DGMF: Fast Generation of Comparable, Updatable Dependency Graphs for Software Repositories

Tobias Litzenberger
TU Dortmund
Dortmund, Germany
tobias.litzenberger@tu-dortmund.de

Johannes Düsing
TU Dortmund
Dortmund, Germany
johannes.duesing@tu-dortmund.de

Ben Hermann
TU Dortmund
Dortmund, Germany
ben.hermann@cs.tu-dortmund.de

***Abstract*—Dependency graphs for software repositories have been utilized in a variety of different research contexts. However, to this date there is no unified data model for such graphs, often prompting researchers to implement domain-specific methodologies from scratch. This greatly hinders comparability and makes it hard to incorporate existing tooling into new contexts. With this work we propose DGMF, a framework for mining dependency graphs via repository-specific, user-defined adapters. DGMF is designed to be fast, to require little repository-specific code, and to produce graphs that are comparable even across different repositories. We present our design and implementation, as well as three predefined adapters and an evaluation.**

***Index Terms*—dependency graphs, repository mining, maven, npm, python**

I. Introduction

Reusing software packages from public repositories is an essential part of modern software development processes [1]. Since packages are interconnected via dependencies, the explicit inclusion of a single artifact may transitively include an arbitrary amount of additional packages. All of these packages may become outdated, will receive bug fixes, or be vulnerable to exploits. This imposes an additional layer of complexity on software development processes, where developers now have to understand and monitor both direct and transitive project dependencies. Studies have identified that this imposition results in new threats to software security [2], maintainability [3], and build stability [4].

In order to address those issues, previous work investigated the propagation of vulnerabilities [2] and bugs [5] inside software repositories via *Dependency Graphs*. Such graphs have also been used to investigate large-scale software evolution [6]–[8] and to build tools for program understanding [9], [10].

While for all of these publications researchers construct some form of a dependency graph, there is no common method for collecting and aggregating the necessary data. In fact, we observe that even when publications address the same programming language and repository [2], [8] the implementations, as well as resulting graph data models and overall performance differ greatly. This makes it hard to incorporate existing tools into new contexts and hinders comparative studies, especially involving multiple repositories.

With this work, we improve on the current state-of-the-art regarding dependency graph construction. We propose the *Dependency Graph Mining Framework* (DGMF), a unified framework for implementing data collection and graph building for dependency graphs of arbitrary repositories. We enable fast generation of comparable, updatable dependency graphs only requiring a limited amount of repository-specific code (*adapters*). We contribute:

- A unified graph data model defining the common properties of software artifacts from arbitrary repositories,
- an extendable implementation of the DGMF using a *Neo4j* graph database backend,
- and repository adapter implementations for *Maven Central*, *NPM*, *PyPi* and *Nuget*.

We made our implementations available on GitHub [11] and Zenodo [12].

II. State-of-the-Art

Large- or small-scale dependency graphs have been extensively studied in software engineering research.

Benelallam et al. present a tool called *Maven-Miner* to accumulate and resolve dependency information for software artifacts on Maven Central [8]. The Maven-Miner's data-model is specifically tailored to the structure of Maven Central, and does not generalize to other repositories. While they do not provide concrete numbers, the authors describe executing their tool as *"a time-consuming process"*, which needs to be repeated in full every time a new graph is required.

Düsing and Hermann use dependency graphs to investigate the propagation of vulnerabilities in different software repositories [2]. They implemented a tool that processes artifacts from Maven Central, the NPM Registry and NuGet.org, storing the results in a database. For this, they propose a data model that applies to those three repositories. Running the miner for all repositories took a total of 70 days. The authors point out that their graphs need to be regenerated from scratch in order to obtain up-to-date data.

Decan et al. argue that due to technical and structural differences, a generalization of empirical findings for one repository to another is difficult [13]. The authors derive their claim after investigating *CRAN*, *PyPi* and *NPM*, finding major differences in the size of their packages, number of dependencies and depth of dependency-relations. The authors do not discuss how they aggregated and processed dependency data for their three repositories. They report that *"very few studies have*

compared different software ecosystems", and conclude that *"further studies spanning and comparing multiple ecosystems are required"*.

Generating the dependency graph of *PyPi* was also subject of a 2019 publication by Bommarito and Bommarito [7]. The authors query the HTTP API at `https://pypi.org` for artifact and dependency information, which is then processed and stored in a Postgres SQL database. Their data model is specific to the PyPi repository and does not generalize to other ecosystems.

Other research gathers information on artifact dependencies without building explicit graph representations: Wittern et al. analyze the JavaScript ecosystem by downloading NPM metadata files (`package.json`) [14]. Bavota et al. [15] investigate the dependencies between Apache's Java projects using the *Markos Code Analyzer* [16].

We observe a number of limitations to state-of-the-art dependency graph miners. First, we see that many researchers design data models and implement miners from scratch for their respective research domain. As a result, many implementations for different repositories with incompatible data models exist, making it hard for other researchers to find suitable tooling for a given problem. Furthermore, since existing data models are designed for specific problem instances, results can hardly be compared across different repositories.

Another limitation is the performance of existing miner implementations. Authors report execution times of up to 70 days (for three repositories) [2], while others do not evaluate their miner's performance at all [7], [13]. To the best of our knowledge, no current miner implementations are capable of incrementally updating existing graphs. This implies that long execution times have to be accepted whenever an up-to-date graph is required.

III. Design

To address the limitations observed in state-of-the-art dependency graph miners we propose the *Dependency Graph Mining Framework (DGMF)*. DGMF is capable of generating dependency graphs for any repository via repository-specific *adapters*. While processing, it transforms dependency data to a *unified, repository-independent* data model. Therefore, the resulting graphs can be used for comparative studies across different repositories. DGMF provides *configurable performance* based on the precision required and resources available. Finally, to avoid expensive re-computations, our framework allows *incremental updates* for existing dependency graphs.

A. Data Model

For DGMF, we introduce a repository-independent data model for representing dependency graphs. This is done for two reasons: It enables comparative studies across ecosystems (as suggested by Decan et al. [13]) and eliminates the need for re-designing data models from scratch every time a new repository is analyzed.

Our data model is designed to be an abstraction of software repositories in general. As shown in Figure 1, it is comprised

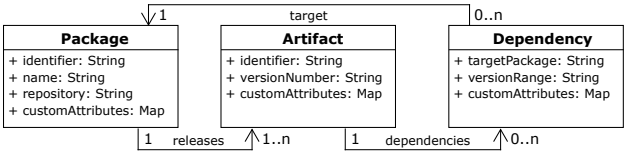

Fig. 1. Classes of the unified data model

of three object classes that represent different entities of software repositories. The role of each entity is defined as follows:

1) A *Package* object represents a particular software library. It is uniquely identified by its name and repository identifier.
2) A single release of a software package is represented by an *Artifact*. Its unique identifier is composed of the respective package identifier and a release version. An artifact may hold several *Dependency* objects.
3) A *Dependency* represents an artifact's dependence on another artifact from the same repository. These objects hold the unique identifier of their respective *Target Package*, as well as a repository-specific textual representation of the range of versions that are addressed via the dependency. We refer to the origin of dependencies as *Source Artifact*.

a) Adapters: DGMF is composed of two kinds of components. General-purpose *Framework Components* encapsulate all functionality related to inputs, the application lifecycle and storage. Repository-specific *Adapters* are responsible for enumerating package names and transforming data into the internal data model. This architecture allows DGMF to work with any repository, as long as there exists a mapping to our internal data model. Therefore, the complexity of developing dependency graph miners for new repositories is reduced.

b) Configurable Performance: In Section II, we observed long execution times for existing dependency graph miners. For DGMF, we address this issue in two ways.

First, we incorporate several general performance optimizations into our design. In particular, we use *placeholder packages* when storing dependencies to packages that have not yet been processed. Furthermore, to increase throughput we use a *parallel streaming architecture* for enumerating, transforming and storing packages.

Second, we provide configuration options for adapting the runtime behavior to a specific use-case. Users may specify the number of parallel transformation streams according to the resources available. Also, DGMF provides different levels of granularity for storing dependencies, which have a large impact on the overall performance. Depending on their requirements, users may choose one of the following levels:

- **Package-to-Package** This level defines dependencies as a relation between two package nodes. If there is at least one artifact of package `A` that depends on package `B`,

than A depends on B. This level is the least complex to generate, but also the least precise.

- **Artifact-to-Package** Here, dependencies are defined between source artifacts and target packages. Resulting dependency edges are annotated with the version range of the dependency. This level preserves all information available, but does not explicitly compute the set of *Target Artifacts* referenced by the version ranges.
- **Artifact-to-Artifact** This level is the most expensive to compute, but also the most precise. Dependencies are represented as edges from one artifact to another. Based on the Artifact-to-Package level, for every dependency we compute the set of target artifacts (resulting from the dependency's version range) and create corresponding edges in the database.

c) Incremental Updates: Current approaches do not build dependency graphs incrementally, but required rebuilding the entire graph from scratch [2], [8]. One reason for this is that the resolution of so-called *floating version ranges* is a time-dependent process, so the addition of new artifacts to a given package may invalidate the resolution of a dependency. Since a full rebuild is a very expensive operation, for DGMF we implemented an algorithm to enable incremental updates. For an existing database, only new artifacts are processed and inserted, while dependency version ranges are re-evaluated where necessary to ensure correctness.

IV. Implementation

We implement DGMF using Java, and provide open access to our implementation via GitHub.

A. Project Structure

As described in Section III, *DGMF* consists of *Framework Components* and repository-specific *Adapters*. An overview is provided in Figure 2, where adapter-components are highlighted in orange.

A `MinerScheduler` controls the application lifecycle. At first, it enumerates all package identifiers for a given repository using a repository-specific `IdGenerator` implementation. Then, package metadata is downloaded and converted to our internal data model via a corresponding `Miner` implementation. Finally, the `DatabaseController` stores all package data in a graph database by converting it into an explicit graph representation. As shown in Figure 2, these transformation and storage operations are executed in parallel.

After the `MinerScheduler` is finished, the graph database contains a dependency graph with either *Package-to-Package* to *Artifact-to-Package* edges. If the *Artifact-to-Artifact* precision level was selected, the `LinkageParser` is invoked to convert all *Artifact-to-Package* edges into the desired precision using a repository-specific `VersionRangeResolver` implementation.

B. Existing Adapters

Our initial implementation of DGMF contains adapters for the *NPM Registry*, the *Python Package Index*, and *Maven Central*. For each repository, we implemented an `IdGenerator` and a `Miner` using respective public HTTP APIs[1][2][3].

Furthermore, we implement a `VersionRangeResolver` for each repository according to the respective version range syntax.

C. Technologies

We haven chosen existing technologies and frameworks for the implementation of DGMF. Most notably we used *Akka Streams* [17] to build a parallel processing pipeline, *Neo4j* [18] as our graph database backend, and *Docker* [19] for deployment.

V. Evaluation

We claim that DGMF (for a given use-case) achieves better performance than state-of-the art implementations (**Claim 1**), can be adopted for other repositories (**Claim 2**), and that our data model is well-suited for comparative ecosystem studies (**Claim 3**). Also, we enable incremental updates that further reduce execution times (**Claim 4**). We evaluate those claims in the following. All analyses were performed between December 15, 2022 and January 20, 2023.

A. General Metrics

In order to evaluate *Claim 1*, we generated all three dependency graphs, which took about 25.85 hours on a server with *Intel Xeon E5-2650* quad-core CPU and 34 GB of RAM. As shown in Table I, the resulting graphs combined contain about 3,598,058 packages and 43,529,670 artifacts, with 27,860,704 dependencies. We observe an improvement in performance compared to other miner implementations [2] (Claim 1).

TABLE I
General Metrics of Repositories

Repo	Packages	Artifacts	Dependencies	Average Time
NPM	2,678,549	29,662,399	23,778,374	1,247.11 min
PyPi	410,559	4,023,847	922,481	44.09 min
Maven	508,950	9,843,424	3,159,849	259.74 min
$\sum$	3,598,058	43,529,670	27,860,704	1,550.94 min

To test the extensibility of the framework, we tasked a Java developer to implement the functionality needed to mine packages for the *Nuget*[4] repository using only our documentation on GitHub. Since tasks such as database management, parallelization, and data modeling are already solved by the framework, implementing and testing the necessary repository adapter took less than four hours. This indicates that DGMF can be adopted for other repositories (*Claim 2*) with reasonable implementation effort.

[1] https://replicate.npmjs.com
[2] https://pypi.org/simple/
[3] https://maven.apache.org/repository/central-index.html
[4] https://www.nuget.org

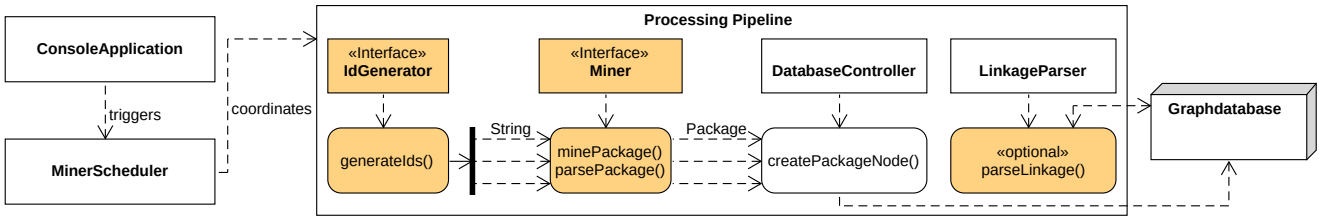

Fig. 2. Architecture, Components and Dataflows of DGMF

B. Dependency Graph Analysis

To evaluate *Claim 3*, we conduct an exemplary comparative study on the transitive dependency distributions for NPM, PyPi, and Maven. Figure 3 shows the distribution of *transitive* outgoing dependency edges is shown with a maximum depth of 6. As an example conclusion, our data shows that more than 75% of the packages in Maven depend on more than 5 other packages, compared to the 33% of PyPi packages. The data was obtained from the Neo4j database with one query per repository.

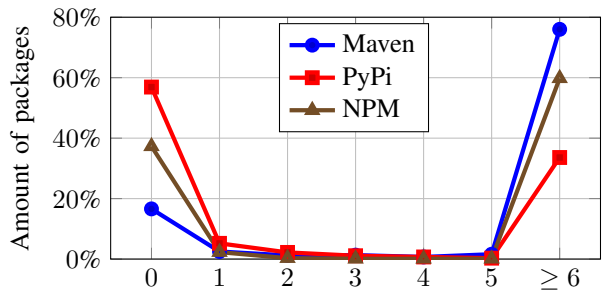

Fig. 3. Distribution of Transitive Outgoing Dependency Counts per Package

In addition, we evaluate the performance of different levels of dependency resolution (see section III-A0b). Our results are presented in Table II and clearly show that more accurate resolution leads to longer generation times. The different levels of resolution for dependencies can be used for different use cases of the analysis framework.

TABLE II
DEPENDENCY COUNT AND PERFORMANCE PER PRECISION LEVEL

Repo	Package-to-Package		Artifact-to-Package		Artifact-to-Artifact	
NPM	23,778,374	20.8h	456,536,545	43.0h	343,382,828	243.9h
PyPi	922,481	0.7h	25,358,350	2.6h	1,684,941,345	135.7h
Maven	3,159,849	4.3h	66,230,554	9.1h	82,916,746	24.5h
$\sum$	27,860,704	25.8h	548,125,449	54.7h	2,111,240,919	404.1h

C. Incremental Updates

To validate that incremental updates indeed yield performance benefits (Claim 4), we conduct the following experiment: For each repository, we first generate a partial dependency graph containing 65% of all packages. We then update the partial graph to contain all packages using DGMF's incremental updates. We use the *Artifact-to-Package* level for this experiment. Table III reports the resulting execution times, as well as the update speedup compared to a full rebuild.

TABLE III
DURATION OF INCREMENTAL UPDATES AND SPEEDUPS

Repo	Full Build	Partial Build	Full Update	Speedup
NPM	43.0h	25.25h	23.80h	1.81
PyPi	2.6h	1.54h	1.22h	2.13
Maven	9.1h	6.45h	2.84h	3.20

We observe that for all three repositories incremental updates are faster than full rebuilds, with speedups ranging from 1.81 to 3.20. For this setting, updates yield net savings in execution time of up to 19.2 hours (NPM).

D. Limitations

Our adapter implementations exhibit some technical limitations. We observe that in some cases the repository APIs return no responses for package ids. We call these instances *Request Errors*. The number of *Request Errors* varied from 0 (PyPi) to 397,852 (NPM). They occur due to packages that do not exist anymore, either because they have been revoked or because of outdated indices. As the respective package data does not exist, these errors do not invalidate the resulting graphs.

We further note that our *PyPi* adapter implementation is limited to the detection of dependencies specified via `requires-dist` [20]. We chose to not process dependencies specified in `setup.py` files, as this would require downloading and unzipping archives for every single artifact, thus dramatically reducing throughput. If desired, the corresponding PyPi adapter can be extended accordingly.

VI. CONCLUSION

In this paper we presented DGMF, a framework for efficiently generating comparable dependency graphs for software repositories. DGMF improves on the current state-of-the-art by introducing a unified data model for comparative ecosystem studies, by improving overall performance, and by being open for extensions in order to cover new repositories. We provide adapter implementations for Maven Central, the NPM Registry and the Python Package Index, and demonstrate that additional adapters can be added with minimal effort. Finally, we evaluate the performance of DGMF, and provide insights into the resulting graphs. Our implementations are freely available on GitHub [11] and Zenodo [12].

REFERENCES

[1] J. L. Barros-Justo, F. Pinciroli, S. Matalonga, and N. Martínez-Araujo, "What software reuse benefits have been transferred to the industry? a systematic mapping study," *Information and Software Technology*, vol. 103, pp. 1–21, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918301083

[2] J. Düsing and B. Hermann, "Analyzing the direct and transitive impact of vulnerabilities onto different artifact repositories," *Digital Threats*, jun 2021, just Accepted. [Online]. Available: https://doi.org/10.1145/3472811

[3] N. B. Tàrrega, M. Zivkovic, A. Oprescu, and S. PCS, "Measuring the impact of library dependency on maintenance." in *SATToSE*, 2020.

[4] C. Macho, S. McIntosh, and M. Pinzger, "Automatically repairing dependency-related build breakage," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 106–117.

[5] W. Ma, L. Chen, X. Zhang, Y. Feng, Z. Xu, Z. Chen, Y. Zhou, and B. Xu, "Impact analysis of cross-project bugs on software ecosystems," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 100–111.

[6] V. Musco, M. Monperrus, and P. Preux, "A generative model of software dependency graphs to better understand software evolution," 2014. [Online]. Available: https://arxiv.org/abs/1410.7921

[7] E. Bommarito and M. Bommarito, "An empirical analysis of the python package index (pypi)," 2019. [Online]. Available: https://arxiv.org/abs/1907.11073

[8] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The maven dependency graph: A temporal graph-based representation of maven central," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 344–348.

[9] S. Venkatanarayanan, J. Dietrich, C. Anslow, and P. Lam, "Vizapi: Visualizing interactions between java libraries and clients," 2022.

[10] R. Falke, R. Klein, R. Koschke, and J. Quante, "The dominance tree in visualizing software dependencies," in *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2005, pp. 1–6.

[11] T. Litzenberger, J. Düsing, and B. Hermann, "Dgmf on github," https://github.com/sse-labs/dgmf, Jan 2023.

[12] T. Litzenberger, J. Duesing, and B. Hermann, "Dependency graph mining framework v1.0.0," https://doi.org/10.5281/zenodo.7561081, Jan 2023.

[13] A. Decan, T. Mens, and M. Claes, "On the topology of package dependency networks: A comparison of three programming language ecosystems," in *Proccedings of the 10th European Conference on Software Architecture Workshops*, ser. ECSAW '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2993412.3003382

[14] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 351–361. [Online]. Available: https://doi.org/10.1145/2901739.2901743

[15] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: The case of apache," in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 280–289.

[16] G. Bavota, A. Ciemniewska, I. Chulani, A. De Nigro, M. Di Penta, D. Galletti, R. Galoppini, T. F. Gordon, P. Kedziora, I. Lener, F. Torelli, R. Pratola, J. Pukacki, Y. Rebahi, and S. G. Villalonga, "The market for open source: An intelligent virtual open source marketplace," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 399–402.

[17] Lightbend, Inc., "Akka documentation - streams," https://doc.akka.io/docs/akka/current/stream/index.html, accessed: 11-16-2022.

[18] Neo4j, Inc., "Neo4j graph database," https://neo4j.com/product/neo4j-graph-database/, accessed: 11-16-2022.

[19] Docker Inc., "Docker," https://www.docker.com/why-docker, accessed: 11-16-2022.

[20] Python Software Foundation, "Python core metadata specifications - requires-dist," https://packaging.python.org/en/latest/specifications/core-metadata/#requires-dist-multiple-use, accessed: 11-14-2022.