータセットに対する可視化手法である [10]．タイタニックの乗客に関するデータが可視化対象の例として挙げられている．乗客は 1 人 1 人が表 1 に示したように，利用クラス（等級），性別，生存したか，の 3 つのパラメーターによってそれぞれ分類される．Parallel Sets の原型となっている手法として，Parallel Coordinates と呼ばれる，n 部グラフ上にこのようなデータを描画する手法が存在する．各パラメータに独立頂点集合を対応させ，各分類を表す点と点の間の辺を見ることで 1 つのデータに関するパラメータの組み合わせが表される．利用クラス，性別，生存したか，の 3 つのパラメータを用いて考える．例えば「3rd クラスを利用している男性の生き残った人」のデータは図 3 のように表せる．これによって個別のデータについてのパラメータの組み合わせを知ることができるが，複数のデータの描画を行う際，Parallel Coordinates では同じパラメータを持つデータが重なって描画されてしまい，数を表すことが難しいという問題がある．

図 4 に Parallel Sets によるタイタニックの乗客データ [8] の可視化例を示した．Parallel Coordinates 点と辺に幅を持たせることで，データ数を表現する．さらに，2 段目以降の分類をパラメータの組み合わせによって分割することで木構造を作り，パラメータの組み合わせとその数の明確な可視化が行われる．横幅全体がデータセット全体，つまり全乗客数となっており，各階層はパラメータに対応している．各階層は分類によって分割され，その幅は属するデータの数に応じた割合となる．また，各階層間を結ぶ線によってパラメータの組み合わせが表わされる．色は 1 層目での分類を表している．この線は木構造を構築することによって下層に行くほど分割され細くなっていく．これを用いると，図 4 の可視化において

[8] http://www.amstat.org/publications/jse/v3n3/datasets.dawson.html

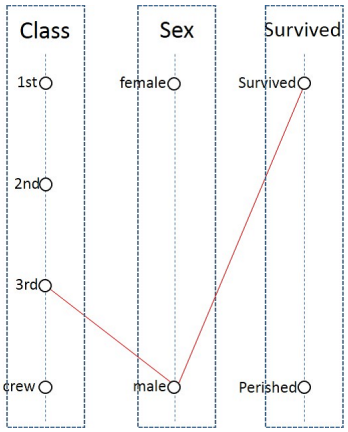

図 3: Parallel Coordinates

「3rd クラスを利用している男性の生き残った人」のデータは矢印で示された部分で表わされる．この場合，赤線で示した幅によって同じ組み合わせでパラメータを持つデータ数の，全体のデータ数に対する割合が表される．この場合には乗客の 4 %の数が Third Class, male, Survived の組み合わせでパラメータを持つので，対応する線が全体の 4 %ほどの幅となっている．

Parallel Sets では，ユーザーが操作をすることによって以下のような機能を使用することができる．

- 各部の具体的な数の表示
- 階層の並び替え
- 分類位置の入れ替え，ソート機能

Parallel Sets による可視化単体で具体的な数を把握することは困難であるが，各階層における分類上や，階層間を結ぶ線にマウスオーバーすることによって数を表示させることが可能である．また，各分類の位置の入れ替え，階層の並び替えが可能である．入れ替えを行うことによって，階層間を結ぶ線は自動的に再描画される．

3.2 ライブラリ利用状況の可視化への適用

既存のシステムにおけるライブラリ利用状況のデータセットを用いて可視化を行う．各データは 1 つのシステムとなり，パラメータとして各ライブラリへの依存状況をもつ．タイタニックの乗客の例と同じように表すと，ライブラリ A，ライブラリ B，ライブラリ C への依存状況に関するデータは表 2 のようになる．パラメータの分類となる，バージョンの部

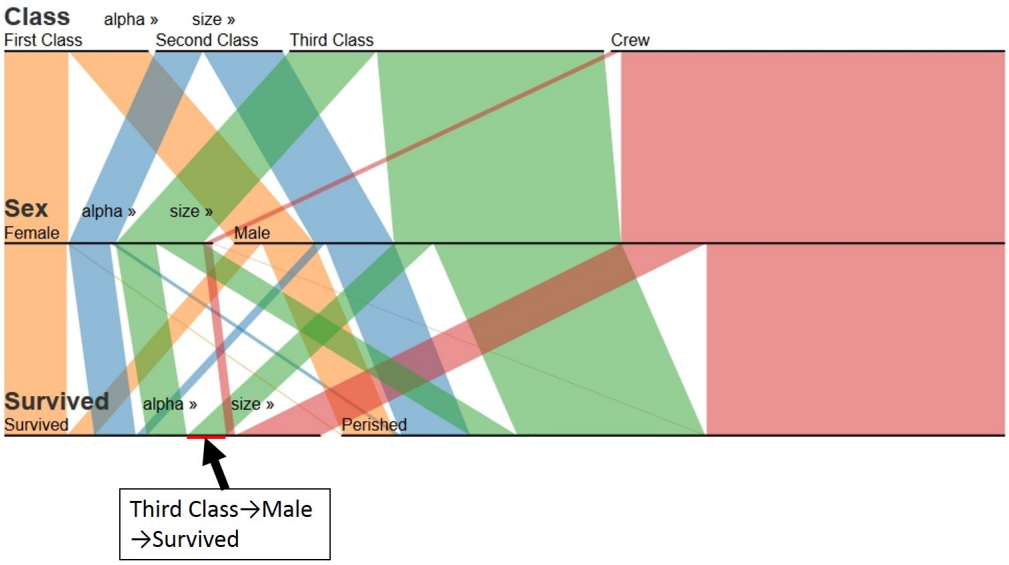

図 4: Parallel Sets によるタイタニックの乗客データの可視化

表 2: ライブラリ利用状況のデータ

Dimension	Values
Library A	Ver. 1.0, Ver. 2.0, none
Library B	Ver. 1.0, Ver. 2.0, none
Library C	Ver. 1.0, Ver. 2.0, Ver. 3.0, none

分はライブラリによって数も内容も異なる．また，”none”はそのライブラリを利用していないことを表す．

このようなデータを Parallel Sets で可視化すると，図 5 のようになる．各階層で，データセットに含まれるシステムが各ライブラリのどのバージョンを利用しているかによって分類されている．また，階層間の線とその幅によって，利用の組み合わせとその利用数を知ることができる．Parallel Sets を用いた理由としては，

- データセット内で同じパラメータの組み合わせを持つデータの量を素早く把握できる
- 他の組み合わせとの比較が可能
- パラメータを個別に見ることが可能

という特徴が有用であると考えたからである．

例えば，図 5 で一番太い線を見ることで，A のバージョン 2.0，B のバージョン 2.0，C の

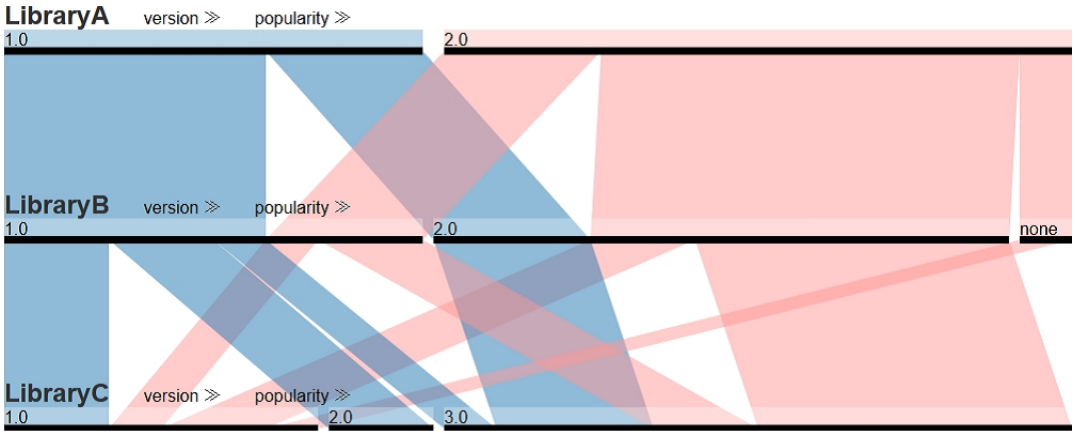

図 5: Parallel Sets

バージョン 3.0 の組み合わせでの利用数が一番多く，人気であることがわかる．他の組み合わせについても同様に線の太さによって利用数が表されているため，組み合わせ同士での比較も容易に可能である．また，バージョン順，利用数順に階層内の分類を並び替えることによって各ライブラリ単体に関する利用状況も知ることが可能になるなど，様々な見方が可能になる．

4　依存関係データベースの作成

ライブラリのバージョンの組み合わせを可視化するための前準備として，ソフトウェアおよびライブラリ間の依存関係のデータを収集し，データベースの作成を行った．

4.1　依存関係データの収集

4.1.1　調査対象

実際に GitHub で公開されているコミット数が 100 以上のプロジェクトの中で Maven2 を使って管理されている約 4000 のプロジェクトをダウンロードし，内部に含まれる pom.xml から依存関係を収集した．プロジェクトのダウンロード作業は 2014 年 8 月に行った．コミット数が多いプロジェクトを対象としたのは，保守期間の短いプロジェクトに比べて長期間保守されてきている実績があり，依存関係の管理がしっかりしている可能性が高いと考えたからである．当初依存関係の収集は全コミットに対して行っていたが，組み合わせに対しての人気度を可視化するにあたって複数バージョンに対して組み合わせを調べた場合，大部分が同じ組み合わせであるライブラリの組み合わせ利用数がプロジェクトのコミット数に応じて増加してしまうという問題が発生した．そのため，各プロジェクトの最新の 1 コミットのみを対象とした．

今回解析対象とした Maven では，ソフトウェア開発プロジェクトを Project Object Model(POM) としてモデル化している．Maven によって管理されているプロジェクトは POM によって定義された，ID，バージョン，ビルド，依存関係などのプロジェクトの構成要素を POM ファイル (pom.xml) に記述している．また，Maven は依存ライブラリの管理ツールとしての側面を持っている．依存ライブラリを pom.xml 内に記述することによって，ビルド時に自動的に Maven Central リポジトリなどのリモートリポジトリからダウンロードされる．依存ライブラリは以下の 3 つの項目によって特定される．

- groupId： 会社名，開発チーム名など
- artifactId： ライブラリ名など，グループ内から 1 つを特定できる固有の名前
- version： バージョンの指定

2 つの id によってライブラリの種類が特定され，その中からバージョンを指定するような形になっている．図 6 が記述例である．これはグループ ID が org.neo4j，アーティファクト ID が neo4j であるライブラリのバージョン 2.4 への依存を表す．

```
<dependencies>
        <dependency>
                <groupId>org.neo4j</groupId>
                <artifactId>neo4j</artifactId>
                <version>2.1.6</version>
        </dependency>
</dependencies>
```

図 6: pom.xml での依存関係の記述

4.1.2 依存関係の調査

よって POM ファイルに対する解析を行うことで，依存関係の調査が可能である．

対象としたプロジェクトに含まれる POM ファイルから，依存関係の抽出を行うことによって依存関係のリストを作成した．推移的依存関係については考慮しておらず，各ファイル内の情報のみを利用している．依存関係の収集には，PomWalker[9] というツールを用いた．PomWalker は，プロジェクトのディレクトリを入力として与えることで，プロジェクト内に含まれる全てのモジュールに対し Pom ファイルを検索，構文解析を行い，依存関係のリストを作成する．これに含まれるのは，システムに関する情報 (groupId，artifactId，version) と，その依存先のライブラリに関する情報 (groupId，artifactId，version) と日付である．抽出したデータは 3 のようになっている．

ライブラリのバージョン指定部では，範囲での指定や，親 POM ファイルでの記述に従うなど，バージョンの指定が曖昧なものが含まれる場合がある．このような曖昧なバージョン指定が行われている依存関係に関しては，簡略化のため，削除を行った．この操作によって，収集した依存関係のうちの約半分が削除された．大幅に数が減った要因としては，対象が大規模なプロジェクトであったことが考えられる．大規模プロジェクトでは，親プロジェクト内の POM ファイルによって，一括で依存関係の管理を行っているものが多い可能性が高い．そのため，*project.version* のような親 POM ファイルに従うことを表す記述の削除回数が多くなった．

表 3: 依存関係リスト

SysGroupId	SysArtifactId	SysVer	DependsDate	LibGroupId	LibArtifactId	LibVer
com.henry4j	text	1.0-SNAPSHOT	2014/2/4	org.apache.mahout	mahout-core	0.9
com.henry4j	text	1.0-SNAPSHOT	2013/9/18	com.google.guava	guava	14.0.1

[9] https://github.com/raux/PomWalker

4.2 データベースの作成

依存関係を管理するにあたって，neo4j という Java ベースのグラフデータベースを用いた．グラフデータベースとは，node(頂点) と 2 つのノード間の relationship(関係)，さらに node と relationship に対して key-value の形で与えられる property(属性)，によって構成される，ノード間の関係性を表現することに適したデータベースである．relationship は有向であり，1 つのノードに対して複数設定することができる．また，トラバーサル (traversal) と呼ばれる，関係先のノード検索を行う際，結果が高速に得られるという特徴を持つ．例えば指定したライブラリに依存しているシステムについて知りたいような場合に，ノードを高速に検索し，列挙することができる．よってソフトウェア間の依存関係を表す用途にも非常に適していると考えられる．

今回，node をソフトウェア，relationship をその依存関係とすることでグラフデータベースを作成した．各ノードはプロパティとして名前とバージョンを設定し，関係 Depends はプロパティに日付を設定する．図 7 はグラフデータベースの構造を視覚化した例である．A のバージョン 1 が B のバージョン 1 に 2015/2/13 に依存関係を持ったことを表す．

4.2.1 データベースへの依存関係の追加

表 3 のような依存関係のリストの各行に対して，以下のような手順で処理を行いデータベースへ追加した．ここで，各ノードの name プロパティは $GroupId + ArtifactId$ としている．

1. name が SysGroupId+SysArtifactId，version が SysVer であるノードをデータベースから検索する．存在しない場合にはノードを新しく作成しデータベースへ追加する．これがシステムのノードとなる．

2. name が LibGroupId+LibArtifactId，version が LibVer であるノードをデータベースから検索する．存在しない場合にはノードを新しく作成しデータベースへ追加する．これがライブラリのノードとなる．

3. 検索または新規追加したシステムのノードからライブラリのノードへの関係 Depends

表 4: 抽出した依存関係の数

All version	23589910
of Latest version	839785
Explicit Systems	427554

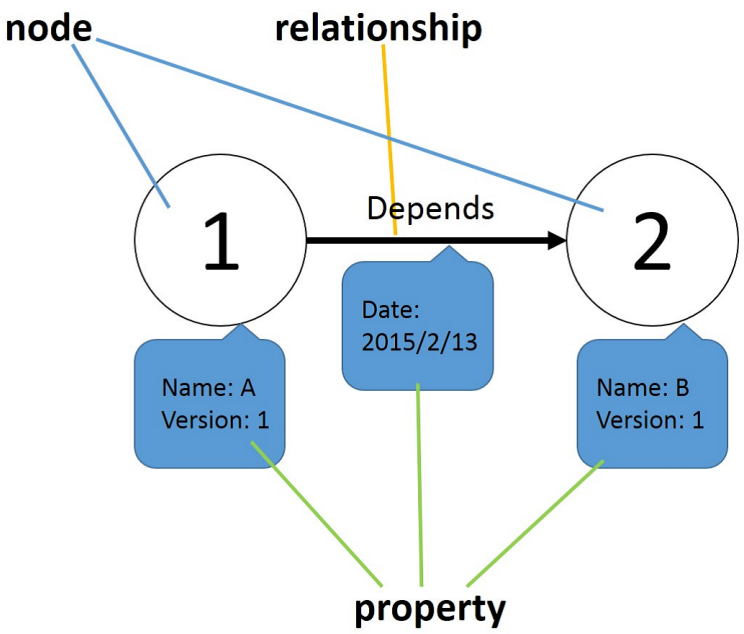

図 7: Neo4j を用いた依存関係データの管理

を作成，Date プロパティに DependsDate を設定する．

4.2.2　データに関する調査

Neo4j グラフデータベースでは，Cypher（サイファー) と呼ばれる問い合わせ言語が使用可能である．これを用いてデータに関する調査を行った．

まず 1 つ目に，依存関係リストに現れたシステムとライブラリの種類数，ライブラリとなっているノードの数を取得した．ここで，システムとは他のノードに対し 1 つ以上の関係 DEPENDS を持っているノード，ライブラリとは他のノードから 1 つ以上の関係 DEPENDS を持たれているノードと定義する．結果を表 5 に示した．

次に各ライブラリの利用数について調べた．ある程度の利用数があるライブラリでなければ組み合わせでの利用があまり期待できず，本研究で目的としている組み合わせでの利用傾

表 5: ライブラリとシステムの数

システムの種類数	ライブラリの種類数	ライブラリ数
36910	29535	151647

向の可視化を行うことが難しい．結果の一部として，調査対象としたプロジェクトの間で利用数の多い，つまり人気の高い上位 10 種のライブラリを表 6 に示した．これにより，対象データ内では JUnit というテストを自動化するフレームワークの利用数が他のライブラリに比べてかなり多いことが分かる．

表 6: 利用数上位

GroupId	ArtifactId	利用数
junit	junit	13369
log4j	log4j	3513
org.slf4j	slf4j-api	3098
javax.servlet	servlet-api	2808
org.auraframework	auradocs-integration-test	2622
commons-io	commons-io	2411
com.google.guava	guava	2408
commons-lang	commons-lang	2015
commons-logging	commons-logging	1873
com.khubla.pragmatach	pragmatach-framework	1864

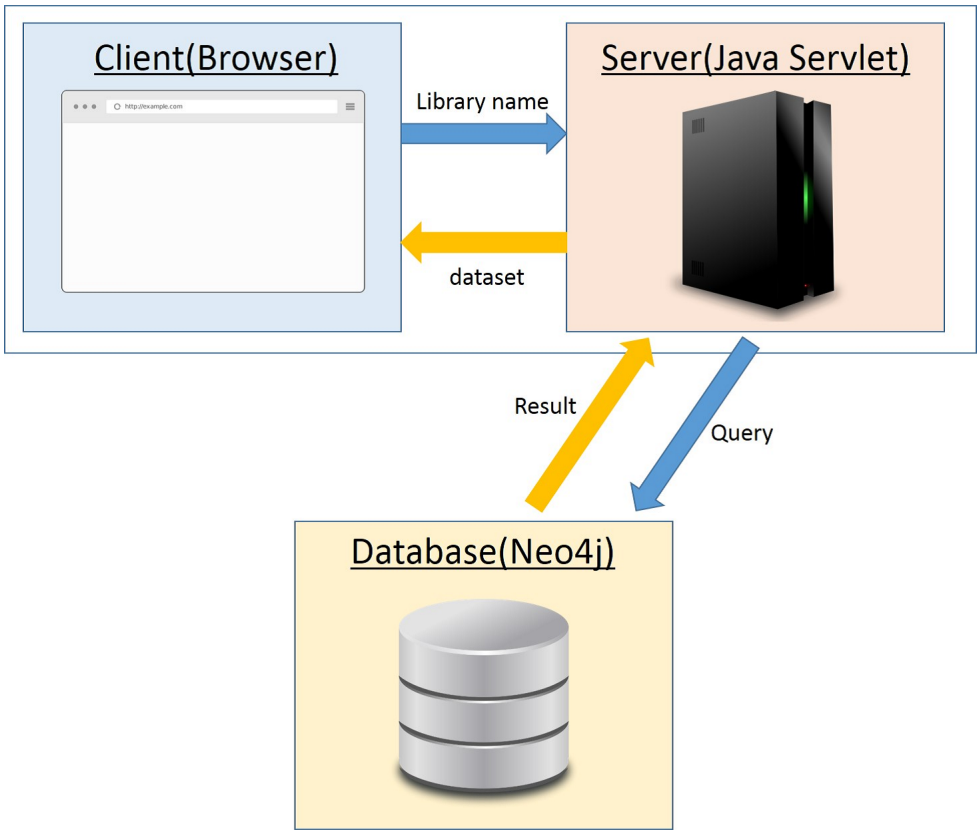

図 8: 可視化ツールの実装

5　可視化ツールの実装

本研究では，ライブラリを複数選択することで，GitHub 内の約 4000 プロジェクトにおける選択ライブラリの組み合わせ利用状況の可視化を行うツールの作成を行った．作成ツールは図 8 のような構造になっている．利用者がブラウザ上でライブラリを選択することで，サーバーがデータベースからの検索を行い，結果が Parallel Sets として可視化される．

可視化が行われるまでの工程は以下のようになる．

1. ユーザーがライブラリを選択する
2. サーバーにリクエストが送られ，データベースに対し検索を行う
3. 検索結果をサーバーからクライアントに返す
4. Parallel Sets を描画する

この章では各部の実装についての解説を行う．

5.1 サーバー部

サーバーとして，リクエストに応じた検索をクライアントからリクエストを受け取ることで，データセットを含む HTML 情報をクライアントに対して出力する Java サーブレットの作成を行った．サーブレットコンテナには Apache Tomcat を使用した．

主な機能は，リクエストに基づいた，データベースに対する検索と取得した情報の整形である．これは以下のような手順で行う．

1. リクエストとして受け取ったライブラリ群に依存関係を持つシステムの列挙
   選択されたライブラリ群のうち，いずれか 1 つ以上に依存しているシステムの列挙を行う．つまり，システムに対して各ライブラリの利用について OR 検索をかける．例えば，junit と，log4j がリクエストされた場合のノードの取得操作を Cypher によるクエリで書くと以下のようになる．

   ```
   MATCH (sys:Product)-[DEPENDS]->(lib:Product)
   WHERE lib.name = "log4j.log4j" OR lib.name = "junit.junit"
   RETURN COUNT(DISTINCT sys)
   ```

2. 依存バージョンの特定
   この手順では，手順 2 において取得したシステムがそれぞれ，リクエストされたライブラリのどのバージョンを利用しているかを特定する．列挙されたそれぞれのシステムが持つ依存関係を全て調べ，依存先の名前が選択されたライブラリに等しいものを探し，そのバージョンをデータセットとして書き出す．OR 検索によって列挙されたシステムのリストであるため，当然いずれかのライブラリに依存関係を持たないものが含まれる．その場合依存バージョンを”none”として使用していないことを示している．

   書き出されるデータセットは表 7 のような形式となる．ここで，各行が 1 つのシステムに対応し，各項目見出しとなっているライブラリに対する依存バージョンの出力を行う．例えば 1 行目は，Log4j のバージョン 1.2.17 と Junit のバージョン 4.11 を組み合わせて使用している 1 つのシステムがあることを表す．2 行目は Log4j のバージョン 1.2.17 を使用しているが，Junit を使用していないシステムを表す．

3. HTML の書き出し
   データセットの配列を生成するコードを含んだ，parallel sets 描画ページの HTML 情報をクライアントに返す．

5.2 クライアント部

クライアント部では，ユーザーがライブラリ名を選択しサーバーへ送信することで，検索結果が Parallel　Sets として可視化される．Parallel Sets の描画には，D3.js[10] という JavaScript のライブラリを使用している．

可視化が行われているページでは，3.1 章で紹介したような Parallel Sets に関する操作に加えて，対象とするデータセットの範囲の指定が可能である．デフォルトでは，選択されたライブラリ全てを利用しているという条件，つまり表 7 のようなデータで示されるようなデータにおいて全てのライブラリに対する依存関係に”none”を含まないシステムだけをクライアント側で絞り込み，可視化を行うが，選択したライブラリを全て組み合わせて使用しているシステムがデータセット内に存在していない，または数が少なすぎる場合がある．このような場合に，サーバーで行われた検索手順で選択ライブラリのうちいずれか 1 つ以上を利用しているシステムに関するデータを取得済みであるため，この範囲までであれば可視化範囲をクライアント側で広げることができる．つまり，選択ライブラリのうちのいくつかを絞り込み条件から外すことによって，可視化対象とするデータを増やすことが可能となる．

図 9 で各円は，対応するライブラリを利用しているシステムの集合を表すとする．この 3 つの円に含まれる範囲のデータセットを，クライアントは再検索を行うことなく利用することができる．例えば，ライブラリ A,B,C を選択した際全てを利用しているシステム，つまり図 9 の左の赤い範囲で示された範囲のデータセットが可視化対象となるが，ライブラリ B の利用を絞り込み条件から外すと，図の右側で示された範囲に含まれるシステムを可視化対象とすることができる．図 10 にこのような絞り込みを行った例を示した．ライブラリ A,B,C 全てを使用している 95 のシステムに関する可視化が行われている．ここで可視化対象となる条件を，ライブラリ A,C を使用しているものに変更することによって，ライブラリ B を利用していないシステムもデータセットに含まれることとなり，100 のシステムが可視化対象となった．すると，図 10 の下部のようにライブラリ B を利用していない 5 つのシステム

表 7: データセット

Log4j	Junit
1.2.17	4.11
1.2.17	none
1.2.17	none
1.2.16	4.8.2

[10]http://koyamatch.com/d3js/index.html

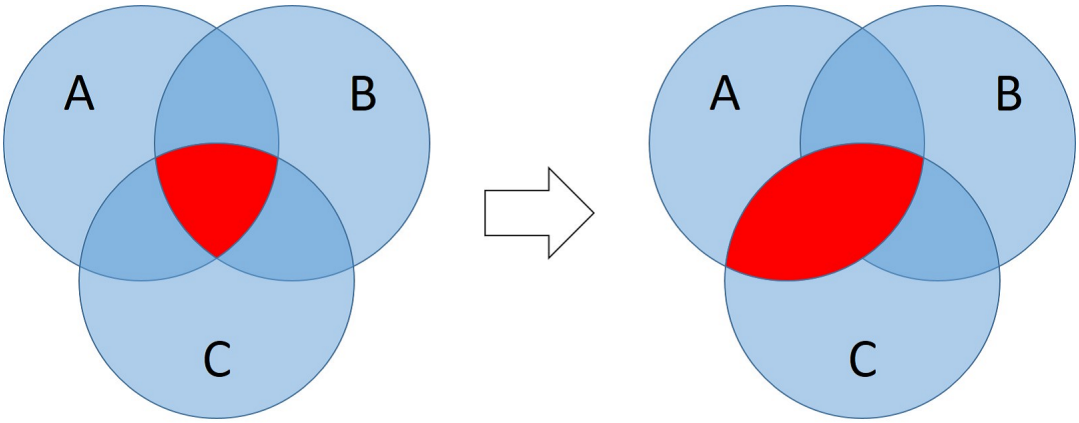

図 9: 可視化するデータ範囲の変更

が none に分類されグラフ上に現れる．このような絞り込みによって，ライブラリ A,C を利用しているシステムのうちライブラリ B も同時に利用しているシステムの数，などといった情報も知ることができるようになっている．

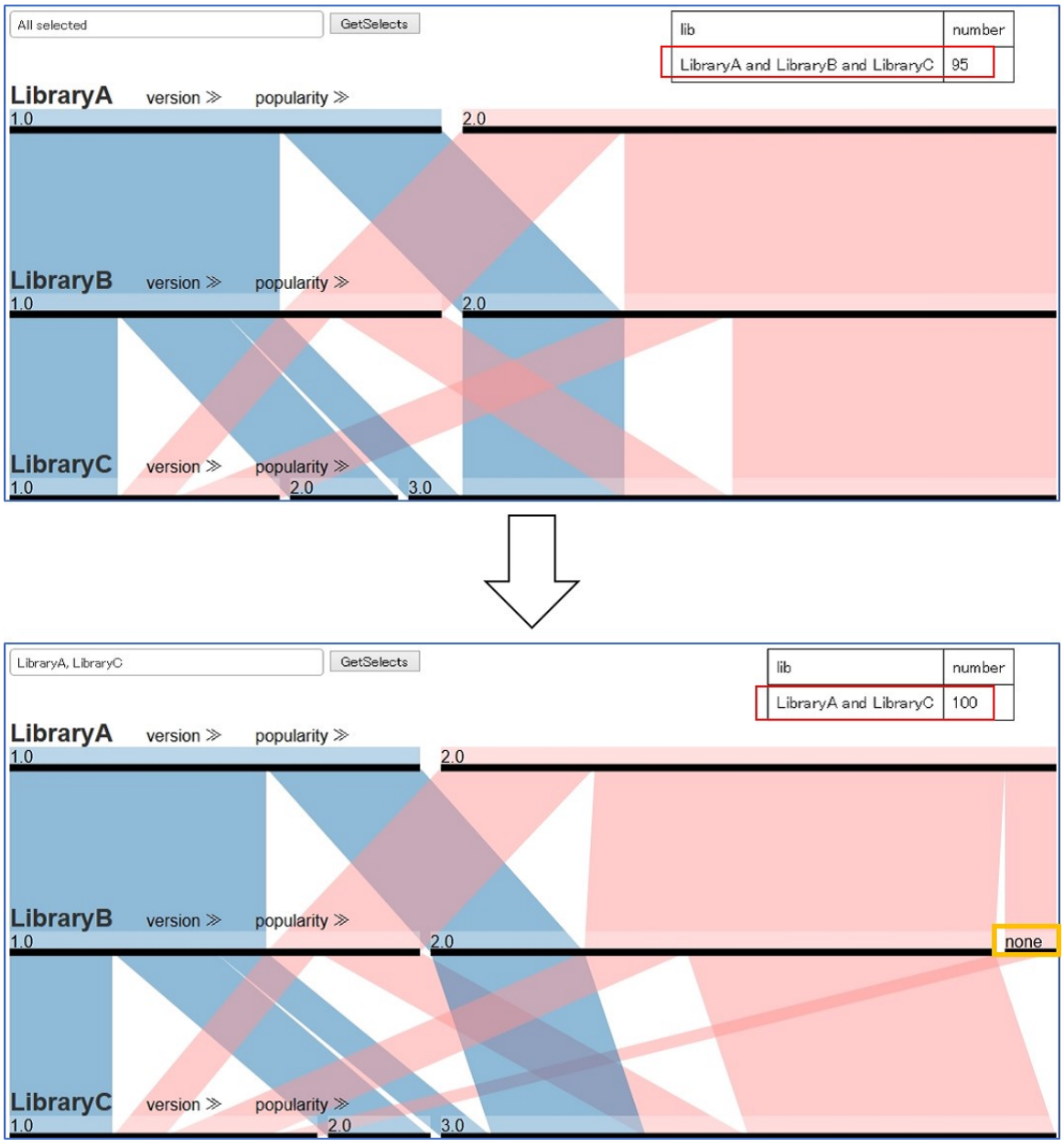

図 10: 可視化するデータ範囲の変更例

6 使用例

本研究で行ったライブラリの組み合わせ利用状況の可視化は，使用ライブラリの組みあわせがある程度決まっている場合の利用を想定している．例えば以下のような状況が考えられる．

1. 開発中のシステムに新しいライブラリを導入する現在使用しているライブラリの組み合わせに対して，新規に導入するライブラリの利用が行われている例は存在するか，存在するならばどのバージョンが利用されているのかを調べる．
2. 現在利用しているライブラリのアップデート作業を行う利用中のライブラリのアップデートを行う際に，他のシステムによって利用されているバージョンの組み合わせを調べる．

今回，実際のライブラリを用いてケーススタディを行った．以降で詳細を述べる．

6.1 ケース設定

開発中の Http のクライアントアプリケーションでは，Commons-collections[a] のバージョン 3.2 と，commons-httpclient[b] のバージョン 3.1 を利用している．機能追加のアップデートを行う際，新しくカレンダーを扱うライブラリとして，Joda-Time[c] を導入したい．

[a] http://commons.apache.org/proper/commons-collections/
[b] http://hc.apache.org/httpclient-3.x/
[c] http://www.joda.org/joda-time/

このケースにおいて，既存のシステムにおける 3 つのライブラリの利用状況から得られる情報を元に利用バージョンの決定を行っていく．以下の 2 つの場合にどのように各ライブラリのバージョンを選択を行うかを示す．

- 現在使用しているライブラリのアップデートを行わず，Joda-Time を導入する
- 現在使用しているライブラリもこの機会に合わせて更新を行う

6.2 現在の環境に合うバージョンの選択

現在使用しているライブラリのアップデートを行わず，Joda-Time を導入する場合．つまり Joda-Time のどのバージョンが現環境に適合する可能性が高いのかを調べる．この場合には開発中のシステムと，同じライブラリの同じバージョンを利用しているシステムが Joda-time を使っているかどうか，また，どのバージョンを使っているかを調べる．ParallelSets を用いて図 11 のようにこの情報を得ることができる．

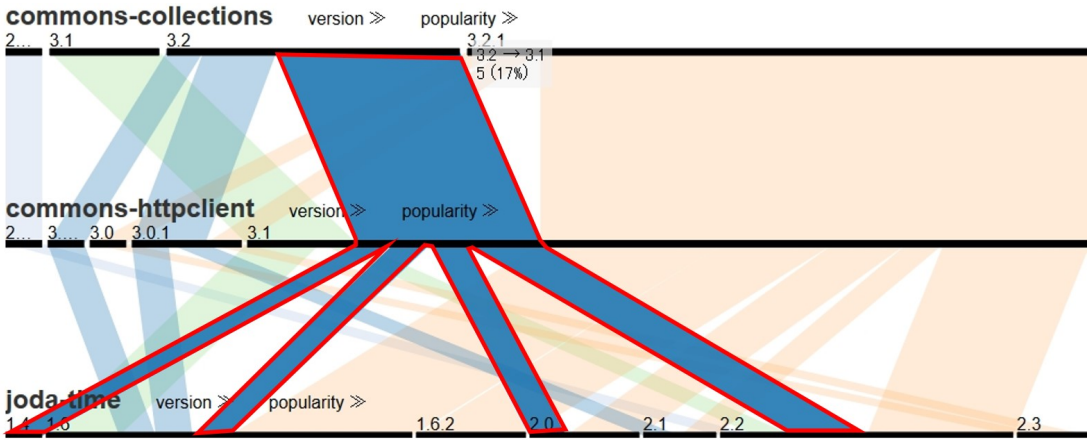

図 11: 組み合わせに対するバージョン選択

図 11 の赤線で示されている部分を見ると，Commons-Collections のバージョン 3.2 と Commons-HttpClient のバージョン 3.1 を使用しているシステムにおいて，Joda-Time のどのバージョンが利用されているかを知ることができる．1 層目，2 層目で 3.2 → 3.1 とつながれている線によって Commons-Collections のバージョン 3.2 と Commons-HttpClient のバージョン 3.1 を使用しているシステムが表されている．この線は 3 層目で 4 つに枝分かれしているため，これらのシステムにおいて Joda-Time の 4 種類のバージョンが利用されていることがわかる．よって候補となる Joda-Time のバージョンは 1.4，1.6，2.0，2.2 の 4 つとなる．どのバージョンにも利用実績があるので，新しいものを使いたい場合には 2.2 を選択するなど，自由な選択ができると判断できる．

6.3　ライブラリのアップデート

使用中のライブラリの更新作業も同時に行う場合，どの組み合わせを利用すべきかを調べる．利用数の多い組みあわせ，新しいバージョンの組み合わせの 2 つの観点から考える．

まず，各階層内の分類を利用数順に並び替えを行うと，12 のようになった．赤線で示された部分で表された 3.2.1 → 3.1 → 1.6 の組み合わせの利用数が一番多いことがわかる．この組み合わせは各ライブラリ単体の最も利用数の多いバージョンとなっている．

次にバージョン順に並び替えを行うと，13 のようになった．ここから可能な限り新しいバージョンの組み合わせを探すと，赤線で示された 3.2.1 → 3.1 → 2.2 の組み合わせが見つかる．また，この組み合わせは 2 番目に利用数が多い．データセットに含まれる Joda-Time の最新バージョンは 2.3 であるが利用数が少なく，さらに，組み合わせて利用されている Commons-HttpClient のバージョンが 3.0 と 3.01 であり，現在使用しているバージョンより

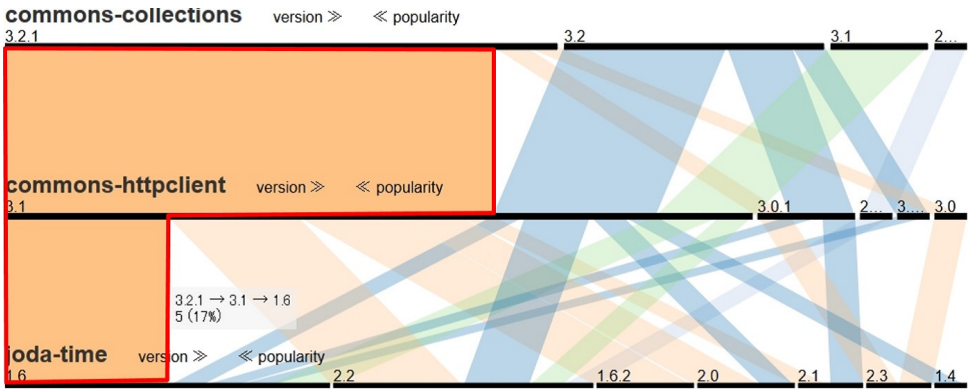

図 12: 最も利用数の多い組みあわせ

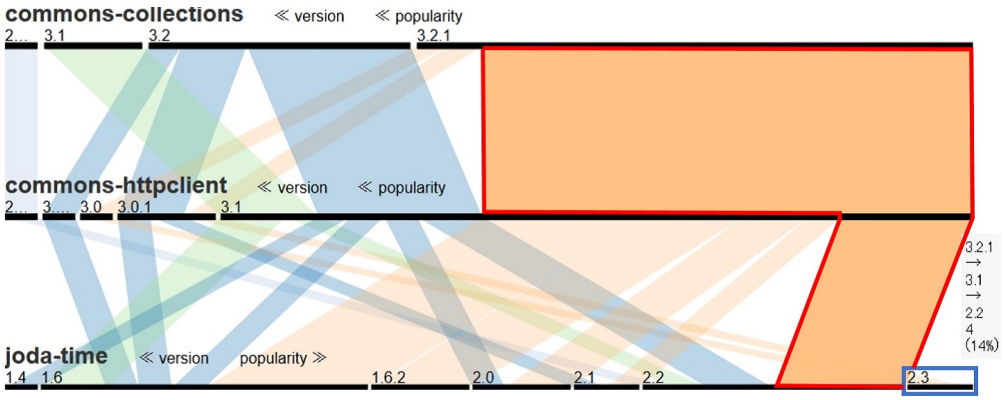

図 13: 新しいバージョンの組み合わせ

も古いものになっている．問題が発生するとは限らないが，このような場合にこの選択を行う際には注意が必要となる．

以上よりこのケースでは，Commons-Collection のバージョン 3.2 と，Commons-HttpClient のバージョン 3.1 と，Joda-Time のバージョン 2.2 の組み合わせでの利用が推奨される．

7　まとめと今後の課題

本研究では，GitHub で公開されているプロジェクトにおけるライブラリへの依存関係について調査を行い，Parallel Sets を用いて利用状況の可視化を行うツールの作成を行った．また，このツールによって可視化される組み合わせでのライブラリ利用状況を元にどのようにバージョンの選択を行っていくかを，実際のライブラリに関するデータを用いて解説した．

今後の課題としては，本研究で行ったデータの収集手順において，明示的にバージョンが指定されていないものは取り除いたが，取り除いたものに関しても実際のシステムは利用を行っている．組み合わせでの可視化を行うにあたって，依存関係が欠けることによって不適切な組み合わせを提示してしまうことはないと考えたため今回は対象外としたが，依存先のバージョンを収集段階で解決することで，データセットの正確性を上げられる可能性がある．また，推移的依存関係を考慮したデータセットの作成も今後の課題となる．

ツールにも多数の改善案が挙げられる，今回作成した可視化ツールではライブラリの組み合わせを予め決めておき，そのライブラリに関するデータの可視化を行う場面を想定していたが，実際には似た機能を持つライブラリ同士の比較を行いたい場面が多く存在すると思われる．そのため，ライブラリの比較を効率的に行えるよう，例えば，Parallel Sets における 1 つの階層で複数のライブラリの利用状況を表すなどの改善案が考えられる．

謝辞

本研究において, 常に適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上克郎教授に心より深く感謝いたします.

本研究において, 適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻松下誠准教授に深く感謝いたします.

本研究において, 逐次適切な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻石尾隆助教に深く感謝いたします.

本研究において, 様々な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 Raula Gaikovina Kula 特任助教に深く感謝いたします.

最後に，その他様々な御指導，御助言等を頂いた大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に深く感謝いたします.

参考文献

[1] Yoshida N. Pei Xia, Matsushita M. and Inoue K. Studying reuse of out-dated third-party code in open source projects. コンピュータソフトウェア vol.30, no.4, pp. 98–104, 2013.

[2] Gradle 日本語ドキュメント 51 章. `http://gradle.monochromeroad.com/docs/userguide/dependency_management.html`.

[3] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. Mining trends of library usage. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops*, pp. 57–62, 2009.

[4] Raula Gaikovina Kula, Coen De Roover, Daniel German, Takashi Ishio, and Katsuro Inoue. Visualizing the evolution of systems and their library dependencies. In *Proceedings of the 2nd Working Conference on Software Visualization*, pp. 127–136. IEEE, 2014.

[5] Fabian Bendix, Robert Kosara, and Helwig Hauser. Parallel sets: Visual analysis of categorical data. Proceedings o f the 11th Symposium on Information Visualization, pp. 133–140. IEEE, 2005.

[6] Version 0.777 Kinds of Compatibility OpenJDK Developers' Guide. `http://cr.openjdk.java.net/~darcy/OpenJdkDevGuide/OpenJdkDevelopersGuide.v0.777.html`.

[7] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th International Symposium on Foundations of Software Engineering*, pp. 187–196, 2005.

[8] java.net ASM incompatible changes. `https://weblogs.java.net/blog/kohsuke/archive/2010/02/12/asm-incompatible-changes`.

[9] S. Raemaekers, A. van Deursen, and J. Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation*, pp. 215–224, 2014.

[10] Robert Kosara. Turning a table into a tree: Growing parallel sets into a purposeful project. Beautiful Visualization: Looking at Data through the Eyes of Experts, Steele J., Iliinsky N.,(Eds.). O ' Reilly, pp. 193–204, 2010.