MARIN: A Research-Centric Interface for Querying Software Artifacts on Maven Repositories

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

Jared Chiaramonte
Arizona State University
Tempe, USA
jchiara1@asu.edu

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

***Abstract*—Maven Central is the largest open repository for JVM libraries, hosting just under 15 million artifacts as of November 2024. Its popularity has made it a prime target for malicious actors to upload malware or exploit vulnerabilities – one in eight open source downloads have been vulnerable in 2023. Consequently, analyzing the artifacts is essential to understanding and improving software security and safety, both for individual projects and on a large-scale.**

However, current implementations of concrete analyses do not separate the infrastructural task of iterating and accessing artifacts from their domain-specific analysis task. Consequently, features are implemented many times in different variations, increasing the potential for bugs as well as the overhead in development and maintenance.

With this work we propose MARIN, a framework for conducting analyses targeting software hosted on Maven Central. MARIN handles common infrastructural tasks in such scenarios, including iterating artifacts, retrieving metadata, parsing binaries, and resolving dependencies. It is designed to have minimal performance overhead, using both internal caches and the local Maven repository to reduce the number of HTTP calls and computations. This way, researchers can solely focus on implementing their domain-specific analysis task – MARIN provides configurable facilities to execute it for all artifacts on Maven Central.

***Index Terms*—Static Analysis, Repository Mining, Maven Central, Large-Scale Analysis**

I. Introduction

Public software component repositories like Maven Central [1] are a crucial part of the software development process. For many projects, third-party code from such repositories constitutes the majority of the overall code base [2], [3]. Besides benefits in productivity, this practice can also introduce security risks to a project – according to *Sonatype*, one in eight open source downloads has been vulnerable in 2023 [4].

Consequently, the reuse of third-party libraries has become an intensively researched field. This includes detecting vulnerabilities [5], software evolution [6] and investigating API (in-)compatibility [7], [8]. One promising approach for doing so is *large-scale static analysis* - an implementation pattern where a large part of a repository is analyzed statically, i.e. without executing the code under analysis [7], [9]–[11].

Researchers often implement such analyses from scratch - reuse of *analysis implementations* or *analysis results* rarely occurs [12], [13]. This approach results in some disadvantages: Implementation effort is expended multiple times, prototypes are less mature, and bugs are more likely. Also, any change in a repository's API could break analysis implementations, thus increasing maintenance overhead.

With this work, we propose *MARIN*, the **MA**ven **R**esearch **IN**terface. MARIN is a JVM-based library that provides an abstract framework for implementing analyses targeting Maven Central. It implements many Maven-specific functionalities in accordance with their specification, removing the need for re-implementation and thus reducing the likelihood of introducing bugs. MARIN introduces a clear separation between the *actual analysis implementation* – which is domain-specific – and the *analysis infrastructure*, which is provided by MARIN. This rectifies the issues mentioned above: Changes to the Maven Central API will only affect MARIN and not propagate further, while common functionalities for working with the repository are implemented exactly once by MARIN, and can be relied on by client analyses.

Given an actual analysis implementation that processes a single Maven artifact, MARIN handles tasks like accessing the index, applying the analysis to a (configurable) set of artifacts, aggregating data from `pom` or `jar` files (as required), as well as incremental restarts.

MARIN implements many Maven-specific functionalities, including the resolution of direct and transitive dependencies, analysis of dependency conflicts and support for dependency version ranges. We represent both *raw* information from an artifact's metadata file (e.g. incomplete version specifications, property references, imports), as well as *resolved* information – both may be useful for implementing concrete analyses.

In short, this work contributes:

- A brief survey of large-scale static analyses implementations motivating the need for a common interface.
- An implementation of MARIN, a JVM-based library that facilitates large-scale program analysis on Maven Central. MARIN is available on *GitHub* [14].
- A performance evaluation for analyses built with MARIN, which is available on Zenodo [15].

Our evaluation finds that configuring MARIN to only retrieve information that is truly required for an analysis can save a lot of execution time – which can be further reduced by up to 88% using our built-in multi-threading support.

II. Motivation and State-of-the-Art

In order to understand requirements for building large-scale program analyses, we survey existing implementations focusing on the features they require and the tools they use.

A. Methodology

To find relevant publications, we focused on the *Mining Software Repositories* (MSR) conference series, specifically its *Data and Tools Showcase Track* and technical papers. This was done since the conference deals with mining information, often on a large scale, from repositories like Maven Central, while the specific tracks also call for *implementations* of such analyses. We extend this initial set by the use of snowballing. The search terms *"Java"* and *"Maven"* were used to filter for relevant publications. Our final data set consists of ten publications on analysis implementations [7], [9], [16]–[23].

For our purposes, we are interested in two different aspects of each publication: the implementation's feature requirements (`A1`) and tools used (`A2`). To obtain meaningful results, we employ an approach based on *open card sorting* [24]. In that, for every publication we extracted notes on both `A1` and `A2`. We then grouped notes into common categories per aspect.

B. Results

For aspect `A1`, we obtained a set of seven categories. Table I illustrates their names, the papers associated with the category, and a *Prevalence Score* $\mathbb{S}_P$ indicating how many of the total papers fall into that category. We can see that the most popular categories are *"Enumerate Repository Contents"* (`C1`) and *"Compare Semantic Versions"* (`C2`) - 90% and 80% of all papers, respectively, fall into those categories. On the other hand, only three papers require enumerating classes (`C7`). In general, we can observe a relatively high agreement among publications, with at least 70% belonging to five categories.

For `A2`, we identify tools and libraries used by the publications in our data set. The results are shown in Table II, with the last column indicating when the respective tool reached its end-of-life. Here, we observe much less agreement compared to `A1` - no tool is used by more than two publications. Five tools reached end-of-live, only *Apache Commons*, *Maven-Model* and the *Maven Artifact API* are still being maintained.

C. Discussion

Our findings are a clear indication on what is required of a research interface for large-scale analyses: Features `C1` through `C5` are used by at least 70% of all publications surveyed. Working with binary files (`C6` and `C7`) is still relevant for some publications, but to a lesser degree.

While many tools focus on parsing metadata and resolving dependencies (`T1`, `T4`, `T8`) or domain-specific tasks (`T3`, `T6`, `T7`), little focus is put on addressing the *large-scale* aspect, especially regarding `C1`. While `T5` was originally designed to fill this gap, its last commit was over ten years ago.

In summary, we observe that while some important features for large-scale static analysis are not supported by tools at all, others can be implemented using multiple different libraries or frameworks.

III. Design

Based on the observations made from our literature survey, we derive a design for *MARIN* that covers as many common large-scale analysis requirements as possible.

A. Requirements

We obtain a set of feature requirements based on our findings in Table I. *MARIN* **must**:

- *enable users to enumerate the identifiers,* `pom.xml` *and* `JAR` *files of Maven Central* ***in order to*** *parse metadata or binaries.*
- *always parse some basic metadata when enumerating artifacts* ***in order to*** *provide access to dependencies, time of release, and version information.*
- *extract some structural information from* `JAR` *files* ***in order to*** *simplify the enumeration of classes and other binary analyses.*
- *resolve transitive and effective dependencies according to the Maven specification* ***in order to*** *enable whole-program analyses.*

We also define some non-functional requirements based on our own experience with large-scale analyses in general. Such requirements are often overlooked, but greatly contribute to the reproducibility and adoption of analysis implementations. *MARIN* **should**:

- *impose minimal performance overhead* ***in order to*** *make large-scale analyses feasible.*
- *provide facilities to pause, restart, and re-run analyses, as well as to recover from unexpected shutdowns,* ***in order to*** *ease deployment and reproducibility.*

B. Data Model

Based on these requirements, we derive a suitable data model to represent the core domain of large-scale static analyses on Maven Central. Figure 1 presents the most relevant parts of this model as an UML class diagram. It is centered around a class named `Artifact`, which represents what is colloquially referred to as a *Version* or *Release* of a library.

As indicated in the diagram, `Artifacts` may be enriched with up to three different types of `ArtifactInformation` - `PomInformation`, `IndexInformation`, and `JarInformation`. We decide to introduce this separation since each kind of information is obtained from a different source, which involves downloading and / or parsing files. In order to only introduce the minimal performance overhead necessary for any given concrete use-case (as per our non-functional requirements), the user may specify which information object kinds an Artifact shall be enriched with.

Parsing `pom.xml` files, we obtain relevant information including the artifact's *description*, *licensing information*, *defined properties* [25] as well as *direct* – and *managed* [26] – *dependencies*. It must be noted that the latter may be incomplete, as Maven allows referencing properties and parts of dependency specifications from other artifacts via the `<parent>`-mechanism or `import`-scoped dependencies [26] – therefore we encapsulate this information in a

TABLE I
RESULTS OF OPEN CARD SORTING PROCESS FOR `A1` SORTED BY PREVALENCE

ID	Feature Category	Papers	$\mathbb{S}_P$
`C1`	Enumerate Repository Contents	[7], [9], [16]–[20], [22], [23]	90%
`C2`	Compare Semantic Versions	[7], [9], [17]–[21], [23]	80%
`C3`	Parse Metadata or Configurations	[9], [16]–[20], [23]	70%
`C4`	Compute Dependencies	[9], [16]–[20], [23]	70%
`C5`	Get Time of Release	[9], [16], [18]–[21], [23]	70%
`C6`	Parse Binaries or Source File(s)	[7], [16], [17], [22]	40%
`C7`	Enumerate Classes in Binaries	[7], [16], [22]	30%

TABLE II
RESULTS OF OPEN CARD SORTING PROCESS FOR `A2`

ID	Tool	Used By	End-of-Life
`T1`	*Eclipse Aether*	[9], [23]	Feb 2016
`T2`	*Apache Commons*	[19], [20]	Still Active
`T3`	*Dependency Graph Miner*	[21]	Dec 2019
`T4`	*Maven-Model*	[19], [20]	Still Active
`T5`	*PomWalker*	[19], [20]	Jan 2014
`T6`	*FindBugs*	[17]	Mar 2015
`T7`	*Clirr*	[7], [16]	Feb 2006
`T8`	*Maven Artifact API*	[7]	Still Active
`T9`	Closed Source / Proprietary	[16], [18]	/

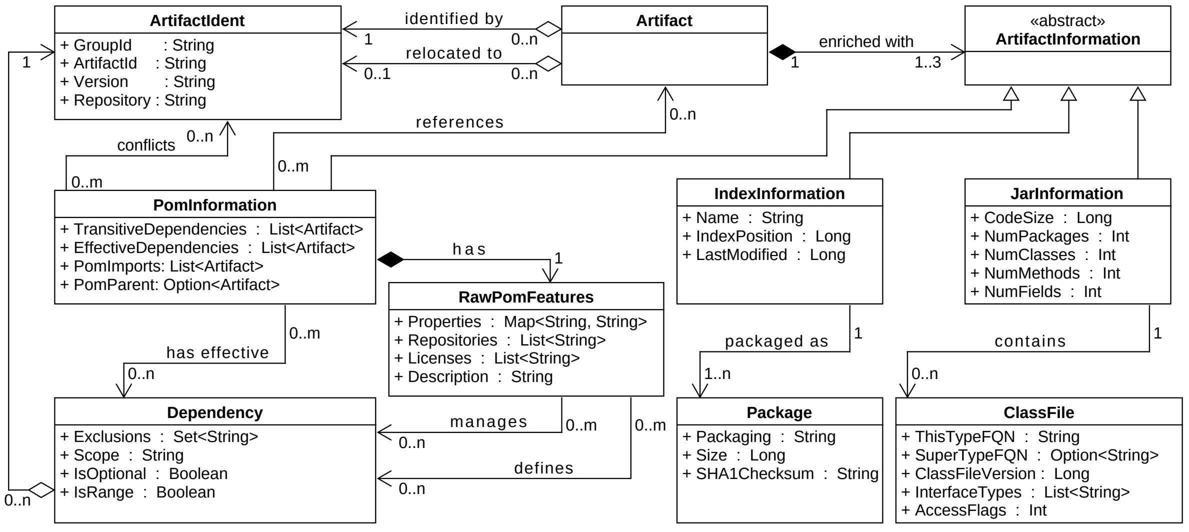

Fig. 1. The Core Data Model of MARIN

class called `RawPomFeatures`. `PomInformation` builds atop of this and provides access to the *effective* data as well, resolving the aforementioned references to other artifacts and building a complete transitive dependency tree. The same features are available for *local* projects that are not hosted on any repository online.

Instances of class `JarInformation` hold basic information on an artifact's implementation, including the total code size and some statistics on the number of programming constructs. Based on our requirements, we further chose to represent a list of classes, each holding enough information to construct the artifact's type hierarchy if necessary.

Finally, `IndexInformation` represents data stored in the *Maven Central Lucene Index* [27] which MARIN uses to enumerate all artifacts within the ecosystem. This class holds information on different *Packages* available, as in Maven a single artifact may be published in multiple different formats.

C. Architecture

Figure 2 shows a UML class diagram of MARIN's central components for user interaction.

The `IndexWalker` can be used to iterate the contents of a Maven repository – this corresponds to our first functional requirement. It provides access to either `ArtifactIdent` or `Artifact` objects, where the latter is enriched with `IndexInformation`.

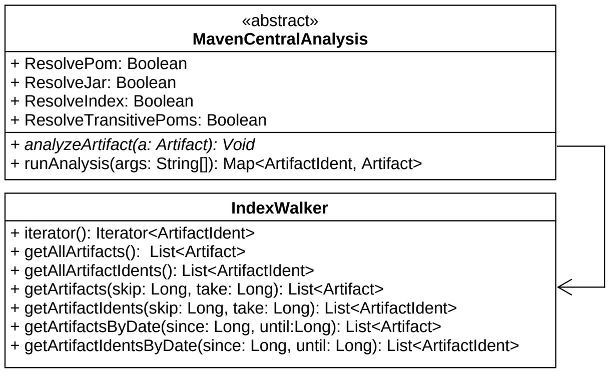
«abstract»
MavenCentralAnalysis
+ ResolvePom: Boolean
+ ResolveJar: Boolean
+ ResolveIndex: Boolean
+ ResolveTransitivePoms: Boolean
+ analyzeArtifact(a: Artifact): Void
+ runAnalysis(args: String[]): Map<ArtifactIdent, Artifact>
IndexWalker
+ iterator(): Iterator<ArtifactIdent>
+ getAllArtifacts(): List<Artifact>
+ getAllArtifactIdents(): List<ArtifactIdent>
+ getArtifacts(skip: Long, take: Long): List<Artifact>
+ getArtifactIdents(skip: Long, take: Long): List<ArtifactIdent>
+ getArtifactsByDate(since: Long, until:Long): List<Artifact>
+ getArtifactIdentsByDate(since: Long, until: Long): List<ArtifactIdent>

Fig. 2. Main Components of MARIN

The center-piece of MARIN is the *abstract* class `MavenCentralAnalysis`. To extend it, users only have to provide an implementation of the method `analyzeArtifact` that defines how a single artifact shall be analyzed, and select which kind(s) of `ArtifactInformation` they require. Calling

TABLE III
AVERAGE EXECUTION TIMES PER CONFIGURATION

Configuration	Description	Average Duration [**hh:mm:ss**]
$\mathbb{C}_1$	Index only	`00:00:09`
$\mathbb{C}_2$	Raw POM only	`00:28:55`
$\mathbb{C}_3$	Transitive POM only	`02:11:35`
$\mathbb{C}_4$	JAR only	`02:15:37`
$\mathbb{C}_5$	All information	`04:42:09`

`runAnalysis` from within a `main` Method will then start a full large-scale analysis of Maven Central, calling `analyzeArtifact` for each individual artifact.

The class by default supports a number of execution modes selected via command-line parameters, including *index pagination*, custom filters on artifacts by publication date, incremental restarts from the last known index position, and multi-threading. Further information about relevant components and their interfaces can be found on the MARIN GitHub page [14].

IV. EVALUATION

We evaluate our implementation of MARIN by running analyses with different resolution configurations and timing them. We select the following configurations, as they represent different use-cases for real-world analyses:

Index Information ($\mathbb{C}_1$), Raw POM Information ($\mathbb{C}_2$), Effective POM Information ($\mathbb{C}_3$), JAR Information ($\mathbb{C}_4$), All Information ($\mathbb{C}_5$)

We analyze the first $250,000$ index entries, yielding $109,794$ unique identifiers for every configuration.

All configurations are executed on a server with an *Intel Xeon E5-2650 quad-core CPU* and 32GB of RAM. We run each configuration three times and report averages here.

A. Results

Table III reports the average execution times per configuration. We can see that the most complex configuration ($\mathbb{C}_5$) also takes the longest time to execute. The most expensive part of the resolution seem to be the resolution of JARs($\mathbb{C}_4$) and the transitive aspect of POM resolution ($\mathbb{C}_3$), each taking over two hours. Our findings illustrate that it is important to select only the information required for a concrete analysis to avoid unnecessary performance overhead.

B. Multi-threading

To further improve the performance for client analyses, MARIN supports multi-threaded execution. We select configurations $\mathbb{C}_3$ and $\mathbb{C}_4$ and compare their multi-threaded execution times with the ones reported for single-threaded execution. We execute both configurations with four and eight threads, using the `--multi <n>` switch provided by MARIN.

Table IV shows the results. For $\mathbb{C}_3$, using multi-threading reduces the runtime significantly, namely by 76% (4 threads) and 88% (8 threads). Reduction rates for $\mathbb{C}_4$ are lower but still significant, reaching up to 39% reduction for eight threads.

TABLE IV
AVERAGE EXECUTION TIMES FOR MULTI-THREADED ANALYSES

Configuration	4 Threads [**hh:mm:ss**]	8 Threads [**hh:mm:ss**]
$\mathbb{C}_3$	`00:31:12`	`00:15:56`
$\mathbb{C}_4$	`01:38:27`	`01:23:14`

Thus, our multi-threaded implementation has a significant impact on reducing runtime.

C. Limitations

We observe that some identifiers listed in the index are no longer hosted on Maven Central. This is due to different processes being in place in the early days of the repository, and affects less than 1% of the artifacts in our evaluation.

Also, we observe a number of errors when parsing `POM` and `JAR` files. These are either due to the files being malformed or caused by limitations of the underlying libraries – *Maven Model* for XML and *OPAL* [28], [29] for `JAR` files.

V. RELATED WORK

In 2023, Litzenberger et al. proposed *DGMF*, a dependency graph mining framework [13]. While it does provide a common model for such graphs, utilities to validate and store them, and extension points for custom implementations, it is still specific to the domain of *dependency graph generation* - it cannot be used for *any* large-scale analysis. The same is true for *Goblin*, a framework for enhancing the Maven Central dependency graph with custom values and timestamp-dependent analysis capabilities [23].

Git2Net is a Python package that provides functionality to enumerate, clone and analyze repositories from GitHub [30]. It focuses less on the actual *contents* of a repository (i.e. metadata files and dependencies), and more on authorship and file edits.

VI. CONCLUSION

In this paper we presented MARIN, a framework for conducting analyses on Maven Central software artifacts. MARIN untangles the often intertwined implementations of domain-specific analysis tasks and repository-specific infrastructural tasks by providing clean, specification-adhering implementations for the latter. Our evaluation shows that being able to configure the amount of information extracted per artifact helps reducing unnecessary performance overhead for concrete analyses, while using MARIN's built-in support for multi-threading can further optimize runtime.

DATA AVAILABILITY

Our artifact is hosted on Zenodo. It contains MARIN and all information necessary to reproduce our evaluation [15].

ACKNOWLEDGMENTS

Jared Chiaramonte worked on this research during his 2024 internship at TU Dortmund funded by German Academic Exchange Service (DAAD) in the RISE Germany funding program. The authors would like to thank the DAAD for enabling the collaboration which made this project possible.

References

[1] Sonatype Inc, “Maven central repository,” https://central.sonatype.com, 2023.

[2] 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

[3] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck, “On the extent and nature of software reuse in open source java projects,” in *Top Productivity through Software Reuse*, K. Schmid, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 207–222.

[4] Sonatype Inc, “9th annual state of the software supply chain,” https://www.sonatype.com/hubfs/SSC/2023%20Sonatype-%209th%20Annual%20State%20of%20the%20Software%20Supply%20Chain-%20Update.pdf, 2023.

[5] S. E. Ponta, H. Plate, and A. Sabetta, “Detection, assessment and mitigation of vulnerabilities in open source dependencies,” *Empirical Software Engineering*, vol. 25, no. 5, pp. 3175–3215, Sep 2020. [Online]. Available: https://doi.org/10.1007/s10664-020-09830-x

[6] E. Constantinou and I. Stamelos, “Identifying evolution patterns: a metrics-based approach for external library reuse,” *Software: Practice and Experience*, vol. 47, no. 7, pp. 1027–1039, 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2484

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

[8] A. Dann, B. Hermann, and E. Bodden, “UPCY: safely updating outdated dependencies,” in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 233–244. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00031

[9] 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.

[10] J. Düsing and B. Hermann, “Analyzing the direct and transitive impact of vulnerabilities onto different artifact repositories,” *Digital Threats*, vol. 3, no. 4, feb 2022. [Online]. Available: https://doi.org/10.1145/3472811

[11] J. Hejderup, M. Beller, K. Triantafyllou, and G. Gousios, “Präzi: from package-based to call-based dependency networks,” *Empirical Software Engineering*, vol. 27, no. 5, May 2022. [Online]. Available: http://dx.doi.org/10.1007/s10664-021-10071-9

[12] J. Düsing and B. Hermann, “Persisting and reusing results of static program analyses on a large scale,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 888–900.

[13] T. Litzenberger, J. Düsing, and B. Hermann, “Dgmf: Fast generation of comparable, updatable dependency graphs for software repositories,” in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023, pp. 115–119.

[14] J. Düsing, J. Chiaramonte, and B. Hermann, “Maven Research Interface (MARIN),” 2024. [Online]. Available: https://github.com/sse-labs/marin

[15] ——, “MARIN: A Research-Centric Interface for Querying Software Artifacts on Maven Repositories - Companion Artifact,” Nov. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.14235313

[16] S. Raemaekers, A. van Deursen, and J. Visser, “The maven repository dataset of metrics, changes, and dependencies,” in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 221–224.

[17] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, and D. Spinellis, “The bug catalog of the maven ecosystem,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 372–375. [Online]. Available: https://doi.org/10.1145/2597073.2597123

[18] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, “Measuring dependency freshness in software systems,” in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE ’15. IEEE Press, 2015, p. 109–118.

[19] R. G. Kula, D. M. German, T. Ishio, and K. Inoue, “Trusting a library: A study of the latency to adopt the latest maven release,” in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 520–524.

[20] R. G. Kula, D. M. German, T. Ishio, A. Ouni, and K. Inoue, “An exploratory study on library aging by monitoring client usage in a software ecosystem,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 407–411.

[21] C. Soto-Valero, A. Benelallam, N. Harrand, O. Barais, and B. Baudry, “The emergence of software diversity in maven central,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 333–343.

[22] B. Theeten, F. Vandeputte, and T. Van Cutsem, “Import2vec: Learning embeddings for software libraries,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 18–28.

[23] D. Jaime, J. E. Haddad, and P. Poizat, “Goblin: A framework for enriching and querying the maven central dependency graph,” in *Proceedings of the 21st International Conference on Mining Software Repositories*, ser. MSR ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 37–41. [Online]. Available: https://doi.org/10.1145/3643991.3644879

[24] J. R. Wood and L. E. Wood, “Card sorting: Current practices and beyond,” *J. Usability Studies*, vol. 4, no. 1, p. 1–6, nov 2008.

[25] The Apache Software Foundation, “Properties - pom reference - maven,” https://maven.apache.org/pom.html#Properties, 2024.

[26] Apache Software Foundation, “Introduction to the dependency mechanism - maven,” https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html, 2024.

[27] The Apache Software Foundation, “Maven indexer core - introduction,” https://maven.apache.org/maven-indexer/indexer-core/, 2024.

[28] M. Eichberg and B. Hermann, “A software product line for static analyses: the OPAL framework,” in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State Of the Art in Java Program analysis, SOAP 2014, Edinburgh, UK, Co-located with PLDI 2014, June 12, 2014*, S. Arzt and R. A. Santelices, Eds. ACM, 2014, pp. 2:1–2:6. [Online]. Available: https://doi.org/10.1145/2614628.2614630

[29] D. Helm, F. Kübler, M. Reif, M. Eichberg, and M. Mezini, “Modular collaborative program analysis in opal,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 184–196. [Online]. Available: https://doi.org/10.1145/3368089.3409765

[30] C. Gote, I. Scholtes, and F. Schweitzer, “git2net: Mining time-stamped co-editing networks from large git repositories,” in *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 2019, pp. 433–444.