EasyChair Preprint

№ 356

Towards an Open Source Stack to Create a Unified Data Source for Software Analysis and Visualization

Richard Müller, Dirk Mahler, Michael Hunger, Jens Nerche and Markus Harrer

July 17, 2018

Towards an Open Source Stack to Create a Unified Data Source for Software Analysis and Visualization

Richard Müller*, Dirk Mahler†, Michael Hunger‡, Jens Nerche§ and Markus Harrer¶
*Leipzig University, Germany
Email: rmueller@wifa.uni-leipzig.de
†buschmais GbR, Dresden, Germany
Email: dirk.mahler@buschmais.com
‡Developer Relations, Neo4j Inc., Malmö, Sweden
Email: michael.hunger@neo4j.com
§Application Development, Kontext E GmbH, Dresden, Germany
Email: j.nerche@kontext-e.de
¶Software Development Analyst, Freelancer, Roth, Germany
Email: contact@markusharrer.de

***Abstract*—The beginning of every software analysis and visualization process is data acquisition. However, there are various sources of data about a software system. The methods used to extract the relevant data are as diverse as the sources are. Furthermore, integration and storage of heterogeneous data from different software artifacts to form a unified data source are very challenging. In this paper, we introduce an extensible open source stack to take the first step to solve these challenges. We show its feasibility by analyzing and visualizing JUnit and provide answers regarding the schema, selection, and implementation of software artifacts' data.**

***Index Terms*—software analysis, software visualization, schema, graph database, query, open source**

I. Introduction

Software analysis and visualization are a vital means for making informed decisions in software development and maintenance projects. The quality of these decisions strongly depends on the quality of the underlying data source. The data should be accurate, complete, consistent, credible, and current [1]. In case of software, this means that its structural, behavioral, as well as evolutionary data [2, p. 3f] should be considered and accessible from a unified data source.

The software visualization pipeline describes the steps to transform data from software artifacts into visual representations. These steps are *data acquisition*, *analysis*, and *visualization* [2, p. 12]. There are various kinds of software artifacts belonging to a software system, such as source code, test results, code analysis results, or version control logs. During *data acquisition*, the relevant data from these artifacts is extracted. The data naturally maps to a multivariate, compound, attributed, and time-dependent graph [3]. This graph consists of entities, their relations, and attributes. In the *analysis* step, the data is aggregated, enriched, and the relevant parts are filtered. The resulting entities and relations are mapped to *marks* and their attributes are mapped to *channels* resulting in a specific *visualization* [4, Ch. 5]. *Views* define the parts of the visualization shown to the user on a display to support a specific task [5].

Creating, storing, and querying the data captured by such graphs is very challenging. Diehl et al. summarize the most important questions in this respect [3].

1) **Schema:** How to model a given aspect of a software system in terms of entities, relations, and attributes?
2) **Selection:** How to select data relevant for a given task from an entire graph?
3) **Implementation:** How to store the graph in a way that is efficient for quickly reading and writing large amounts of data?

Our contribution in this paper is the introduction of an open source stack providing answers to these questions. We present *jQAssistant*, an extensible tool that scans different kinds of software artifacts and stores the extracted data as a graph. Further, we introduce *Neo4j* as a suitable storage, analysis, and filter tool for heterogeneous software data. Finally, we present a prototype as a proof of concept visualizing integrated structural, behavioral, and evolutionary data with *D3* and *React*.

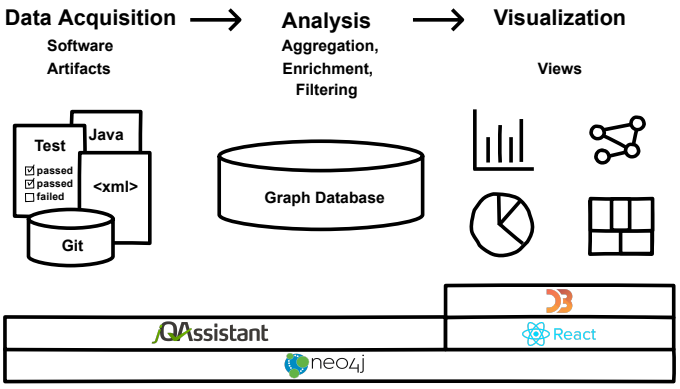


Fig. 1: Open source stack to extract, analyze, and visualize heterogeneous information of software artifacts.

II. Open Source Stack

The proposed open source stack creates a unified data source for software analysis and visualization. jQAssistant scanners extract heterogeneous data from software artifacts and store it in a Neo4j graph database. jQAssistant rules and Neo4j's query language *Cypher* are used to aggregate, enrich, and filter the important parts. Finally, the filtered data is mapped to D3 components embedded in a React app and visualized in a browser. The components of the open source stack are summarized in Fig. 1.

A. Neo4j

Neo4j[1] is an open source graph database that is built to store, manage, and query large amounts of connected data. It is a native graph database as it implements the data model efficiently down to the storage level.

1) Model: Its data model is a *property graph*, linking *labeled nodes* with *named, directed relationships* both of which can carry arbitrary sets of *properties* as key-value pairs. There is no rigid schema which makes it suitable for linking variably shaped information from different data sources. The abstract graph data model is shown in Fig. 2.

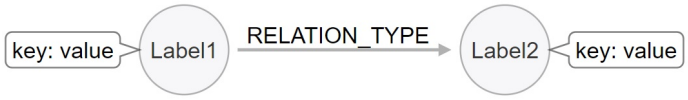


Fig. 2: Abstract graph data model of Neo4j.

2) Cypher: The graph query language Cypher [6] matches given patterns in the graph using a visual, ASCII art-based syntax, e.g., `(node1:Label1)-[:RELATION_TYPE]->(node2:Label2)`. Cypher supports all regular query language operations as well as comprehensions, data flow concepts, and user-defined functions and procedures. The language evolution is driven by the openCypher[2] organization of independent researchers and vendors.

3) Driver: Neo4j has flexible deployment options, from embedding it into an application, running it as a server or on cloud infrastructure. The database supports both transactional operations as well as large-scale graph analytics. Drivers for most programming languages, e.g., Java, Python, R, .Net, and JavaScript allow to execute Cypher statements and return tabular or graph results. The driver is the interface between the data handling component and the visualization component of the stack.

B. jQAssistant

jQAssistant[3] is a tool based on Neo4j for scanning software artifacts' data and analyzing the extracted data by applying rules. It has been developed for automated verification of a software system's implementation compared to its specification. Therefore, jQAssistant is commonly used as part of build pipelines on CI (continuous integration) servers. It integrates with the build tool Apache Maven[4] but can also be executed independently as command line utility. The core of jQAssistant is a framework which provides interfaces for scanner, rule, and report plugins. The framework is agnostic to any specific programming language or technology that shall be scanned or analyzed.

1) Scanner: The scanner extracts data from software artifacts and stores it as a graph in the Neo4j database. The process is controlled by the framework. Plugins are responsible for interpreting specific types of information like directory structures, `.class` files containing Java bytecode, or URLs. Each plugin creates a sub-graph representing the extracted data. In defined technical contexts, relevant information may be exchanged between plugins to create connected graphs.

2) Rules: In a subsequent step the graphs can be analyzed by applying rules on them: *concepts* for data enrichment and *constraints* for detecting violations.

A concept adds abstractions to the graph which are defined in the design or architecture language of the system. For example, in the library JUnit, a direct child package of the root package `org.junit` may be interpreted as `Component`. Hence, a concept with the id `junit:Component` can be defined that adds a label to each corresponding node using a Cypher query.

```
MATCH (p:Package)-[:CONTAINS]->(c:Package)
WHERE p.fqn = "org.junit"
SET c:Component // set the label "Component"
RETURN c as component
```

It is possible to have dependencies between concepts. For example, the concept `junit:ComponentDependencies` requires the enriched graph of `junit:Component` for aggregating dependencies between contained types to the component level. This can be achieved by creating relationships of type `DEPENDS_ON`. The property `weight` indicates the degree of coupling.

```
MATCH (c1:Component)-[:CONTAINS*]->(t1:Type),
(c2:Component)-[:CONTAINS*]->(t2:Type),
(t1)-[:DEPENDS_ON]->(t2)
WHERE c1 <> c2
WITH c1, c2, count(*) as weight
// creates a relation for component dependencies
MERGE (c1)-[d:DEPENDS_ON]->(c2)
SET d.weight = weight
RETURN c1, c2, weight
```

A constraint is also expressed as a Cypher query. It is violated if it returns a result that is not empty and usually depends on one or more concepts.

With these means, scanned data from different sources can be automatically aggregated, enriched and connected. For example, LOC (lines of code) from method level can be aggregated to classes and packages or to newly added components. Moreover, the components can be enriched with further data such as average change frequency or test coverage.

3) Plugins: The main distribution of jQAssistant provides a set of plugins with scanners and rules that support file formats with a focus on the Java programming language and its

[1]https://neo4j.com
[2]https://www.opencypher.org
[3]https://jqassistant.org
[4]https://maven.apache.org

related technologies. The scanners provided by the Java plugin create graphs representing core elements of the language, i.e. nodes labeled with `Package`, `Type`, `Field`, `Method`, and `Annotation` that are connected by corresponding relationships, e.g., `(:Package)-[:CONTAINS]->(:Type)` or `(:Type)-[:DECLARES]->(:Method)-[:RETURNS]->(:Type)`. The label `Type` represents a Java type that can be qualified further by another label like `:Type:Class` or `:Type:Interface`.

Furthermore, there are plugins[5] contributed by the jQAssistant community. Next, the Git, Jacoco, and PMD plugins are briefly described.

Git[6] is a distributed version control system. Its data is organized as a graph which naturally fits into a graph database. The jQAssistant Git plugin imports the meta-data but not the file content. Imported nodes are `Repository`, `Author`, `Commit`, `Change`, `File`, `Branch`, and `Tag`. The connection to the Java bytecode graph is done via `File` nodes. An example of a concept are the merge commits which are marked nodes with the label `Merge`.

JaCoCo[7] is a Java code coverage library. It is mainly used for checking code coverage of unit tests. The jQAssistant JaCoCo plugin imports the XML report file. Imported nodes are `Report`, `Package`, `Class`, `Method`, and `Counter`. The connection to the Java bytecode graph is done via `Package`, `Class`, and `Method` nodes. An example of a constraint is to define test coverage rules such as that a method with a given complexity requires a specified test coverage.

PMD[8] is an extensible, cross-language, and static source code analyzer with a rich set of rules for Java. The jQAssistant PMD plugin imports the XML report file. Imported nodes are `Report`, `File`, and `Violation`. The connection to the Java bytecode graph is done via `File` nodes. An example of a concept is to enrich existing Java class nodes with the number of PMD violations.

C. D3 and React

D3 (Data-Driven Documents) [7] is a JavaScript library to manipulate data and to create interactive, web-based visualizations. It uses established web standards such as HTML, SVG, and CSS and provides a rich set of visualization techniques.

React[9] is a JavaScript library for building web-based user interfaces. It is a perfect companion of D3 as it turns D3 visualizations in reusable visualization components.

III. Proof of Concept

To provide a proof of concept, we present a *Software Analysis and Visualization Dashboard*[10] as one visualization frontend for the open source stack. The dashboard supports project leaders in decision-making and provides interactive views concerning architecture and dependencies as well as resource, risk, and quality management of a software system. It is implemented as React application and uses D3 visualization components[11]. The data for each view is dynamically queried from a Neo4j database.

As an example use case for analysis and visualization, we have chosen the open source project JUnit[12]. At first, the bytecode, Git log, test coverage results, and static code analysis results are scanned with jQAssistant and stored in a Neo4j database. During analysis, predefined rules aggregate, enrich, and connect this data. A screencast demonstrating the data acquisition, analysis, and visualization of JUnit is available in the repository of the dashboard. Next, we present some selected views based on this unified data source.

A. Dependency Analysis

Dependency analysis is important to assess the coupling and cohesion of a software system. To visualize the dependencies, structural data is necessary. The following Cypher query is based on the concept `junit:ComponentDependencies` and returns the fully qualified name for each component and its dependencies including the weight.

```
MATCH (source:Component)-[d:DEPENDS_ON]->
(target:Component)
RETURN source.fqn as component, target.fqn as
dependency, d.weight as weight
```

The returned data is used to create a dependencies analysis view with a chord diagram. The fully qualified names of the components are arranged radially around a circle and the dependencies are drawn as arcs. The corresponding view for dependency analysis is shown in Fig. 3 (a).

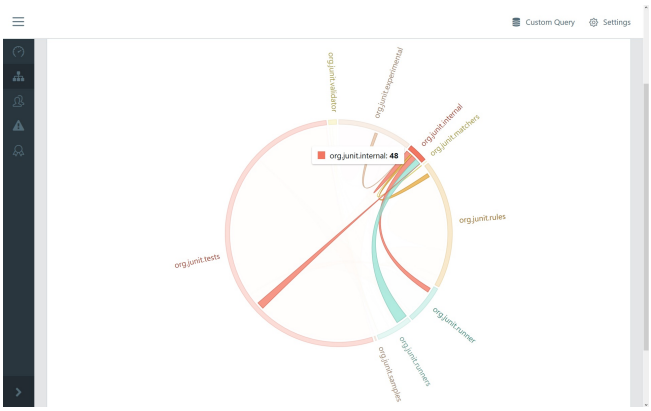
B. Hotspot Analysis

Hotspot analysis supports assessing the risk of a software system [8, p. 19]. Hotspots are complex parts of the source code that change often. They are usually candidates for improvements or refactoring. To visualize these hotspots, structural data and evolutionary data are necessary. The following Cypher query returns the fully qualified name for each type, its LOC, and its number of commits.

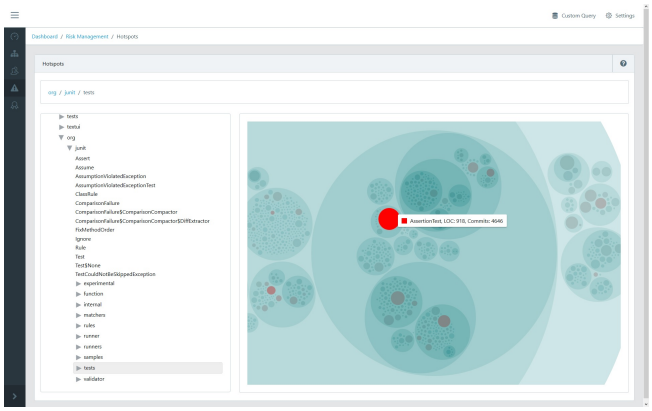
```
MATCH (c:Commit)-[:CONTAINS_CHANGE]->
()-[:MODIFIES]->(f:File)
WHERE NOT c:Merge
WITH f, count(c) as commits
MATCH (t:Type)-[:HAS_SOURCE]->(f),
(t)-[:DECLARES]->(m:Method)
RETURN t.fqn as type,
sum(m.effectiveLineCount) as loc,
sum(commits) as commits
```

The returned data is used to create a hotspot analysis view based on circle packing [8, p. 20]. The fully qualified names of the types are mapped to nested circles with LOC as the size and the number of commits as the color of a circle. The corresponding view for hotspot analysis is shown in Fig. 3 (b).

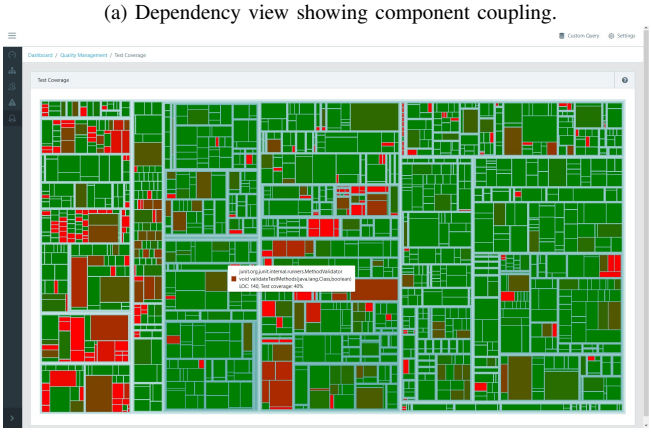
[5]https://github.com/kontext-e/jqassistant-plugins
[6]https://git-scm.com
[7]https://www.jacoco.org
[8]https://pmd.github.io
[9]https://reactjs.org/
[10]https://github.com/softvis-research/jqa-dashboard
[11]https://github.com/plouc/nivo
[12]https://github.com/junit-team/junit4



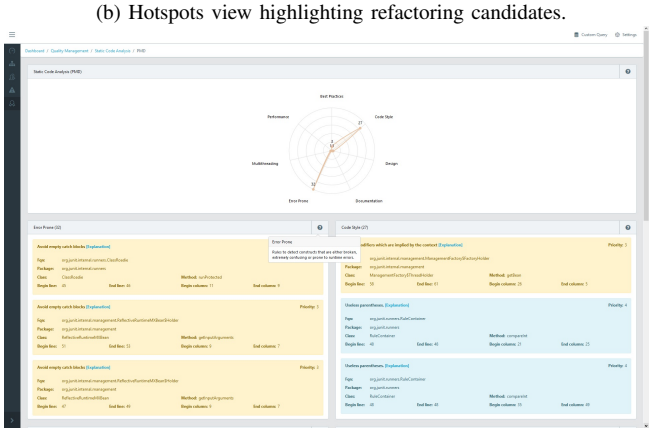
(a) Dependency view showing component coupling.



(b) Hotspots view highlighting refactoring candidates.



(c) Test coverage view highlighting untested code.



(d) Lists showing static code analysis results from PMD.

Fig. 3: Different views of JUnit based on a unified data source.

C. Test Coverage Analysis

Test coverage analysis supports assessing the quality of a software system. To visualize the test coverage, structural and behavioral data are necessary. The following Cypher query returns the fully qualified names for all classes, the signature of their methods with the corresponding covered instructions and LOC.

```
MATCH (c:Jacoco:Class)-[:HAS_METHOD]->
(m:Method:Jacoco)-[:HAS_COUNTER]->(cnt:Counter)
WHERE cnt.type="INSTRUCTION"
RETURN c.fqn as fqn, m.signature as signature, (cnt.
covered * 100) / (cnt.covered + cnt.missed) as
coverage, cnt.covered + cnt.missed as loc
```

The returned data is used to create a test coverage analysis view with a treemap. The fully qualified names of the types and method signatures are mapped to nested rectangles where the LOC define the size and the coverage defines the color of a rectangle. The corresponding view for test coverage analysis is shown in Fig. 3 (c).

D. Static Code Analysis

Static code analysis is another means to assess the quality of a software system. Here, the source code is checked against predefined rules. PMD provides checks regarding best practices, code style, design, documentation, error-proneness, multithreading, and performance. The following Cypher query returns the fully qualified name of a file and further data describing the violation.

```
MATCH (:Report)-[:HAS_FILE]->(file:File:Pmd)
-[:HAS_VIOLATION]->(violation:Violation)
RETURN file.fqn, violation
```

The returned data is used to create a radar chart showing the number of violations in the categories. Furthermore, all violations are summarized by category and listed in separate boxes. Each violation is colored according to its priority. The static code analysis view is shown in Fig. 3 (d).

IV. Related Work

Moose [9] and *Rascal* [10] provide languages for parsing, modeling, and querying software artifacts' data. Moose uses a family of meta-models, namely *Famix* [11] for structural, *Dynamix* [12] for behavioral, and *Hismo* [13] for evolutionary software artifacts' data. These formats are great to store the corresponding information. But the extraction and the integration of the data is still challenging. Rascal is a domain-specific language for software analysis and manipulation. Its main differences to the open source stack are that Rascal focuses on static source code analysis and provides additional source code transformation as well as generation features.

There are some recent approaches using a graph database for storing and querying extracted software artifacts' data. *VerXCombo* [14] stores library dependencies in a Neo4j database and visualizes them with parallel sets using D3. The provided views support system maintainers in decision making to either update or introduce new third-party libraries. The *Swarm Debugging Prototype* [15] stores data from debugging sessions in a Neo4j database and visualizes them as method call graphs and sequence stack diagrams. The provided views aid developers to decrease the required time for deciding where to toggle a break-point and locate bug causes. *VIMETRIK* (Visual Specification of Metrics) [16] is an interactive visual data exploration and data-mining tool to create software metrics. The aggregated data is then visualized using suitable views provided by *KNIME* [17].

We identify three main differences to our proposed open source stack. First, we aim at integrating structural, behavioral, and evolutionary data from different software artifacts in a unified data source. Second, the stack already includes scanners for different software artifacts licensed as open source. Third, the stack provides loosely coupled components that can be tailored for specific needs or project requirements. On the one hand, the scanners are freely selectable. On the other hand, the visualization components are not limited to D3 and React. For example, they can be easily replaced by *Jupyter*[13], *Pandas*[14], and *matplotlib*[15] for 2D visualizations or *A-Frame*[16] for 3D visualizations.

V. Open Questions

We have presented an open and extensible stack for software analysis and visualization. As a proof of concept, we have analyzed and visualized JUnit. The open source stack provides first answers to the questions raised by Diehl et al. [3] with regard to schema, selection, and implementation of software artifacts' data. However, this is just a first step as there are still open questions and challenges.

Which further languages and software artifacts should be supported? jQAssistant mainly supports Java-based software artifacts. Its open and extensible architecture provides best opportunities to develop plugins to support further languages and other types of software artifacts which is especially important in today's polyglot software projects.

What is a suitable schema to store structural entities at different points in time and simultaneously keep the data source consistent? At the moment, the data source contains the complete history log but only one snapshot of the Java bytecode. The major objective is to have code entities, their relations, and attributes at different points in time. Hence, we are working on a Java source code scanner[17] that is able to scan different versions of source code.

How to leverage context-specific views? With the graph-based approach, not only different data sources can be connected, but also different abstraction levels and perspectives on the whole software development lifecycle can be created. Refactorings could be motivated based on change frequency, actual usage data of production systems, and upcoming user stories to find valuable spots for quality improvements. First ideas exist[18], but which possible views support development teams, product owners, or even managers the most?

How to find hidden structures? At code level, most of the actual software structure is hidden. While packages and modules can be used to make it explicit, they do not always capture the right granularity or grouping. By analyzing dependencies with graph algorithms and deep learning techniques, it is possible to detect implicit clusters. These clusters might be derived from a unified data source integrating source code, stack traces, and recorded IDE interactions of developers.

[13]https://jupyter.org

[14]https://pandas.pydata.org

[15]https:/matplotlib.org

[16]https://aframe.io

[17]https://github.com/softvis-research/jqa-javasrc-plugin

[18]https://feststelltaste.de/swot-analysis-for-spotting-worthless-code

References

[1] ISO/IEC 25012:2008, "Software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Data quality model," 2008.

[2] S. Diehl, *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[3] S. Diehl and A. C. Telea, *Multivariate Graphs in Software Engineering*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, vol. 8380, ch. 2, pp. 13–36.

[4] T. Munzner, *Visualization analysis & design*. CRC Press, 2014.

[5] E. H. Chi, "A taxonomy of visualization techniques using the data state reference model," in *IEEE Symp. Inf. Vis.*, 2000, pp. 69–75.

[6] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An Evolving Query Language for Property Graphs," in *ACM SIGMOD Int. Conf. Manag. Data*, 2018, p. 13.

[7] M. Bostock, V. Ogievetsky, and J. Heer, "D3 Data-Driven Documents," *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 12, pp. 2301–2309, 2011.

[8] A. Tornhill, *Software Design X-Rays - Fix Technical Debt with Behavioral Code Analysis*. The Pragmatic Bookshelf, 2018.

[9] O. Nierstrasz, S. Ducasse, and T. Gîrba, "The story of moose: an agile reengineering environment," in *Proc. 10th Eur. Softw. Eng. Conf. held jointly with 13th SIGSOFT Int. Symp. Found. Softw. Eng.*, vol. 30. Lisbon, Portugal: ACM, 2005, pp. 1–10.

[10] P. Klint, T. van der Storm, and J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *2009 9th IEEE Int. Work. Conf. Source Code Anal. Manip.*, 2009, pp. 168–177.

[11] S. Ducasse, N. Anquetil, U. Bhatti, A. C. Hora, J. Laval, and T. Gîrba, "MSE and FAMIX 3.0: an interexchange format and source code model family," p. 40, 2011.

[12] O. Greevy, "Dynamix - a meta-model to support feature-centric analysis," in *1st Int. Work. FAMIX Moose Reengineering*, 2007.

[13] T. Gîrba, J. M. Favre, and S. Ducasse, "Using meta-model transformation to model software evolution," *Electron. Notes Theor. Comput. Sci.*, vol. 137, no. 3, pp. 57–64, 2005.

[14] Y. Yano, R. G. Kula, T. Ishio, and K. Inoue, "VerXCombo: An Interactive Data Visualization of Popular Library Version Combinations," in *23rd Int. Conf. Progr. Compr.*, 2015, pp. 291–294.

[15] F. Petrillo, G. Lacerda, M. Pimenta, and C. Freitas, "Visualizing interactive and shared debugging sessions," in *3rd IEEE Work. Conf. Softw. Vis.*, 2015, pp. 140–144.

[16] T. Khan, H. Barthel, A. Ebert, and P. Liggesmeyer, "Visual analytics of software structure and metrics," in *3rd IEEE Work. Conf. Softw. Vis.*, 2015, pp. 16–25.

[17] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, K. Thiel, and B. Wiswedel, "KNIME - the Konstanz information miner," *ACM SIGKDD Explor. Newsl.*, vol. 11, no. 1, p. 26, 2009.