Goblin: A Framework for Enriching and Querying the Maven Central Dependency Graph

Damien Jaime, Joyce El Haddad, Pascal Poizat

GOBLIN: A Framework for Enriching and Querying the Maven Central Dependency Graph

Damien Jaime
Sorbonne Université, CNRS, LIP6
F-75005, Paris, France
SAP France S.A
F-92300, Levallois-Perret, France
damien.jaime@lip6.fr

Joyce El Haddad
Université Paris Dauphine-PSL, CNRS, LAMSADE
F-75016, Paris, France
joyce.elhaddad@lamsade.dauphine.fr

Pascal Poizat
Sorbonne Université, CNRS, LIP6
F-75005, Paris, France
Université Paris Lumières, Université Paris Nanterre
F-92000, Nanterre, France
pascal.poizat@lip6.fr

ABSTRACT

Dependency graphs support software maintenance and software ecosystem analysis. Several metrics can be used on top of these graph models but the set of such metrics is to evolve over time. Further, some metrics have a dynamic nature, requiring being able to "rewind" dependency graphs at some point in time. To address these issues we propose the GOBLIN framework. It is composed of a dependency graph metamodel with time-related information, a miner to retrieve the graph from Maven Central, and a tool for on-demand metric weaving into dependency graphs. As a whole, GOBLIN is a customizable framework for ecosystem and dependency analysis. This is illustrated with a set of complementary experiments. Our tools, datasets, and experiments are freely available online.

KEYWORDS

software ecosystem, dependency graph, framework, dataset, mining software repositories, maven central

ACM Reference Format:
Damien Jaime, Joyce El Haddad, and Pascal Poizat. 2024. GOBLIN: A Framework for Enriching and Querying the Maven Central Dependency Graph. In *Proceedings of 21st International Conference on Mining Software Repositories (MSR 2024).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Software reuse reduces development time and improves software quality [1]. Using package managers, it is simple to reuse code as project dependencies. Yet, these direct dependencies may themselves depend on other packages, yielding indirect dependencies. It may then become complex to get a grasp of the whole set of dependencies of a project. Further, dependency interplay, such as incompatibilities between versions, hamper maintenance when it comes to updating dependencies.

MSR 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

To help, one may rely on dependency graphs (DG). We list and compare several DG metamodels in the next section. Their common base is the representation of artifacts, releases (versions), and dependency relations. On top of these models, it is needed to compute different metrics before being able to measure the quality of a project (from a dependency perspective) and to support or even suggest updates. There are several metrics of interest here. The set of CVEs (Common Vulnerabilities and Exposures) concerning a dependency is central for security. Other metrics incorporate time, *e.g.*, freshness [3] or rhythm [5].

These metrics could be part of a DG metamodel. Yet, they are numerous and evolve over time (and research). We believe that a better solution is to *weave* them on-demand over DG models. Some of these metrics have a dynamic nature, *e.g.*, CVEs, freshness, and rhythm, are all not constant in time given some package. Dependencies may support ranges (instead of requiring version `g:a:1.0` of a dependency, one may require any version between `1.0` and `2.0`, not included). The version of a package that ends up being used at some point in time may then change with different releases being made. This means that it should be possible to "rewind" a DG model at some point in the past.

As a solution to these requirements, we propose a DG framework, GOBLIN.[1] We apply GOBLIN to the Java Maven Central ecosystem to demonstrate its use and because it is part of a requirement for the analysis and update of DG models of our enterprise partner. GOBLIN includes: a DG metamodel (GOBLIN-DG), a miner (GOBLIN-Miner) to generate the whole DG model for Maven Central, and an on-demand metrics weaver (GOBLIN-Weaver) that supports both project-related scenarios and ecosystem-wide analysis.

In Section 2, we introduce GOBLIN-DG and compare it to existing datasets. Section 3 presents the architecture of GOBLIN, GOBLIN-Miner, and GOBLIN-Weaver. Section 4 aims to demonstrate the use of GOBLIN on typical DG-related applications.

2 MAVEN CENTRAL DATASETS COMPARISON

This section aims to provide a comparative list of Maven Central DG datasets, including ours. The first part of Table 1 is relative to availability and reproducibility. All datasets are available except for one for which only the generation code is. We were able to regenerate two of the datasets: ours on October 5th, 2023, and the one from [10] on October 6th, 2023, with the "Artifact-to-Package" configuration to get a graph architecture comparable to ours. The former (ours)

[1] Our framework, which weaves metrics into DG models, is named after Manufacture des Gobelins, which produced weaved goods for the kings.

Table 1: Comparison of Maven Central dependency graph datasets

Ref.	External features					Internal features							
	Link (dataset / code)	Date (of dataset)	Can be recomputed	Computation time (at 10/2023)	Can be incremented	Node types	Edge types: Dependency	Edge types: Version	Available additional information (for nodes or for edges): Scopes	Ranges	Ghosts	Release Date	Access to Latest R
[11]	data.4tu.nl (dataset)	2011-07-30	–	n/a	–	R/P/C/M	R→R	R→R	–	–	✓	–	indirect
[2]	zenodo.org (dataset)	2019-09-10	✓	unknown	–	R	R→R	R→R	✓	–	–	✓	indirect
[4]	zenodo.org (dataset)	2022-09-20	–	n/a	–	R	R→R	–	✓	–	–	–	indirect
[10]	github.com (code)	n/a	✓	6.8 hours	✓	A/R	R→A	A→R	–	–	–	–	direct (A→R)
This	zenodo.org (dataset)	2023-10-05	✓	4.1 days	✓	A/R	R→A	A→R	✓	✓	✓	✓	indirect

contains 588,185 artifacts, 12,154,070 releases and 98,190,646 dependencies. To be compared to 606,608 artifacts, 11,771,269 releases and 77,622,145 dependencies for the latter. The difference is probably due to the retrieval methods. There is however a notable difference in computation time (measured on a Red Hat Enterprise Linux 8.7, 64BG memory, 16 CPUs Intel(R) Xeon(R) CPU E7-8880 v4 @ 2.20GHz, machine, with 12 allocated threads). Yet, Goblin-miner has the ability to update a dataset without regenerating it from scratch, which makes it more amenable to regular use. For instance, to update our dataset from October 05, 2023, to October 14, 2023, 1h06m was necessary to add 50,427 new releases. From April 14, 2023, to October 14, 2023, (six months), 6h23m to add 1,179,148 new releases. From October 14, 2022, to October 14, 2023, (one year), 11h27m to add 2,378,414 new releases.

The second part of the table describes the graph's structure and the information it holds. We identify several node types: Artifacts (A, with group:artifact information), Releases (R, with version information), Packages (P), Classes (C), and Methods (M). All datasets include dependency edges, but their representation varies, either release-to-release (R→R) or release-to-artifact (R→A). We may also find version-related edges, either release-to-release (*i.e.*, a next relation) or artifact-to-releases (*i.e.*, versions of an artifact). Additionally, other pieces of information can be present: scopes (*e.g.*, compile or test), ranges (version range specifications), ghost dependency information (*i.e.*, dependencies to an artifact not found on the ecosystem), release dates, and methods to determine an artifact's latest version (available either directly or through edges navigation).

The dataset from [11] stands out with richer information about classes and methods, but it is older and its generation code is unavailable. The other four datasets vary based on their inclusion of only releases, or both artifacts and releases. We opted for both, with R→A dependencies, for greater flexibility. In addition to incorporating target versions (including ranges) on dependency edges, this enables us to accurately represent dependency requirements. It should be noted that the DG metamodel from [10] is the only one with genericity mechanisms (multiple ecosystems, multiple formats: Package-to-Package, Artifact-to-Package and Artifact-to-Artifact).

Our DG metamodel is presented in Figure 1. It is implemented as a Neo4J graph.[2] The release timestamps enable us to track time-based changes and "rewind" in time (both being among our objectives). Dependency contexts (including ranges) are also important for this. Scopes help in filtering out non-essential dependencies (*e.g.*, tests). Unlike the other datasets, ours combines all these three critical pieces of information.

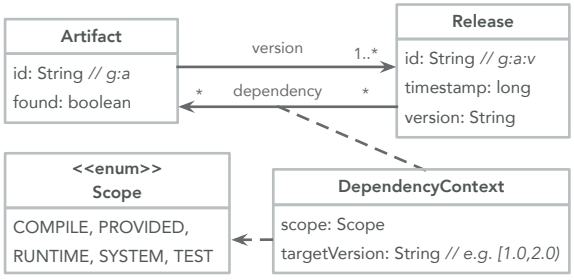

Figure 1: Goblin dependency graph metamodel

3 THE GOBLIN FRAMEWORK

In this section, we present the architecture, the main tools, and the use of Goblin. This is illustrated in Figure 2. The Goblin framework is composed of a DG metamodel (Sect. 2), a miner for DG dataset generation and updates (Sect. 3.1), and an on-demand weaver that enables one to both add metrics to a DG and query it (Sect. 3.2).

3.1 Goblin-Miner

To create our dataset, we developed a Java program that builds the DG in two steps. First, we retrieve all releases in the Lucene

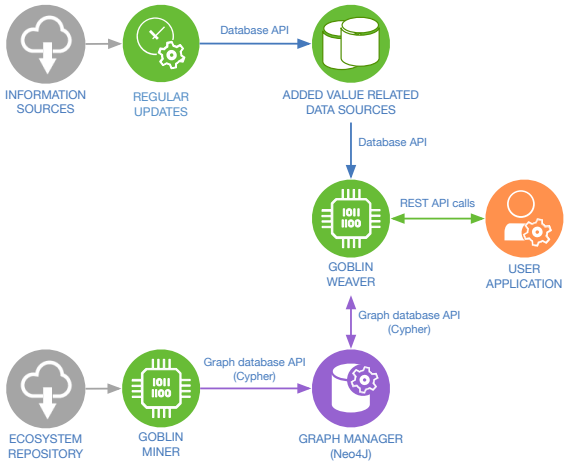

Figure 2: Architecture of the Goblin framework

[2]Hence the PrimitiveObsession† use of Strings, and long for timestamps.

Maven Central Index archive[3], extract the data from it, and create the artifact and the release nodes, together with the version edges, in a Neo4J database. Then, we go through all releases to retrieve their direct dependencies with the `org.eclipse.aether` library. Updating our dataset works the same way: we download the latest Lucene archive from Maven Central and carry out the same process, but only for releases with a timestamp higher than the highest timestamp present in the DG. Goblin is, by now, focused on Maven and Java. Yet, our DG structure can be used for other ecosystems, and our framework has been designed to be extensible.

3.2 Goblin-Weaver

When working on a DG, *e.g.*, for dependency quality assessment, library replacement, or ecosystem analysis, it is required to add different kinds of information to nodes and/or edges. These can be of a very different nature. Let us take the example of CVEs that one can associate with nodes. This metric can be computed either *locally* (the CVEs of a release) or *aggregated* (the CVEs of a release and of all its direct and indirect dependencies). It can also be computed *at some point in time* or derived *over a period of time*. It is the role of the weaver to compute such metrics and add them on-demand since it is not possible to have all of them in the DG (too many of them, dynamic nature).

The way to call the weaver is using the REST API route:

```
POST /cypher {"query":QUERY,"addedValues":[(added value)*]}
```

where `QUERY` is a Neo4J's Cypher query. In response:

- the weaver calls the Neo4J graph database where the DG is stored, using the Cypher query given by the user;
- for each of the objects on which an added value applies (this can be either nodes or edges), it computes an added value for it, possibly using a specific database and performing new Cypher queries transparently to the user;
- it returns the result in JSON format.

Several added-value algorithms are already available:

- direct CVEs (`CVE`) and aggregated ones (`CVE_AGGREGATED`), using a local copy of the OSV dataset[4];
- freshness [3], either local (`FRESHNESS`) or aggregated for direct and indirect dependencies (`FRESHNESS_AGGREGATED`);
- speed [5] (`SPEED`).

The weaver is designed to be extensible, allowing a user to easily compute information specific to their research needs. As an example (more experiments are given in Sect. 4), let us take the `log4j-core` artifact. A query to the weaver that returns all 57 releases of it, with no added value, takes 12ms. With added values we get 14ms for `[CVE]`, 3s for `[CVE_AGGREGATED]`, 158ms for `[FRESHNESS]`, and 13s for `[FRESHNESS_AGGREGATED]`. The computation of aggregated values takes more time, which is normal since the weaver has to compute the values for all the direct and indirect dependencies of the target release before performing aggregation.

Since some Cypher queries are recurrent, one also has the possibility to query "shortcut" routes (with specific parameters). Such queries are then translated to queries on the `/cypher` route. A Swagger documentation is available for the weaver.

[3]https://maven.apache.org/repository/central-index.html
[4]https://osv.dev/

3.3 Using Goblin

In order to use Goblin a user has to perform the following steps.

Extension (optional). If the set of added value constructors does not meet one's needs, it is possible to create new ones. To achieve this, one has to develop the new constructor logic in a class that implements a specific interface and registers it to the weaver. It is also possible to define new routes for Cypher requests that are often used. This is the approach we followed when defining the existing set of routes. Our goal is that, over time, users contribute new added values and new routes so that the burden of DG analysis and use mainly lies in the post-processing step.

Calling the weaver. This consists of asking the weaver to query the Neo4J database and to include some added values at the same time. We recall that the general form of REST API calls for this is:

```
POST route {(param:value)*,"addedValues":[(added value)*]}
```

with different routes (including the low-level `/cypher` one) having different sets of parameters.

Post-processing (optional). Once the information (*i.e.*, a subpart of the DG extended with added values) is retrieved, the user may perform post-processing. This can be done in any language since only calling a REST API and analyzing JSON feedback is required. For the experiments, below in Section 4, we used Java and Python.

4 EXPERIMENTS

This section provides examples of how to use Goblin. We cover scenarios at different target levels: ecosystem, library and project. We also cover the use of both absolute time and time-derived metrics.

4.1 Experiment 1 - Ecosystem analysis

To test the scalability of Goblin, this first experiment is positioned at the level of the entire ecosystem. First, we ask "*How many of the 12,154,070 releases on Maven Central contain at least one CVE?*". To answer, we use route `/cypher` with the following payload:

```
{   "query": "MATCH (r:Release) RETURN r",
    "addedValues": ["CVE"] }
```

The weaver returned all release nodes with their CVE information in 93s. We then did a post-processing to count the number of releases containing one or more CVEs. The result showed that 0.462% of releases (56,198 out of 12,154,070) had at least one CVE. As the weaver returns all node information (including release date), we created a histogram showing the year-by-year evolution in the number of CVEs of releases in the ecosystem, as shown in Figure 3.

A little more tricky now, we are going to ask "*How many of the latest releases of each artifact contain at least one CVE?*". To answer, we modified the Cypher query to get, for each artifact, its last release based on the timestamp as follows:

```
{   "query": "MATCH (a:Artifact) WITH a MATCH
    (a)-[:relationship_AR]->(r:Release) WITH a
    , r ORDER BY r.timestamp DESC WITH a,
    COLLECT(r)[0] AS mostRecentRelease RETURN
    mostRecentRelease",
    "addedValues": ["CVE"] }
```

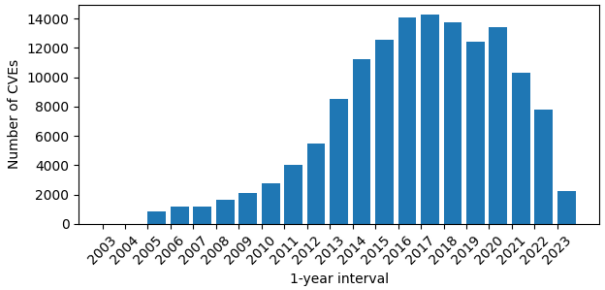

Figure 3: Evolution of vulnerability in Maven Central

The result, obtained in 49s, showed that 0.031% of last releases (176 out of 567,315) had at least one CVE. Note here that this request, more complex, could have been an API route.

4.2 Experiment 2 - Choice of a library

We operate now at library level, with time-derived metrics, asking "*How to choose between two libraries with the same features?*". We compared two popular JSON processing libraries (Jackson-databind and Gson) by examining their aggregated CVEs and the number of their dependents in Maven Central for their last 20 releases. We first gathered these most recent 20 releases of each artifact with their aggregated CVEs, excluding release candidates, using a query of the form:

```
/cypher {"query":"...","addedValues":["CVE_AGGREGATED"]}
```
×2

Execution takes 337ms for Jackson-databind and 140ms for Gson. Now for each release `Ri`, we ask for its dependents using:

```
/release/dependents {"gav":"Ri","addedValues":[]}
```
×2×20

It takes respectively 10s (for the 20 Jackson-databind releases) and 17s (for the 20 Gson releases). Finally, we post-process the result to get Figure 4 which displays the evolution of aggregated CVE and dependents for both packages.

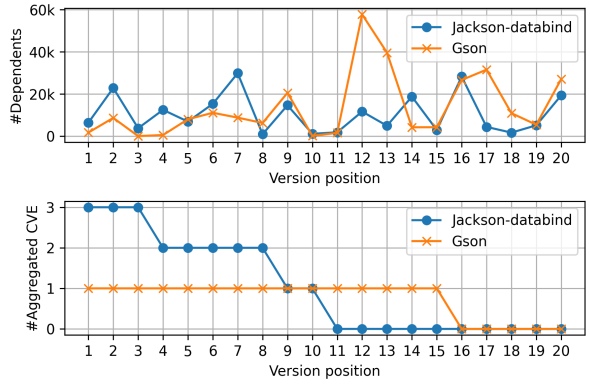

Figure 4: JSON libraries Comparison

4.3 Experiment 3 - Relative release rhythm

4.3.1 Client side. Let us now operate at project level, with time-derived metrics again, by asking "*Based on the direct dependencies of a project, what would be the best rhythm to update them all at once?*". This can assist developers in planning their dependency management strategies and anticipating potential updates or changes.

To answer this, we retrieve the direct dependencies present in a `pom.xml` file and ask the weaver to provide us with the corresponding artifacts (`GAi`) and their speed (number of releases per day) [5] by using:

```
/artifact {"ga":"GAi","addedValues":["SPEED"]}
```
×2

Using this speed information, we may display the average, minimum, and maximum release rhythm of their dependencies. For example, for a project with two direct dependencies, the execution time was 64ms and the average release time for new versions of the project's dependencies was 21.88 days. It would hence be healthy to check the dependencies of this project every 22 days.

4.3.2 Provider side. Here, we are also at project level, with a time-derived metric, by asking "*In the viewpoint of a package provider, what would be the best rhythm to release a new version?*". We start by identifying the provider `P` speed by using:

```
/artifact {"ga":"P","addedValues":["SPEED"]}
```

and then the speed of all the provider dependents `Di` by using:

```
/cypher {"query":"...Di...","addedValues":["SPEED"]}
```
×4883

For this experiment, we examined the `httpcore` artifact, which is in the top 100 in popularity on Maven Central with 4,883 dependents. Execution time was 16s, the provider speed was 112 days while on average its clients' speed was 65.08 days.

5 AVAILABILITY

Our Goblin-DG dataset for Maven Central is available at [6]. Our tools are available at [7] (Goblin-miner) and at [8] (Goblin-weaver). Our experiments repository is at [9] with results at [6].

6 CONCLUSION

In this paper we have presented the Goblin framework. It includes a DG metamodel, an ecosystem miner, and an on-demand metrics weaver. Goblin has been applied to the Maven Central ecosystem. In order to demonstrate the use of Goblin, we have designed several experiments corresponding to typical use cases: ecosystem analysis, library comparison, and client-provider rhythm analysis.

The primary limitation of Goblin is the cost of computing transitive values. While this is not a major issue for individual projects or libraries, it becomes significant when analyzing an entire ecosystem. We are studying possible optimizations such as memoization and transitive edge pre-computation. A work in progress is to apply Goblin to other software ecosystems. NPM is required at our enterprise partner. The study of dependency ranges will also be more of interest there (this feature is seldom used in Maven Central).

ACKNOWLEDGMENTS

This work is funded by PhD grant 2021/0047 from ANRT. Experiments have been achieved thanks to machines at SAP France S.A.

REFERENCES

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why Do Developers Use Trivial Packages? An Empirical Case Study on Npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. 385–395.

[2] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. 2019. The Maven Dependency Graph: A Temporal Graph-Based Representation of Maven Central. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*. 344–348.

[3] Joël Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. 2015. Measuring Dependency Freshness in Software Systems. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15, Vol. 2)*. 109–118.

[4] Andreas Dann, Ben Hermann, and Eric Bodden. 2023. UpCy: Safely Updating Outdated Dependencies. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. 233–244.

[5] Damien Jaime, Joyce El Haddad, and Pascal Poizat. 2022. A Preliminary Study of Rhythm and Speed in the Maven Ecosystem. In *21st Belgium-Netherlands Software Evolution Workshop (BENEVOL '22)*.

[6] Damien Jaime, Joyce El Haddad, and Pascal Poizat. 2023. Goblin: A Framework for Enriching and Querying the Maven Central Dependency Graph. https://doi.org/10.5281/zenodo.10306054

[7] Damien Jaime, Joyce El Haddad, and Pascal Poizat. 2023. *Goblin Ecosystem Dependencies Miner*. https://github.com/Goblin-Ecosystem/goblinDependencyMiner

[8] Damien Jaime, Joyce El Haddad, and Pascal Poizat. 2023. *Goblin-Weaver*. https://github.com/Goblin-Ecosystem/goblinWeaver

[9] Damien Jaime, Joyce El Haddad, and Pascal Poizat. 2023. *Maven Dataset And Weaver Experience*. https://github.com/Goblin-Ecosystem/mavenDatasetExperiences

[10] Tobias Litzenberger, Johannes Düsing, and Ben Hermann. 2023. DGMF: Fast Generation of Comparable, Updatable Dependency Graphs for Software Repositories. In *20th International Conference on Mining Software Repositories (MSR '23)*. 115–119.

[11] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2013. The Maven repository dataset of metrics, changes, and dependencies. In *10th Working Conference on Mining Software Repositories (MSR '13)*. 221–224.

Received XX January XXXX; revised XX January XXXX; accepted XX January XXXX