Vol. V, No. III (Summer 2020)
p- ISSN: 2520-0348
e-ISSN: 2616-793X
ISSN-L: 2520-0348

Global Social Sciences Review (GSSR)
URL: http://dx.doi.org/10.31703/gssr.2020(V-III).19
DOI: 10.31703/gssr.2020(V-III).19
Pages: 174 – 180

Naveed Jhamat[*] | Zeeshan Arshad[†] | Kashif Riaz[‡]

Towards Automatic Updates of Software Dependencies based on Artificial Intelligence

Abstract *Software reusability encourages developers to heavily rely on a variety of third-party libraries and packages, resulting in dependent software products. Often ignored by developers due to the risk of breakage but dependent software have to adopt security and performance updates in their external dependencies. Existing work advocates a shift towards Automatic updation of dependent software code to implement update dependencies. Emerging automatic dependency management tools notify the availability of new updates, detect their impacts on dependent software and identify potential breakages or other vulnerabilities. However, support for automatic source code refactoring to fix potential breaking changes (to the best of my current knowledge) is missing from these tools. This paper presents a prototyping tool, DepRefactor, that assist in the programmed refactoring of software code caused by automatic updating of their dependencies. To measure the accuracy and effectiveness of DepRefactor, we test it on various students project developed in C#.*

Key Words: Automatic Updates of Software Dependencies, Upldate Based on Artificial Intelligence, Automatice Software Dependencies Updation

Introduction

In the software development world, Software reusability is a common practice nowadays. Developers are heavily dependent on techniques and technologies that assist in software code reusability. Software reusability inspires developers to use small reusable software components instead of reinventing the wheel. This result in software that heavily relies on a variety of libraries and packages. There are many third-party companies that provide software libraries and software packages [1]. A software that is built on using these third parties software component is known as dependent software. Any new updation or changes in these third party software requires cascading changes in dependent software products. In most cases, developers are not aware of the fact that their third party libraries or packages are outdated, and developers are using their software products with these outdated dependencies. Undoubtedly, these outdated dependencies should be updated with the newer ones. Often due to ignorance but sometimes intentionally by developers, they are hesitant to update their older outdated dependencies because of the risk of breakage, i.e. software crashing from various points [2]. A dependent software should have to adapt to the changes in their dependencies, and this adaptation is compulsory when new security and performance updates are launched by third-party vendors on their external dependencies in which the software is dependent. Adopting updated dependency is not as easy as it is assumed because adopting new dependencies may require some cascading change in the source code of dependent software. Changes that are required in the dependent software due to updating dependencies is known as migration efforts. These migration efforts are often compulsory in the software in which new dependencies is adopted because of the risk of breaking changes. The impact of adopting new dependencies on the software product is very hard to predict and measure. Developers need to spend a huge amount of effort and time to predict the impact of adopting new dependencies in a software product. Developers are either not aware of newer updates in software dependencies, or their focus is not to keep dependency up to date. Manual checking for new updates, detecting their impact and performing updation in dependent software is

[*]Assistant Professor, Department of Information Technology, University of the Punjab, Gujranwala Campus, Lahore, Pakistan.
[†]Lecturer, Department of Information Technology, University of the Punjab, Gujranwala Campus, Lahore, Pakistan. Email: zeeshan.arshad@pugc.edu.pk
[‡]Department of Computer Science, Government Post Graduate College Satellite Town, Gujranwala, Punjab, Pakistan.

considered to be a costly, error-prone and time-consuming process. Existing work related to software dependency management advocates the need to shift *towards Automatic updation of dependent software* code to implement update dependencies. Various type of software dependency management tools is available as per our literature review in the next section. Existing software dependency tools, although they notify the availability of new updates, detect their impacts on a dependent software system and also identify potential breakages or other vulnerabilities [3]. The methodology adopted by these automatic dependency management tools either uses badges notifications or automatic pull requests (with continuous integration, i.e. CI) to motivate, suggest and assist developers in updating their dependencies. However, support for automatic source code refactoring to fix potential breaking changes is missing from these software dependency management tools.

Literature Review

Reusing third-party dependencies such as libraries and packages is quite common for developers but updating new changes in these dependencies is not as common [3]. Raula et al. [3], in their empirical study on 4600 GitHub projects and 2700 libraries dependencies, indicate that most of these project was heavily dependent on third-party libraries and packages, but most of these libraries were found outdated. According to their finding, they found that these projects were, although heavily dependent on third party or external libraries, but only 81.5% of projects were using outdated libraries. After interviewing various developers of large and medium-sized companies, they [3] found that migration efforts and developers awareness are key factors for considering updates. Migration efforts such as the cost of rework and finding spare time for added responsibility demotivate developers to make positive decisions about updates. The working routine of these developers mainly focuses on vital functional changes without any focus on updating the outdated libraries. Their study also revealed that developers are not using any precautions while updating vulnerable dependencies, and 69% of the developers are even unaware of vulnerabilities in those dependencies. They motivate the research community to propose a strategy that provides useful awareness to developers with a visual aid such as Library Migration Plot for quick update [3]. They conclude that without proper suggesting visual environment for the developer to guide and provide awareness on newer updates, the outdated dependencies will continue to persist in future.

To make developers aware of new updates, notification methodologies adopted by dependency management tools. Among these notification methodologies, most of them are based on either badges or automatic pull request [4]. Badges notifications are shown as red badges in the form of a list. These lists are produced from detecting the outdated software dependencies of that software on which we are supervising. A developer can check this list to know whether any new update is available for its packages or third party libraries. A developer can then click on any given red badge to go to the website from where the newer update can be downloaded. If no new update is available, then the system will show all dependencies as green badges only. On the other hand, automatic pull request, instead of showing badge notifications, automatically detect newer update that is available for the software system under study and then by automatically pull request method download and install the newer update in third party components of the dependent software. However, without human intervention, automatic pull request notification is considered riskier as they did not guide about the potential breakages that may appear in a dependent software system. To measure the usefulness of these two methodologies from the developer's perspective, Mirhosseini and Parnin [4] compared 7470 GitHub projects and found that automatic pull request notification is slightly more effective than badges notification. However, they found that from the collection of automatic pull request notifications, developers merged only one-third of the notifications due to migration efforts or the risk of breaking changes.

Solomon et al. [1] (Apr 2020) advocate that software update release patterns can be used to reduce the risk of breaking software from updates. They suggest that the time of releasing new updates matters quite much and can reduce the risk of breaking changes. Among their suggestions, they suggest that the use of properly documented third party software components is a better option than a non-documented component. A properly documented software component, when releases its new update, make necessary required changes in its documents, these change whether related to the design, coding and implementation of the software, will be updated accordingly in documents. These documented updates in the software component are less risky and easier to adopt as compared to the software component without proper documentation [1]. Similarly, among their other suggestions, they suggest that scheduling your updates around business cycles is more suitable, that is, at the end of the week, end of the month or at the end of the year. According to their findings, the time of releasing update matters, an update that came during the week is hard to implement on weekdays and can cause more problem than the update that implemented during the weekend. In the weekend, developers could find some extra time to recover the breaking changes if it happens. An update that came during the month is hard to implement in the middle of that month and can cause more problem than the update that implemented at the end of the month. At the end of the month, developers could find some extra time to recover the breaking changes if it happens. Similarly, an update that came during the year is hard to implement in the middle months of that year and can cause more problem than the update that implemented at the end of the year. At the end of the year, developers could find some extra time to recover the breaking changes if it happens. From their review on the update, they find that many of the company release update in this pattern mentioned above. Their analysis and review of the update release pattern showed that more than 70% of updates are coming at the end of the weekend, i.e. on Saturday, Sunday than during weekdays. And their analysis and review on update release pattern shown that more than 80% of updates are coming at the end of the month, i.e. in the last week or last days of the month, than in the middle or start of the new month. Similarly, their analysis and review on update release pattern shown that more than 80% of updates are coming at the end of the year, i.e. in November, December than in the middle or start of the year. They also suggest that updating from a more stable version instead of an unstable (beta) version is a safer option [1]. When a newer update came, it may carry some errors and need to be fixed before proper implementation; this beta version, if implemented, immediately carries risks with itself. So, adopting a newer update immediately instead of getting it into a stable version is risky. On the other hand, a more stable update is less risky to implement. Additionally, in a dependent software system, few dependencies, i.e. libraries and third-party packages has more impact and scope in term of breaking changes while other dependencies are having a lesser impact on dependent software [1]. According to their finding, they suggest that first start to update those components that have more dependencies and impact than components with lesser dependencies on the dependent software system. First, updating higher impact dependencies will reduce the risk and effort of breaking changes that may appear, than to implement less impactful dependencies. However, their analysis is based on the oversight of the dependency release and implementation patterns, and they considered that their work could be further improved with more granular analysis of software components and by the use of or by applying machine learning models and algorithms [1].

Among a variety of dependency management tools that exist today, David-DM [5] is a popular tool that checks for outdated node.js dependencies in software projects and generates red badges if outdated dependencies were found otherwise green. However, David-DM did not provide any actionable solution except for badge notifications. Similarly, Vulnerability Alert Service (VAS) [6] also generates alerts for developers after detecting Common Vulnerabilities and Exposures (CVE) found in Maven dependencies but again did not provide solutions to fix these vulnerabilities. Greenkeeper [7] is another dependency management tool that is based on automated pull requests that automatically updates outdated dependency in package management files such as npm or maven but lacking the facility to repair breaking changes. Travis bot CI [8, 9] is a tool that runs continuous integration build and a test suite to check for breaking changes in dependent open-source software before suggesting automatic pull request, but an automatic response to repair breaking changes is not available. From library vendors' perspective, APIDIFF is a tool to detect type, method and field level breaking changes in two library versions of the java project [10]. This tool can be generalized to other languages and from a library client's perspective but provision to fix breaking changes between versions is still absent.

Using third-party libraries and package is a routine matter nowadays for software developers, and literature reveals many dependency management tools developed to assist developers in the last few decades. Recently, Pashchenko et al. [2] (Nov 2020), in their comprehensive quantitative study and dependency management review, suggest that developers strongly consider security risks when using third party libraries and components. They found that developers normally resist updating to newer dependencies because of breaking changes. Existing tools only assist in the detection of breaking changes but not dealing with coping up with these breakages. Due to the unavailability of dependency tools, many developers are forced to perform dependency check on their own, while current dependency tools (if used) only assist in detection with many false positive and low priority frequent alerts [2]. Pashchenko et al. strongly recommended that it is essential to develop dependencies and security analysis tools for the development community that can provide insight into breaking changes with potential solutions [2].

Existing work presents a variety of dependency management tools for developers that provide assistance and guidance about potential breakage and other vulnerabilities in dependent software triggered by updating dependencies. Literature in finding a tool that can assist developers to automatically fix breaking changes is missing. One of the main reason why such a tool is not available that can suggest and fix breaking changes is a large number of potential breakages and type of breakage. A need is there to first categories the similar type of breaking changes into one type and then to suggest a common set of solutions for a single type or kind of breakages. Thus, categorization of breaking changes can assist in suggesting a common set of solution(s) to fix common kinds of breakages.

Table 1. List of Existing Dependency Management Tools

Tool Name	Language	Year	Feature	Lacking
David-DM [5]	Node.js	2012	Detect outdated node.js dependencies	No Solution to update these dependencies
Greenkeeper [7]	Java, Node.js	2014	Automatically update dependencies	No facility to detect and correct breakages
Travis bot CI [8, 9]	Java, Node.js, Php, Ruby, Python etc.	2016	Test for breaking changes & suggest suitable fixes	No support for automatically fix breaking changes
APIDIFF [10]	Java	2018	Detect breaking changes	No provision to automatically fix breaking changes

Impact analysis of breaking changes and problem caused by breaking changes has been discussed [10, 11, and 12], but a classification of breaking changes related to updating dependencies is missing from the literature.

Research Questions

Critical analysis of software dependency management literature and tools spotlight a need for a comprehensive dependency management tool with an ability to fix breaking changes. For the development of such a tool following research questions first need to be answered.

RQ1: What are the adopted methodology, effectiveness & limitations of dependency management tools (if any) that provide solutions to fix potential breaking changes?

RQ2: What are the developer's future expectations from these tools?

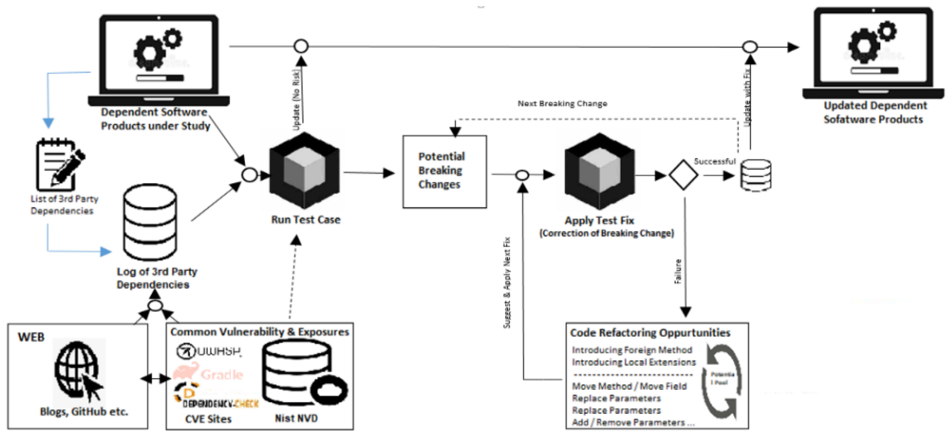
RQ3: What must be the various characteristics on which our dependency management tool must be based on?

Findings

Literature reveals a number of software dependency management tools, but none of these tools wins the developer trust to use frequently in the development market. Developers are reluctant to update frequently because of the risk of breaking changes, and existing tools only guide about the breakages, but none of the tools assists developers to automatically provide solution and then eradicate the breakages without breaking software with success. From the development and developer point of view, and after finding a gap in the literature, we are going to build such a tool that will answer our research questions.

Tool Methodology

Based on software dependency management literature review and by finding the answers to the above questions, we reach the point that existing tools are although have the ability to detect and apply newer updates in dependent software system many of them did not provide the option to show the impact of these updates on the existing system such as breaking changes in the code of the software. While tools that guide about the breaking changes that may be caused by updating dependencies in the software, missing the ability to automatically correct these dependencies. Such tools, although effective to some extent but largely failed to win developers interest and trust because of lacking the ability to suggest suitable refactoring in case of breaking change. Thus, the developer prefers to continue with outdated dependencies but with operational software instead of software with updated dependencies but with a high risk of potential breakage and without any suitable solution to deal with these breakages. Thus, based on the fact and by finding the answers to our research questions, we build a comprehensive dependency management prototyping tool, i.e. DepRefactor, that have the


Dependent Software Products under Study
Update (No Risk)
Next Breaking Change
Update with Fix
Updated Dependent Sofatware Products
List of 3rd Party Dependencies
Log of 3rd Party Dependencies
Run Test Case
Potential Breaking Changes
Apply Test Fix (Correction of Breaking Change)
Successful
Failure
Suggest & Apply Next Fix
WEB
Blogs, GitHub etc.
Common Vulnerability & Exposures
OWASP
Gradle
DEPENDENCY-CHECK
CVE Sites
Nist NVD
Code Refactoring Oppurtunities
Introducing Foreign Method
Introducing Local Extensions
Move Method / Move Field
Replace Parameters
Replace Parameters
Add / Remove Parameters ...
Potential Pool

Methodology Architecture of Dep-Refactor

ability to refine the code of a dependent software system that can otherwise potentially be broken while updating new dependencies. Given below is the proposed Methodological architecture of our tool, i.e. DepRefactor

DepRefactor possesses an internal database that can maintain a list of dependent software products and their associated third-party dependencies to be supervised. The internal database has all the required detail in its attributes about all the packages and third-party libraries related to or found in the software under study. Our database also maintains data of all updates available for various dependencies under consideration, along with its vulnerabilities status. Data about new updates will be obtained from that product websites, or from blogs and from GitHub, while data about CVE (Common Vulnerabilities & Exposures) in newer updates will be obtained from CVE sites and NIST NVD (National Vulnerability Database) [13 & 14]. National vulnerability database accessible through APIs suggests and guide about the potential breakages and associated risk found in newer updated dependency. DepRefactor normally updates software if no serious dependency and vulnerability are found guided by CVE and NVD; otherwise, DepRefactor asks the developer to load the list of potential breakages and vulnerabilities if he/she is still interested in updating or adopting new updates. In case of dependency and risks found in the newer update than before updating the software component, all potential breaking changes will be administered, controlled and resolved by DepRefactor. As mentioned earlier, the CVE site will provide detail about potential breaking changes and based on the nature of potential breaking changes; we can test various refactoring opportunities till a suitable fix. Part of the software code that may have a chance to potentially break after the implementation newer update is first converted in a way that it can stand with the newer software components but keeping in mind that it will not disturb the overall functionality and output of the software at all. DepRefactor did it through refactoring. Refactoring is the art of refining the internal structure or code of the software without disturbing the overall functional behaviour of that software [15]. Refactoring can be applied to the software system where poor coding practices were found, i.e. code smells [16 - 17]. Various type of refactoring opportunities can be suggested for any potential breakage based on the nature of breakage. Introducing foreign methods and introducing local extensions are the two most commonly used Refactoring opportunities to cope up with breakages that appears due to updating a dependency. Move method, remove method, replace method, move a field, remove a field, replace field, replace the object, remove parameters, replace parameters and move parameters are a few of the other refactoring opportunities that we have found quite useful while dealing with breakages that may appear due to updating dependencies [18 & 19]. One or more or then the other refactoring opportunities can be used to test and fit against the potential breakage as a solution. Once the test fixing of all potential breaking changes is successfully completed, we can then apply these fixes to dependent software code while updating software new dependencies. After applying all these refactoring opportunities on the source code of dependent software, we then run the behavioral testing algorithm on the software to measure whether the suggested refactoring opportunities cause any behavioral changes in the functionality of the software system under study. In case if no functional or behavioral change found in the working and output of the software under study, we then confidently move towards the updation of the new update. While updating the new third party components, DepRefactor first keeps a copy of both that is the original system before applying any refactoring and the system after applying various suitable refactoring. In case updating the dependency cause any other unforeseen breakage in the system that DepRefactor is unable to detect or correct, then we can shift back to the older, i.e. original system.

We have tested our tool efficiency, and accuracy on a few students project developed in C# that are using external API and other updates. Our tool not only provides an alert about newer updates that are available from vendors but also provide an environment to cope up with potential breaking changes by suggesting suitable refactoring opportunities associated with each potential breaking change. We find our tool quite effective having an accuracy of 82.7%, while tested on 13 student projects build in C# language.

Future Work

DepRefactor tool has an excellent ability to fix potential breaking changes of software build in C# language while updating automatic external dependencies in them. In future, we will have attention to add a refactoring feature in DepRefactor to support refactoring of software build in other languages such as Java, Node.js, Python, and VB etc. The accuracy of DepRefactor can be improved by testing it on various public software that is larger in size and having more frequent newer updates.

Conclusion

Research literature strongly advocates the essence of a comprehensive dependency management tool that will not only detect the availability and impact of new updates on dependent software accurately but also provide an automatic solution to fix potential breaking changes (migration efforts in source code) triggered by updating dependencies. This proposal, after critically analyzing software dependency literature, suggests a comprehensive dependency management tool with a refactoring based methodology (DepRefactor). DepRefactor tool has the ability to fix potential breaking changes found in software products while updating external dependencies

References

Berhe, S., Maynard, M., & Khomh, F. (2020). Software Release Patterns When is it a good time to update a software component? *Procedia Computer Science*, *170*, 618-625.

Kula, R. G., German, D. M., Ouni, A., Ishio, T., & Inoue, K. (2018). Do developers update their library dependencies? *Empirical Software Engineering*, *23*(1), 384-417.

Cadariu, M., Bouwers, E., Visser, J., & van Deursen, A. (2015, March). Tracking known security vulnerabilities in proprietary software systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (pp. 516-519). IEEE.https://greenkeeper.io/ https://travis-ci.com/

Hilton, M., Tunnell, T., Huang, K., Marinov, D., & Dig, D. (2016, September). Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 426-437). IEEE.

Dackebro, E. (2019). An empirical investigation into problems caused by breaking changes in API evolution.

GitHub. (2020) “Showing available repository results.”, in https://github.com/.

NISTNVD. (2020) “National Vulnerability Database.”, in https://semver.org/.

Tsantalis, N., Mansouri, M., Eshkevari, L., Mazinanian, D., & Dig, D. (2018, May). Accurate and efficient refactoring detection in commit history. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (pp. 483-494). IEEE.

Rasool, G., & Arshad, Z. (2015). A review of code smell mining techniques. *Journal of Software: Evolution and Process*, *27*(11), 867-895.

Rasool, G., & Arshad, Z. (2017). A lightweight approach for detection of code smells. *Arabian Journal for Science and Engineering*, *42*(2), 483-506.

Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D., & Garcia, A. (2017, September). How does refactoring affect internal quality attributes? A multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering* (pp. 74-83).

AlOmar, E. A., Mkaouer, M. W., Ouni, A., & Kessentini, M. (2019, September). On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 1-11). IEEE.

Xavier, L., Brito, A., Hora, A., & Valente, M. T. (2017, February). Historical and impact analysis of API breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 138-147). IEEE.

Mirhosseini, S., & Parnin, C. (2017, October). Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 84-94). IEEE.https://david-dm.org/

Brito, A., Xavier, L., Hora, A., & Valente, M. T. (2018, March). APIDiff: Detecting API breaking changes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 507-511). IEEE.

Pashchenko, I., Vu, D. L., & Massacci, F. (2020, October). A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (pp. 1513-1531).