An Interactive Approach for Inspecting Software System Measurements

Taimur Khan[1(✉)], Henning Barthel[2], Karsten Amrhein[1], Achim Ebert[1], and Peter Liggesmeyer[1,2]

[1] University of Kaiserslautern, Gottlieb-Daimler-Str., 67663 Kaiserslautern, Germany
{tkhan,ebert,liggesmeyer}@cs.uni-kl.de, amrhein@rhrk.uni-kl.de

[2] Fraunhofer IESE, Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
{Henning.Barthel, Peter.liggesmeyer}@iese.fraunhofer.de

Abstract. In recent times, visual analysis has become increasingly important, especially in the area of software measurement, as most of the data from software measurement is multivariate. In this regard, standard software analysis tools are limited by their lack of ability to process huge collections of multidimensional data sets; current tools are designed to either support only well-known metrics or are too complicated to use for generating custom software metrics. Furthermore, the analyst requires extensive knowledge of the underlying data schemas and the relevant querying language. To address these shortcomings, we propose an interactive visual approach that focuses on visual elements, their configurations, and interconnectivity rather than a data ontology and querying language. In order to test and validate our methodology, we developed a prototype tool called VIMETRIK (Visual Specification of Metrics). Our preliminary evaluation study illustrates the intuitiveness and ease-of-use of our approach to understand software measurement and analysis data.

Keywords: Software comprehension · Software measurement · Interactive visual analysis · Visual query specification · Software visualization

1 Introduction

Studies estimate that up to 80 % of the software costs occur in the maintenance phase, 40 % of which goes into understanding the software system [10]. In this context, various measurements or software metrics are often utilized to obtain objective, reproducible, and quantifiable measurements to assist software analysts in quality assurance testing, software debugging, and software performance optimization. Independent of the application area, such measurements are closely scrutinized to ensure that the system in question performs optimally, is safe and reliable, and is of high quality.

Analyses of such measurements of software systems tend to result in the scrutiny of a large amount of analysis data by means of software metrics. Most mainstream analysis tools, such as the Fraunhofer M-System [6], specify and examine these software metrics using a database approach. In this case, an analyzable representation of the source code is generated, stored in a relation or graph database, and queried to create measurement data (e.g., quality or maintainability metrics) [3, 5].

However, these traditional approaches require the analyst to not only have in-depth knowledge of the underlying data schema but also expert knowledge of the relevant querying mechanisms. For using such an approach, the analyst needs an insight into the data ontology and must formulate queries using either the Structured Query Language (SQL) for relational databases [5] or a graph querying language such as GReQL [3] for graph databases. In contrast, we propose an interactive visual approach that focuses on visual elements, their configurations, and interconnectivity rather than a data ontology and querying language.

In this paper, the above-mentioned shortcomings are addressed through an innovative means for facilitating the specification and visualization of user-defined software system measurements. The key ingredients of our approach are a schema-less data access path to an underlying graph data model, a workflow-based approach for defining metrics, and the ability to visually depict the results of these queries not only in traditional forms (e.g., through tabular views, scatter plots, box plots, histograms, line charts, etc.) but also through modern interactive visualization paradigms (e.g., the city view [7]). The novelty of our approach lies in how we combine the specification and visualization of software measurements through data workflows. Ultimately, our goal is to empower end users with the ability to apply tailored metrics and visualization metaphors to visually explore the characteristics of a software system according to their individual needs and requirements.

In order to validate our ideas, we developed a live-data prototype tool called VIMETRIK (Visual Specification of Metrics). Our preliminary evaluation study indicates the prospects and the feasibility of our approach to analyze software measurement data.

2 Related Work

Our approach aims at providing easy and intuitive means for analyzing a software system at different levels of details in terms of software measurements or metrics, such as computing cyclomatic complexity, analyzing dependencies or call traces, or using statistical analysis to find issues. However, current software analysis tools (e.g., [4, 6, 11]) tend to be complicated as they require in-depth knowledge of the underlying data schemas and the relevant querying language.

Closely related tools are: Sextant [13], a tool for specifying and visualizing software metrics for Java source code and WiggleIndexer,[1] a tool that allows the indexing of Abstract Syntax Trees (AST) using a graph database. Although these tools have some similar features, they do not employ workflows in the visual analysis of software systems. WiggleIndexer has a similar graph database model and querying mechanism;

[1] WiggleIndexer – Indexing of AST using graph databases (https://github.com/raoulDoc/WiggleIndexer).

however, users need to have in-depth knowledge of both the model and the underlying Cypher querying language. On the other hand, although Sextant provides several visualization capabilities, it contains significantly fewer details about the system and the analysis needs to be performed in the Service Modeling Language (SML) instead of with workflows.

3 The Methodology

In this section, we present the details of our methodology, which consists of a graph database model capable of capturing full details of a software system's source code and a workflow-based approach for defining metrics.

3.1 The Graph Database Model

Generally, traditional measurement tools use a relational database to store information about the target software system, which leads to a trade-off between scalability and the number of source code details stored. Furthermore, relational databases are often faced with implementation-related issues, such as schema evolution over time, extensive joins of large tables, etc.

In contrast, we propose using a graph database approach aimed at storing full source code details, which scales to large programs and provides an easy means for restructuring. Our proposed graph model is designed to handle queries about software artifacts and is based on AST. The resulting Directed Acyclic Graph (DAG) contains additional links between the vertices of a syntax tree. Our aim in providing such a graph model is to address the many crosscutting queries that our domain experts may have.

For the sake of brevity, Fig. 1 shows only the top-level entities of our proposed graph model as well as the links between these entities. The key ingredients of this top-level model are: an *analysis* root, which may contain links to projects, primitives, arrays, and a literals root; *projects*, which may be connected to top-level packages or, for convenience, directly to compilation units; *packages*, which may have links to sub-packages; and *compilation units*, which contain a top-level type. By traversing the above-mentioned nodes, we touch upon nodes and edges that form the top-level hierarchy of a software system. Additional edges between these elements provide further insight into the various aspects of the underlying software system. For example: *IMPORTS* provides an understanding of the coupling between compilation units and *HAS_LITERAL* offers a means for tracking the usage of literals.

Similarly, our graph model contains representations of other source code elements such as fields, methods, statements, and expressions. It forms a tree-like structure that not only contains the hierarchy of each source file but, more importantly, links to elements found in other source files, too. These additional edges in our graph model connect related elements in a meaningful manner and are a key ingredient for cross-cutting queries, such as finding artifacts that connect method or variable usage to their declaration. Furthermore, nodes and edges in our graph model have properties that

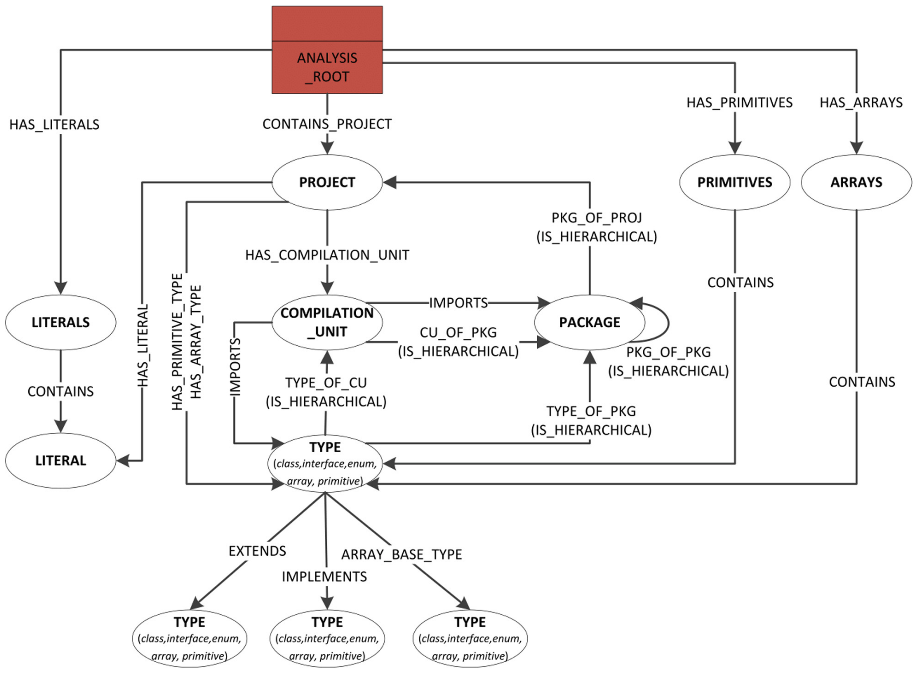

ANALYSIS _ROOT
HAS_LITERALS
CONTAINS_PROJECT
HAS_PRIMITIVES
HAS_ARRAYS
PROJECT
PRIMITIVES
ARRAYS
PKG_OF_PROJ (IS_HIERARCHICAL)
HAS_COMPILATION_UNIT
CONTAINS
LITERALS
HAS_LITERAL
HAS_PRIMITIVE_TYPE
HAS_ARRAY_TYPE
IMPORTS
COMPILATION _UNIT
IMPORTS
CU_OF_PKG (IS_HIERARCHICAL)
PACKAGE
PKG_OF_PKG (IS_HIERARCHICAL)
CONTAINS
TYPE_OF_CU (IS_HIERARCHICAL)
TYPE_OF_PKG (IS_HIERARCHICAL)
LITERAL
TYPE (class,interface,enum, array, primitive)
EXTENDS
IMPLEMENTS
ARRAY_BASE_TYPE
TYPE (class,interface,enum, array, primitive)
TYPE (class,interface,enum, array, primitive)
TYPE (class,interface,enum, array, primitive)

Fig. 1. Graph model of top-level entities

allow us to distinguish the graph elements according to their types, their access parameters, or through other artifacts.

Finally, it is important to mention that as graph databases rely on semi-structured data, they can be easily modified in order to cater to changing requirements. As a result, our model can be easily modified through the addition of new nodes, edges, or properties.

3.2 Querying Through a Workflow-Based Approach

The central focus of our methodology is on supporting users who have no prior knowledge of the underlying graph model, database, or query mechanisms. To do so, we break down larger software measurement concerns into smaller well-managed modules. The focal point of designing modules that process software entities, which we refer to as *Software Fact Extractors*, is to allow users to assemble and adapt an analysis flow comprised of standard building blocks. Thus, the main function of these modules is for users to combine them in a meaningful manner, through pipes that carry data, in order to produce useful software measurement results.

In order to highlight how the above-mentioned modules can be combined in a workflow to produce a software measurement, we present a simple workflow example in Fig. 2, which calculates the number of packages per project. In this figure, blue nodes represent our Software Fact Extractors, while the yellow node symbolizes a data manipulator. Since we are interested in projects that are part of the source analysis and not in external libraries, we configure the “AllProjects” module with the *INTERNAL* access option. From the user’s perspective, this node produces a listing of all the internal projects in the database and connects to the “PackagesInProject” module. However, internally it needs to execute a Cypher query to produce the results. Similarly, the “PackagesInProject” node traverses the *PKG_OF_PROJ* and *PKG_OF_PKG* links of Fig. 1 to provide a project-to-package mapping. Finally, a data manipulation node labeled “Aggregation” groups the projects according to a count of project-to-package mappings.

4 The VIMETRIK Tool

In order to demonstrate our methodology, we have developed an interactive visual data exploration and data-mining tool called VIMETRIK. Our tool is built on top of the KNIME[2] Eclipse plug-in that enables users to model workflows. However, as a pre-requisite to working with our workflows the user needs to extract source code facts, following the methodology described in Sect. 3, into a Neo4j graph database with the help of a pre-packaged library. From the user’s perspective, this is a simple process as they just need to load Java projects into the Eclipse Java Package Explorer and click on “Extract Facts”.

In KNIME, the most common processing unit is a “Node”, which represents any of the visual workflow nodes. Our custom nodes gather the Cypher query results as “DataTables” and pass them to connected nodes for processing or visualization. Additionally, these nodes provide a token that contains the graph database connection details. This approach enables us to process elements of our graph database and at the same time have the full functionality of all the nodes available in KNIME. Interoperability between our Software Fact Extractors in KNIME and the Neo4j database is attained through the management of KNIME’s DataTable data structure and the specification of our graph queries.

Using our VIMETRIK tool, we have developed several workflows that calculate metrics at different levels of the code structure, such as the Chidamber & Kemerer metrics suite [2], the QMOOD metrics suite [1], McCabe’s cyclic complexity measure [8], etc. Figure 3a shows an example of such a workflow.

Once the relevant software metrics are modeled using our workflow-based approach, the measurement results can be analyzed using different coordinated views of the data. On the one hand, users can attach and configure standard KNIME “Data View” nodes that produce traditional views (i.e., parallel coordinates, scatter matrix, or tabular view). On the other hand, users can generate interactive software visualizations via a VIMTERIK “CustomNetworkViewer” node. This node allows the users to configure the type of visualization (i.e., city, sunburst, or hyperbolic), apply some generic settings, and map measurement results as entity properties. Figure 3b shows one of these possibilities, with the results being encoded in a city view [7].

[2] KNIME Analytics Platform (http://www.knime.org/knime/).

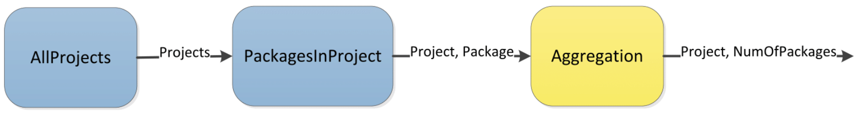
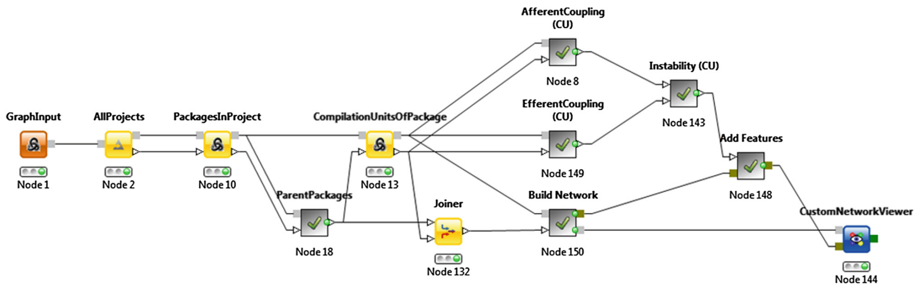
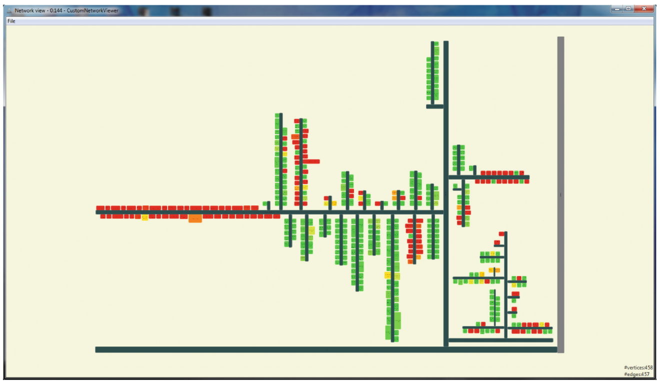


Fig. 2. A simple workflow for number of packages



(a) An example of the VIMETRIK workflow



(b) Custom view of the measurement results

Fig. 3. Integrated analysis and visualization approach

5 The Preliminary Evaluation Study

A preliminary evaluation study was conducted to determine if software analysts could use our proposed methodology to perform software measurement tasks effectively (in terms of completion and accuracy) and, importantly, without dealing with the complexity of a data ontology or querying mechanisms. To validate these claims, we chose experts (21 Master or PhD students, 4 females and 17 males) from the related fields of Software Engineering (SE), Computer Graphics (CG), and Database and Information Systems (DBIS). None of them had any prior experience with VIMETRIK or KNIME;

however, all had object-oriented programming experience and 19 had either SQL or basic query knowledge.

Our hypotheses were that by using our tool, users with no prior experience would be able to achieve a completion rate of at least 85 % (H_1) and an accuracy of at least 85 % (H_2) for basic software measurement tasks. Furthermore, we expected the participants to be highly satisfied (with regard to acceptability and usability) with our tool.

We selected the Apache Tomcat system as the target system for our investigation. The software measurement tasks (size, cyclic complexity, and coupling) were classified into three categories (lightweight, intermediate, and advanced) according to the number of Fact Extractor nodes required for each task. At the end of each test, a closed-ended Likert scale (ranging from 1 *"strongly disagree"* to 5 *"strongly agree"*) questionnaire and an open-ended questionnaire were used to collect user satisfaction feedback and suggestions.

Each subject was given a short tutorial and was asked to complete nine identical tasks. The subjects were awarded 0 % to 100 % for the completion or correctness of each task; however, the completion and correctness task results were weighted according to their difficulty: *lightweight = 1*, *intermediate = 3*, and *advanced = 5*. We applied descriptive statistical methods (i.e., sample mean, standard deviation, and median) to the gathered experiment data. We also tested our hypotheses statistically using the Wilcoxon T test, with a confidence level for rejecting the null hypotheses at 95 %. Due to space restrictions, we present only a summary of the results.

The overall completion percentage for all groups was 94.38 % (*SE = 94.03 %*, *CG = 95.43 %*, and *DBIS = 93.68 %*). The completion hypothesis H_1 was tested using the Wilcoxon T test (*Z = 4.04*, *p = 1.0E-04*), which justified our hypothesis ($p<0.05$; thus, the null hypothesis was rejected). Similarly, the overall correctness percentage for all groups was 94.22 % (*SE = 93.71 %*, *CG = 95.71 %*, and *DBIS = 93.24 %*). The correctness hypothesis H_2 was also tested using the Wilcoxon T test (*Z = 4.12*, *p = 3.78E-5*), which justified our hypothesis ($p<0.05$; thus, the null hypothesis was rejected).

In the closed-ended questionnaires (four questions from the performance and effort expectancy parameters of the TAM model [12] and eleven questions from the utility, intuitiveness, learnability, and personal effect parameters from the work of Nestler et al. [9]), the subjects (irrespective of the user group) gave an average score of 4 out of 5. This indicates that the subjects were highly satisfied with our tool. Furthermore, they provided a few useful suggestions in the open-ended questionnaires that we aim to tackle in the future.

6 Conclusion

This preliminary study shows both the prospects and the feasibility of our proposed solution. In particular, participants with no knowledge of the underlying database model, the querying mechanism, or software analysis in general were able to use our tool to understand software measurement and analysis data. The intuitive and easy-to-use nature of our approach is highlighted by the fact that independent of their expertise, they were able to complete basic software analysis tasks with a completion rate and accuracy of over 85 %. This would not be possible with current tools, which require users to have in-depth knowledge of data ontology and advanced querying skills. Furthermore, all participants were highly satisfied (with regard to usability and acceptance) with our approach.

This work has raised interesting issues for future work. One relevant extension is the addition of modules and mechanisms to include software measurements that handle source code analysis rules, such as those used by PMD, Checkstyle, and FindBugs. Another interesting future work is to monitor the evolution of these measurements in order to support various change requirements. Furthermore, as a consequence of the preliminary results a full-scale comparative study is planned with professional software analysts.

References

1. Bansiya, J., Davis, C.G.: A hierarchical model for object-oriented design quality assessment. IEEE Trans. Softw. Eng. **28**(1), 4–17 (2002)
2. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994)
3. Ebert, J., Bildhauer, D.: Reverse engineering using graph queries. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Nagl Festschrift. LNCS, vol. 5765, pp. 335–362. Springer, Heidelberg (2010)
4. Hajiyev, E., Verbaere, M., de Moor, O.: *codeQuest:* scalable source code queries with datalog. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 2–27. Springer, Heidelberg (2006)
5. Harrison, W.: A flexible method for maintaining software metrics data: a universal metrics repository. J. Syst. Softw. **72**(2), 225–234 (2004)
6. Münch, J., Wickenkamp, A.: M-System NT - Ein flexibles, datenbank-basiertes Mess- und Analyse-System. In: MetriKon, pp. 55–64. Shaker Verlag, Kaiserslautern, 14–16 November 2005
7. Khan, T., Barthel, H., Ebert, A., Liggesmeyer, P.: eCITY: a tool to track software structural changes using an evolving city. In: ICSM, pp. 492–495. IEEE (2013)
8. McCabe, T.J.: A complexity measure. In: ICSE, p. 407. IEEE Computer Society Press, Los Alamitos, (1976)
9. Nestler, S., Artinger, E., Coskun, T., Yildirim, Y., Schumann, S., Maehler, M., Wucholt, F., Strohschneider, S., Klinker, G.: Assessing qualitative usability in life-threatening, time-critical and unstable situations. GMS Med. Inf. Biomed. Epidemiol. 7(1) (2011)
10. Pollo, M., Piattini, M., Ruiz, F.: Advances in Software Maintenance Management: Technologies and Solutions. Idea Group Publishing, Hershey (2003)
11. Roover, C.D., Noguera, C., Kellens, A., Jonckers, V.: The SOUL tool suite for querying programs in symbiosis with Eclipse. In: Probst, C.W., Wimmer, C. (eds.) PPPJ, pp. 71–80. ACM (2011)
12. Venkatesh, V., Morris, M.G., Davis, G.B., Davis, F.D.: User acceptance of information technology: toward a unified view. MIS Q. **27**(3), 425–478 (2003)
13. Winter, V., Reinke, C., Guerrero, J.: Sextant: a tool to specify and visualize software metrics for Java source-code. In: WETSoM, pp. 49–55, May 2013