Towards Automated Updates of Software Dependencies

Dhanushka Jayasuriya
University of Auckland
Auckland, New Zealand
djay392@aucklanduni.ac.nz

Abstract

Modern software applications use existing software components to fast forward the development process, resulting in software dependencies. These components will evolve for various reasons and could also be updated to address critical issues existing in the component, like vulnerabilities that could impact the software that depend on them. Therefore, an application's dependencies should be updated to their latest version to avoid these issues. However, changes in a software component can cause critical failures in the dependent software if the new version includes incompatibilities compared to its previous version. These incompatibilities between two library versions are known as breaking changes. Each software system that depends on the component will need to update its code to accommodate these breaking changes, making it difficult to automate the update process.

This research analyzes a set of dependency updates and code changes made in response to fixing breaking changes in Java projects. The breaking changes will be used to create a comprehensive taxonomy of the types of breaking changes that occur in Java projects. The data gathered on the dependency updates will be analyzed to identify patterns that can be applied across different projects to automate the dependency update process.

CCS Concepts: • **Software and its engineering** → **Maintaining software**.

Keywords: dependency, breaking changes, backward compatibility

ACM Reference Format:
Dhanushka Jayasuriya. 2022. Towards Automated Updates of Software Dependencies. In *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '22), December 5–10, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3563768.3565548

SPLASH Companion '22, December 5–10, 2022, Auckland, New Zealand
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9901-2/22/12…$15.00
https://doi.org/10.1145/3563768.3565548

1 Motivation

The use of existing software components is common among most modern software applications [3, 9, 31, 38]. The ease of access to these modules and the ability to accelerate the entire development process when incorporated into the software are the primary motivators for using these existing components [7, 15, 27]. These components can be a package, module, or library and will be referred to as a library throughout this paper.

Software constantly evolves due to new requirements, requirement enhancements, bug fixes, or environment changes-[5]. Software libraries also evolve and release new library versions. Software which depends on a library will refer to a specific version of the library, which is identified as a dependency on the software [31]. Updating these libraries has been convenient using centralized package distribution systems such as Maven Central for Java, PyPI for Python, NPM for JavaScript, and RubyGems for Ruby [15, 27]. Hence when a library is modified to make use of these modifications, the dependent software needs to update the dependency to get the new changes. However, software engineers often do not give attention to maintenance-related tasks such as dependency updates unless it is assigned to them, and most developers are unaware of the threats posed by the vulnerabilities existing in the dependencies [21]. Updating dependencies can be a time-consuming task that can affect a considerable part of the code base and lead to rigorous verification of the entire application, which may cause delays in application delivery [3, 26–28]. If the dependency can be updated by only changing the declared version number, it would be easy. However, if the library version update contains incompatible changes, it takes additional manual effort and time to resolve them [15, 35]. These backward incompatibilities introduced between library versions are known as breaking changes, and the manual intervention needed for fixing these often causes delays in updating software dependencies and also is a barrier in automating the dependency update process [32]. Thus, there is a need to identify the different types of changes that can introduce breaking changes and use this knowledge to discover novel techniques that will enable software systems to automatically and safely accept critical updates, like security vulnerability fixes, even when the new versions include complex breaking changes.

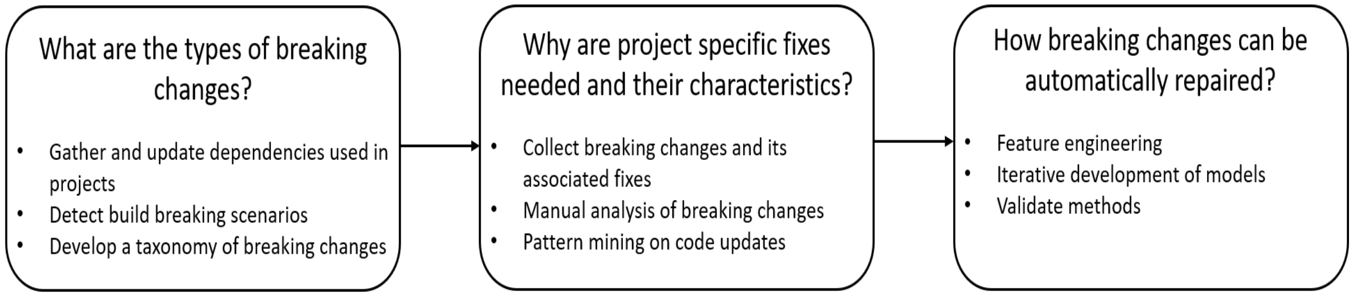
What are the types of breaking changes?
Gather and update dependencies used in projects
Detect build breaking scenarios
Develop a taxonomy of breaking changes
Why are project specific fixes needed and their characteristics?
Collect breaking changes and its associated fixes
Manual analysis of breaking changes
Pattern mining on code updates
How breaking changes can be automatically repaired?
Feature engineering
Iterative development of models
Validate methods

Figure 1. Three step approach for the research

2 Problem

The quickly evolving software industry and the emerging changes in the world itself introduce the need to develop software applications rapidly in an efficient manner. To accommodate this rapid development, many applications tend to use open-source or commercially acquired libraries in the development process. As these libraries evolve and release new versions, downstream projects that rely on them should update to use the new version. However, first, the developers must determine if the latest version of the library is compatible with the system before the update.

One way that libraries can signal whether breaking changes have been introduced in their latest release is to use the semantic versioning convention of X.Y.Z release numbers, which represent whether the updates are Major (X), Minor (Y), or Patch (Z) changes [33]. According to the scheme, Minor and Patch versions should not contain breaking changes, but prior research states that open-source libraries have violated this versioning scheme and introduce breaking changes in Minor and patch releases as well [10, 15, 25]. Because of this, developers commonly avoid using tools to automatically accept new versions of the components they depend on and, therefore, miss critical updates [12, 34].

Breaking changes cause an application using an older version of the library to encounter source, binary, or behavioral incompatibilities when updated to a newer version of the same library [11, 30]. Breaking changes related to source and binary incompatibilities are visible immediately if the application does not compile or build after skipping tests if they existed. Behavioral breaking changes can be detected if the existing test suite fails or if the application compiles fine, but behaves differently than expected. Tools have been created to detect breaking changes introduced between two versions of a library [1, 2, 5, 20, 22, 24, 28, 36] and approaches exist to detect the location in client code that is impacted due to the breaking changes [6, 23, 25, 27, 30, 41]. Research has introduced approaches to automatically fix breaking changes introduced due to binary and source incompatibilities [8, 14, 17, 29, 40], but the main limitation of these tools is that they only focus on breaking changes related to refactoring incompatibilities and do not cover breaking changes due to behavioral incompatibilities. Also, the changes must already be applied in a set of code bases for the patterns to be extracted and replicated in outdated client applications. Therefore, there is a need to identify the behavioral breaking changes that can be introduced between two library versions and how these identifications can be used to automatically apply code changes to avoid these breaking changes in client applications.

3 Approach

To automatically repair breaking changes across a software ecosystem, we must first understand how breaking changes are fixed in software projects and when and why project specific fixes are needed for some breaking changes. We will follow the three steps displayed in Figure 1, which we describe in this section.

3.1 What are the types of breaking changes?

Under this section, we will identify the different types of breaking changes introduced when a dependency is updated in a software project. We will select a set of projects for the analysis and identify the dependencies they are currently using. Then we will detect if there are any outdated dependencies used in these repositories. To identify if updating the dependency to its latest version would cause compiler errors on the repository, we will update the dependency to the latest available version and build the projects with the tests skipped. If there are any compiler errors, it can be due to a source or binary incompatibility. We will rebuild the projects that raised compiler errors in the previous step, with all library versions introduced between the current and the latest version, until the version that introduced the compiler errors is detected. The same steps will be executed to identify the behavioral breaking changes introduced by the library update through building the test suites of the projects.

We will manually analyze the compiler errors and the test suite failures detected to develop a comprehensive taxonomy of the types of breaking changes. While some lists of breaking change types exist (e.g. [5, 13]), they are limited to breaking changes that modify the API which are introduced due to refactoring-related breaking changes. Using our approach we will be able to capture the behavioral breaking changes introduced and if there are any additional breaking changes related to source or binary incompatibilities.

3.2 Why are project-specific fixes needed and identify the characteristics of project-specific fixes

To understand how breaking changes are fixed in software projects and identify the characteristics of these fixes, we will analyze the version control history of the repositories selected in the previous section. We will first identify the commits that contain dependency updates to the projects and gather all changes made under that commit. The follow-up commits to the commit making a dependency update will be helpful if the dependency has introduced any breaking changes and those breaking changes are resolved in the follow-up commits. For this, we will look at the issues and pull requests the commit is linked with. We will gather additional details about the breaking changes by reviewing the component's release notes and upgrade guides and by searching the repository of the associated projects to collect comments made by the developers about the fix (e.g. on the associated commits, pull requests, and issues [18]). Using these data, we will manually analyze projects which are making the same dependency update, detect if they contain breaking changes, and examine the similarity of the fixes applied in the code.

To supplement this manual analysis, pattern mining tools will be used to identify patterns in the fixes across projects. We will manually analyze the identified patterns using thematic analysis to understand the types of changes represented by each pattern and identify the types of project-specific changes made for each type of breaking change [4].

3.3 How breaking changes can be automatically repaired across an ecosystem?

Using the knowledge gained in the previous steps, we hope to identify different types of breaking changes, find their associated fixes, and finally, use this data to build a tool for automation. We will perform feature engineering to translate the identified project and code characteristics and patterns into features that machine learning models can leverage. We will iteratively develop and validate our methods to ensure they are applicable for real-world software projects.

4 Methodology

To select a set of repositories for the analysis, we will use the latest libraries.io dataset [39] and select Java projects using Maven as their build tool. We will select Java projects for the analysis since Java is a statically typed language with many tools for code analysis. Maven build projects were favored since it maintains all configuration-related information, such as dependencies in the pom.xml file, making extracting those easier. We will also select projects on GitHub as the repository could be accessed for further analysis. Also, we will select projects that are not a forked repository and have more than five stargazers to maintain the quality of the repositories selected as suggested in other research [16, 28].

Next, using Maven commands, we will build these repositories and extract the dependencies they currently use and the latest version available for them. Using scripts, we will update the dependencies and use Maven commands to build the project with and without the tests skipped, capturing the compiler errors and test failures. To support the manual analysis in detecting static breaking changes, we will use the JAPICMP [24] tool to list the changes between the two versions of the library needed for the analysis.

For the commit analysis in the repositories, we will use PyDriller [37] to analyze the commits containing changes to the pom.xml file and write scripts to detect if any dependency was updated on those commits. To extract the issues and pull requests linked to the commits we are interested in, we will use the GitHub GraphQL API and web scrapping techniques on the GitHub web UI.

To automatically detect patterns in the code changes associated with a breaking change, we will use FIXMINER [19], which was originally created to mine patterns from bug fix patches. We will provide the set of associated fixes for each breaking change as the input to FIXMINER to generate patterns that would assist in the manual analysis. Finally, using these patterns we will create a tool to automate the dependency updates.

Our hypotheses are: that we can have a taxonomy of breaking changes, and this taxonomy would be helpful to aim for the automated update. Using the approach and the tools mentioned, we hope to identify different types of breaking changes, find their associated fixes, and finally, use this data to build a tool for automation.

5 Conclusion

Currently, we are working on the first section of our approach. We have selected a set of projects for the analysis and gathered its current dependency usage and the latest versions of the dependencies available. Further, we have gathered the set of projects that cause compiler failures when updated with the latest version of the dependency. We are now preparing the data needed for the manual analysis to build the taxonomy of breaking changes.

Our taxonomy will enable future research to automatically repair breaking changes introduced during dependency updates or provide client projects annotated updates on potential breaking changes.

References

[1] 2018. *SignatureTests.* http://wiki.apidesign.org/wiki/SigTest
[2] 2022. *Debian – Details of package japitools in stretch.* https://packages.debian.org/stretch/devel/japitools

[3] Hussein Alrubaye, Mohamed Wiem Mkaouer, Igor Khokhlov, Leon Reznik, Ali Ouni, and Jason Mcgoff. 2020. Learning to recommend third-party library migration opportunities at the API level. *Applied Soft Computing* 90 (06 2020), 106140. https://doi.org/10.1016/j.asoc.2020.106140

[4] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (1 2006), 77–101. https://doi.org/10.1191/1478088706qp063oa

[5] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API breaking changes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 507–511. https://doi.org/10.1109/SANER.2018.8330249

[6] Chow and Notkin. 1996. Semi-automatic update of applications in response to library changes. In *1996 Proceedings of International Conference on Software Maintenance*. 359–368. https://doi.org/10.1109/ICSM.1996.565039

[7] Joel Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. 2015. Measuring Dependency Freshness in Software Systems. In *MOBILESoft 2015 : second ACM International Conference on Mobile Software Engineering and Systems (2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), Vol. 2)*. IEEE Press,, 109–118.

[8] Barthélémy Dagenais and Martin P. Robillard. 2011. Recommending Adaptive Changes for Framework Evolution. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20, 4 (9 2011), 481–490. https://doi.org/10.1145/2000799.2000805

[9] Alexandre Decan and Tom Mens. 2021. What Do Package Dependencies Tell Us about Semantic Versioning? *IEEE Transactions on Software Engineering* 47, 6 (6 2021), 1226–1240. https://doi.org/10.1109/TSE.2019.2918315

[10] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep Me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2187–2200. https://doi.org/10.1145/3133956.3134059

[11] Jens Dietrich, Kamil Jezek, and Premek Brada. 2014. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 64–73. https://doi.org/10.1109/CSMR-WCRE.2014.6747226

[12] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency Versioning in the Wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 349–359. https://doi.org/10.1109/MSR.2019.00061

[13] Danny Dig and Ralph Johnson. 2006. How Do APIs Evolve? A Story of Refactoring: Research Articles. *Journal of software maintenance and evolution: Research and Practice* 18, 2 (3 2006), 83–107. https://doi.org/10.1002/smr.328

[14] Yue Duan, Lian Gao, Jie Hu, and Heng Yin. 2019. Automatic Generation of Non-intrusive Updates for Third-Party Libraries in Android Applications. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 277–292.

[15] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. 2018. Efficient Static Checking of Library Updates. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, 791–796. https://doi.org/10.1145/3236024.3275535

[16] Hao He, Runzhi He, Haiqiao Gu, and Minghui Zhou. 2021. A Large-Scale Empirical Study on Java Library Migrations: Prevalence, Trends, and Rationales *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 478–490. https://doi.org/10.1145/3468264.3468571

[17] J. Henkel and A. Diwan. 2005. CatchUp! Capturing and replaying refactorings to support API evolution. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 274–283. https://doi.org/10.1109/ICSE.2005.1553570

[18] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 92–101. https://doi.org/10.1145/2597073.2597074

[19] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* 25, 3 (3 2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z

[20] Lukas Krejci. 2022. *Revapi*. https://revapi.org/revapi-site/main/index.html

[21] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do Developers Update Their Library Dependencies? *Empirical Software Engineering* 23, 1 (2 2018), 384–417. https://doi.org/10.1007/s10664-017-9521-5

[22] Lars Kühne. 2005. *Clirr*. http://clirr.sourceforge.net/

[23] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–163.

[24] Maven. 2022. *japicmp-base*. https://siom79.github.io/japicmp/

[25] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type Regression Testing to Detect Breaking Changes in Node.js Libraries (Artifact). *Dagstuhl Artifacts Series* 4, 3 (2018), 8:1–8:2. https://doi.org/10.4230/DARTS.4.3.8

[26] Samim Mirhosseini and Chris Parnin. 2017. Can Automated Pull Requests Encourage Software Developers to Upgrade Out-of-Date Dependencies?. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, 84–94.

[27] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. 2020. Detecting Locations in JavaScript Programs Affected by Breaking Library Changes. *Proc. ACM Program. Lang.* 4, OOPSLA (11 2020), 1–25. https://doi.org/10.1145/3428255

[28] Suhaib Mujahid, Rabe Abdalkareem, Emad Shihab, and Shane McIntosh. 2020. Using Others' Tests to Identify Breaking Updates. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 466–476. https://doi.org/10.1145/3379597.3387476

[29] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. 2010. A Graph-Based Approach to API Usage Adaptation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 302–321. https://doi.org/10.1145/1869459.1869486

[30] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. 2021. Breaking Bad? Semantic Versioning and Impact of Breaking Changes in Maven Central. *arXiv preprint arXiv:2110.07889* (2021).

[31] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable Open Source Dependencies: Counting Those That Matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Oulu, Finland) *(ESEM '18)*. Association for Computing Machinery, New York, NY, USA, Article 42, 10 pages. https://doi.org/10.1145/3239235.3268920

[32] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. 2020. A Qualitative Study of Dependency Management and Its Security Implications. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and*

Communications Security (Virtual Event, USA) *(CCS ’20)*. Association for Computing Machinery, New York, NY, USA, 1513–1531. https://doi.org/10.1145/3372297.3417232

[33] Tom Preston-Werner. n.d. *Semantic Versioning 2.0.0*. https://semver.org/

[34] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2014. Semantic Versioning versus Breaking Changes: A Study of the Maven Repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 215–224. https://doi.org/10.1109/SCAM.2014.30

[35] Eric Ruiz, Shaikh Mostafa, and Xiaoyin Wang. 2015. Beyond api signatures: An empirical study on behavioral backward incompatibilities of java software libraries. *Department of Computer Science, University of Texas at San Antonio, Tech. Rep* (2015).

[36] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: Detecting Refactorings in Version Histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 269–279. https://doi.org/10.1109/MSR.2017.14

[37] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. ACM Press, New York, New York, USA, 908–911. https://doi.org/10.1145/3236024.3264598

[38] Inc Synopsys. 2021. 2021 Open Source Security and Risk Analysis Report.

[39] Inc Tidelift. 2022. *Libraries.io - The Open Source Discovery Service*. https://libraries.io/data

[40] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: inference and application of API migration edits. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*, Yann-Gaël Guéhéneuc, Foutse Khomh, and Federica Sarro (Eds.). IEEE / ACM, 335–346. https://doi.org/10.1109/ICPC.2019.00052

[41] Zhaoxu Zhang, Hengcheng Zhu, Ming Wen, Yida Tao, Yepang Liu, and Yingfei Xiong. 2020. How Do Python Framework APIs Evolve? An Exploratory Study. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 81–92. https://doi.org/10.1109/SANER48275.2020.9054800