DOI: 10.15439/2015F238
ACSIS, Vol. 5

Profile-driven Source Code Exploration

Emília Pietriková, Sergej Chodarev
Technical University of Košice, Department of Computers and Informatics
Letná 9, 04200 Košice, Slovak Republic
Email: {emilia.pietrikova, sergej.chodarev}@tuke.sk

***Abstract*—The following study deals with static analysis of Java source codes and it is dedicated to those readers who are interested in techniques aiming at evaluation of programming abilities of job candidates or students. In our case, the goal of the static analysis is to assemble the most significant and interesting data about source code author (programmer). If properly visualized, such assembled data may form programmer's profile which, to impartial observer, may further determine author's real programming abilities and his/her habits, both good and the bad ones. The present study represents first experiments attempting to form programmer's profile by static analysis of language element frequency. Conclusion offers a broader view, combining also other techniques as a future plan to generate knowledge profiles more precisely.**

I. Introduction

KNOWLEDGE, skills and their level are often the focus of attention in many disciplines. In order to be successful, people are often compared with each other. In the area of programming it is similar, however, the range of skills-tracking possibilities is quite limited. In the following study, we present early stages of profile-driven source code analysis where our interest is focused on source code exploration with the intention of knowledge profile generation. Such a profile represents an objective evaluation of current knowledge and skills, individual progress compared to the past, or possible deficiencies to be addressed.

Knowledge profile may be beneficial for both beginners and experienced programmers as well as for lecturers. Profiles can be helpful during overall student assessment, moreover, they can be used when identifying course drawbacks towards improvement of the course. In labor market, job candidates may find programming profile generators beneficial as well. That is, this study is dedicated to those researchers who deal with source code analysis, focusing on author of the code.

There exists a large variety of automated tools dedicated to source code analysis. These tools deal with code from various perspectives, e.g. security evaluation, quality, design etc. Outputs of such tools mostly include reports reflecting various metrics, graphs or warnings. They, however, do not collect a profile of the programmer knowledge [1], [2]. More from the related work can be found in Section V.

In this study, our intention is to generate a programming knowledge profile from source code with a possibility of its comparison with different profiles. This includes comparison of the current profile with a profile which was actual in the past. This way, the profile report may point out author's progress. Profiles of the group of programmers can be compared with each other to reveal possible differences in their knowledge. It should be also possible to compare a personal profile to some explicitly defined knowledge level (e.g. needed to fulfill specific task). We consider tracking and comparing source code in the form of summarizing profiles as a contribution to a new view of knowledge, to a better analysis and filtration of irrelevant data. Yet, to our best knowledge, such a profile-driven tool has not been developed.

In the following sections, we describe the concept of knowledge profiles (Section II) and we introduce an initial prototype proposed and developed as a source code exploring tool (Section III). This tool operates on the basis of static analysis and it represents a partial solution of the presented task. The tool works with Java language constructs and it visualizes knowledge profiles based on various statistics and metrics. We discuss results generated by the tool on a medium-sized project as well as on a large project (Section IV).

II. Knowledge profiles

In general, we understand knowledge profile as a description of knowledge and bindings between its elements necessary to handle a specific task.

In our study, we focus on knowledge profiles in an area where it is possible to formally define such a profile and to construct it automatically from particular input artifacts. Primarily, we deal with an area of programming where the artifacts are represented by source code and a profile is formally defined over a language in which the source code is created. We distinguish two types of profiles: subject and object profile.

Subject profile represents an expression of what the subject (author of the code) knows, how deep is his/her knowledge, what kind of issues is the subject capable to solve. In programming, this means that the subject (programmer) knows, for example, how to use `if` command, how to call or declare a function, how to use generic programming [3]. That is, the subject profile represents the range of tasks actually solvable by the programmer.

Object profile represents a profile of knowledge necessary to handle a specific task (or tasks) over some object. A programming book may define knowledge profile of prerequisites, i.e. what any reader should know before reading the book in order to understand its contents. There can even be a differential object profile determining what is the reader supposed to learn (know after understanding the book contents). Such a differential profile can be determined for each book chapter

978-83-60810-66-8/$25.00©2015, IEEE

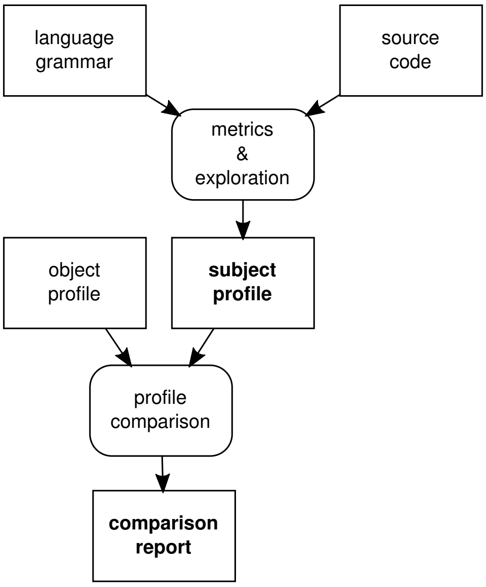


Fig. 1. Simplified knowledge profile generator scheme

as well. That is, the object profile represents the range of tasks which are supposed to be known by the programmer (but the actual state may be different).

The profile should allow to verify whether a specific programmer has sufficient knowledge to solve a task. Moreover, it should identify missing knowledge. For this reason, the profile needs to be structured. Such an assumption is supported by the fact that each programming task or its solution are structured as well [4]. In other words, if an actual (incomplete) task solution is structured than it is possible to assume that the same or similar knowledge, which has already been applied, will be necessary to complete the task.

In an early stage of our research the profile prototype is represented by a simple table, later we assume its transformation to a tree or a graph with annotated nodes or edges [5].

A. *Profile Construction*

A profile can be constructed manually, however, an important part of our research is to generate profiles automatically from artifacts (source code). That is, one profile is supposed to be constructed after processing a finite number of source code files through their analysis. This way it is possible to generate an object profile and also a subject profile provided source code created by the subject is available. For experimental purposes, object profiles may be created manually. In order to construct a subject profile, it is necessary to analyze source code synthesized (created) by the subject.

The idea is depicted using the scheme in Fig. 1. The object profile is optional, so it does not have to be necessarily present. However, language and source code are compulsory. Without these two artifacts, subject profile cannot be generated. If both subject and object profiles are present, a comparison profile can be generated. Such a profile can be bind to a specific task through the object profile.

Source code analysis can be performed through parser of a particular language. An assumption is that particular grammar rules define concepts and the rules used by the code author mean that the programmer understands language constructs describing and defining the language. Complete language syntax is not necessary when processing source code, however, syntax definition should be accustomed to required knowledge expression. An appropriate form of rules should be as expressed in Eq. 1 not 2.

$$If \rightarrow \text{"}\texttt{if}\text{"}\ \text{"("}\ Expression\ \text{")"}\ Statement \tag{1}$$

$$A \rightarrow \text{"}\texttt{if}\text{"}\ \text{"("}\ B\ \text{")"}\ C \tag{2}$$

That is, the form should be human-interpretable, e.g. in order to understand *if*, one should understand expressions and statements.

Obviously, such a naive approach is not sufficient when creating a complex profile. The fact that the code author who called a function might indicate that he understands it, however, one function call does not provide a clear evidence that the subject perfectly understands every detail related to this function. This is why there is a need for metrics definition, based also on empirical observation. In the metrics, we may take into account multiplicity of one method use assuming the more is one method used, the more the subject understands it. Moreover, we may assess method complexity (code length and documentation length) [6]. Such metrics definition represents a separate part of the research regarding knowledge profile generation.

B. *Use cases*

Presented approach towards knowledge profiles generation can find its practical benefits in the following:

- book profile (object profile) – based on subject profile, one may select the most helpful book,
- candidate selection (subject profile) – regarding a task or group of tasks (object profile) supposed to be solved,
- determination of skills necessary to handle some task (object profile) – based on subject profile,
- statistical evaluation of what people frequently use/not use – may indicate the difficulty of use (subject profiles)
- determination of language constructs complexity or library complexity.

The verification of the proposed method of knowledge profile generation can be done within the educational process, e.g. by creating a record of changes in student profile after passing a programming course. Such a record may be beneficial during the exam, indicating student improvement.

As stated in [7], static analysis tools generate lots of data. Therefore, in addition to appropriate techniques of profile creation, two other topics are related and represent a separate part of the research: usability and visualization of knowledge profiles. Assembled data regarding subject or object profile cannot be beneficial if the way of their visualization as well as the user interface are disarranged or too complicated to make any sense.

III. Prototype

To evaluate the concept, a prototype has been implemented, that allows to analyze program source code written in Java language. The prototype represents only the first iteration of our research in the area. It uses the counts of language constructs used in the code to generate a profile. The profile itself is represented as a table containing the counts for each source code file and also summary data. The table is serialized in JSON format.

The data are then visualized in different ways to allow their further examination and comparison.

The tool provides four ways to display profile data:

1) *Detailed tables* — display counts of the used language constructs for each source code file. Constructs are divided into several logical groups (e.g. arithmetic operators or control flow statements) that are displayed in separate tables.
2) *Summary tables* — display summary counts for all files with values of statistical variables like arithmetic mean, modus, median, standard deviation etc, that characterize distribution of a language construct between source code files.
3) *Heat maps* — represent a matrix with total counts for each language construct, where cells of the matrix are colored according the counts (darker color means higher occurrence) and additional statistical data is displayed in a tooltip window (see Fig. 2 that displays comparison of several profiles).
4) *Box plots* also called box-and-whisker plots [8] — visually display summary data together with their distribution (see Fig. 3).

The tool can display simple profiles – data collected for some set of source code files that are produced by single person (subject profile) or are part of a single project (object profile). In addition, there are two compound types of profiles that consist of several simple profiles:

- *group profiles* that display data for several profiles and allows to summarize and compare them,
- *comparison profiles* that allow to compare several profiles with a single master profile.

For the comparison purposes, the most valuable type of display turned out to be *the heat map*. It allows to display a large set of data in a compact form which is easy to explore and therefore allows to visually find anomalies that may indicate significant results.

To implement the parser of Java language, ANTLR parser generator [9] was used. The visualization is based on web technologies such as AngularJS framework[1] and HighCharts interactive graph plotting library[2].

IV. Experiments

We have performed several experiments regarding the developed tool.

A. Analysis of language constructs used in large projects

The tool has been tested on several existing projects, both medium and large. The goals was to assess how Java language constructs are used in them and to test extraction of object profiles.

The results show that in medium-sized projects there is a lot of language constructs that are not used at all. For example, one of the tested projects – YAJCo parser generator [10] – did not adopt any bitwise and bit shift operators, a large part of the arithmetic operators and some other constructs.

Even in a large project, like Google Guava library[3], there are constructs that are never used, including some bit shift operators, try-with-resource blocks, and a default argument for annotation parameters. Bit shift operators, however, have been used in a form of compound assignment operators. On the other hand, try-with-resource was added to the language in version 7, so it has probably been avoided for compatibility reasons.

On the other hand, in the large project most of the Java language constructs was used at least once. This means that profiles based solely on language construct counting would not be comprehensive enough for such projects. For this reason we plan to extend the prototype with advanced analysis and additional metrics.

B. Comparison of student assignments

To evaluate the comparison of profiles, we planned to compare projects of similar size and in the same domain. For this reason we have chosen source code developed by our students as part of their assignments. We used the assignments from the Object-Oriented Programming course. For most students, this course is a first introduction into Java language and object-oriented methodology. This way we were able to compare subjects with similar starting knowledge working on the same problem. We also added a solution developed by a teacher to the comparison.

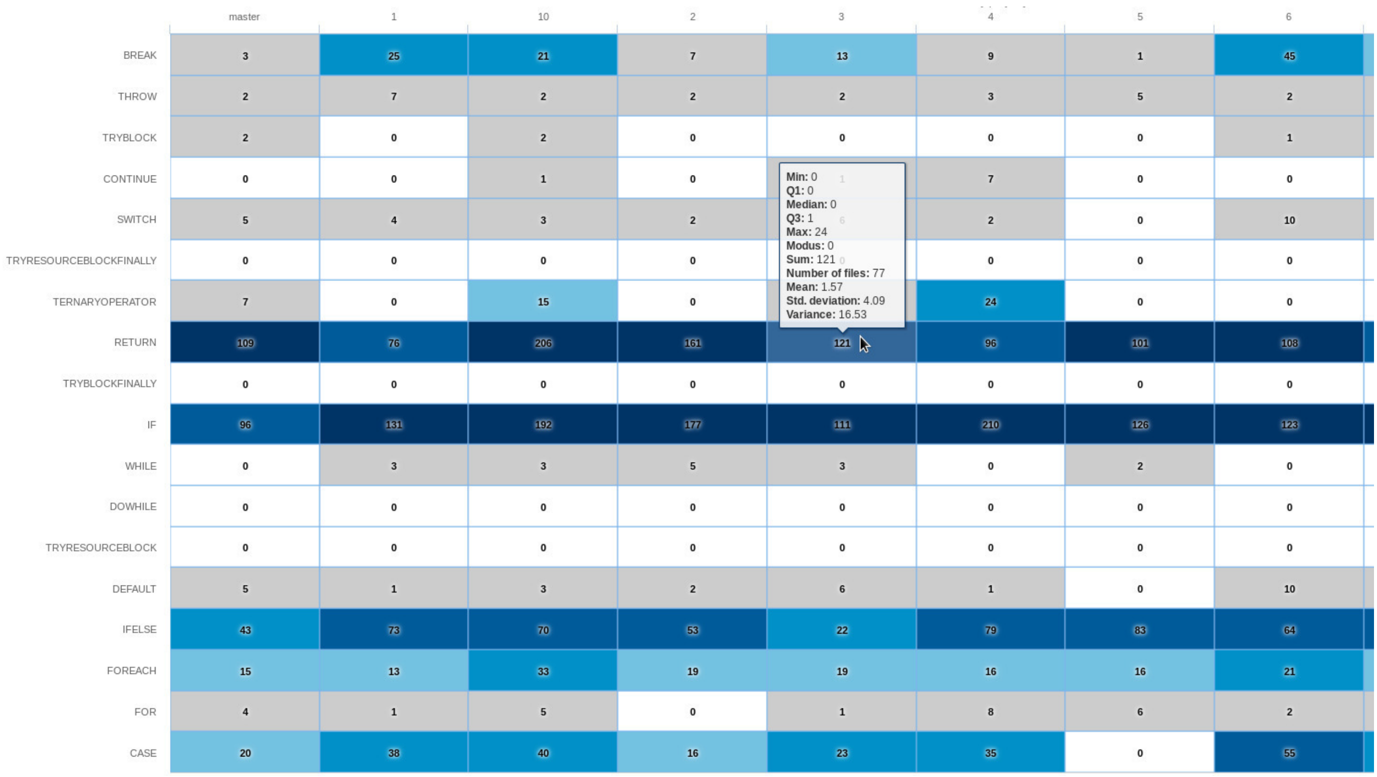
Fig. 2 shows a fragment of the comparison results. Rows correspond to different language constructs, for example *break* statement or *try* block. Columns represent different projects. Students' projects are identified by numbers while teacher's project is called *master*. The table contains a total number of construct occurrences within analyzed codes. After pointing to some table cell, a tooltip window appears containing statistical parameters that represent distribution of the language construct per source code file (see return statement for student 6 in Fig. 2).

The comparison has show that in general the results of both students' and teacher's solution are quite similar. There are, however, some notable differences. For example, student 3 used the largest number of different language constructs,

[1] https://angularjs.org/

[2] http://www.highcharts.com/

[3] https://github.com/google/guava



	master	1	10	2	3	4	5	6
BREAK	3	25	21	7	13	9	1	45
THROW	2	7	2	2	2	3	5	2
TRYBLOCK	2	0	2	0	0	0	0	1
CONTINUE	0	0	1	0	1	7	0	0
SWITCH	5	4	3	2	6	2	0	10
TRYRESOURCEBLOCKFINALLY	0	0	0	0	0	0	0	0
TERNARYOPERATOR	7	0	15	0		24	0	0
RETURN	109	76	206	161	121	96	101	108
TRYBLOCKFINALLY	0	0	0	0	0	0	0	0
IF	96	131	192	177	111	210	126	123
WHILE	0	3	3	5	3	0	2	0
DOWHILE	0	0	0	0	0	0	0	0
TRYRESOURCEBLOCK	0	0	0	0	0	0	0	0
DEFAULT	5	1	3	2	6	1	0	10
IFELSE	43	73	70	53	22	79	83	64
FOREACH	15	13	33	19	19	16	16	21
FOR	4	1	5	0	1	8	6	2
CASE	20	38	40	16	23	35	0	55

Min: 0
Q1: 0
Median: 0
Q3: 1
Max: 24
Modus: 0
Sum: 121
Number of files: 77
Mean: 1.57
Std. deviation: 4.09
Variance: 16.53

Fig. 2. Fragment of the students assignments comparison displayed as a heat map

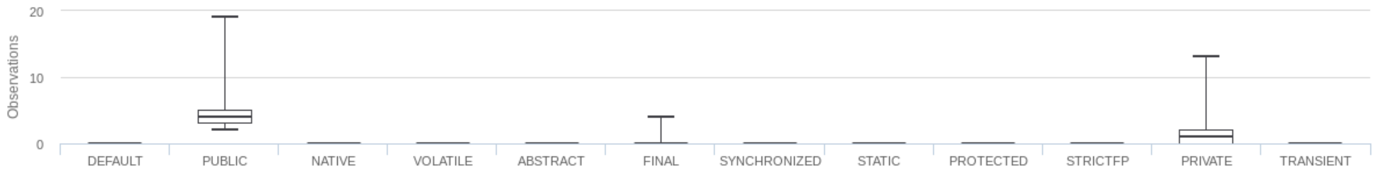


Fig. 3. Example of the box plot displaying counts of modifiers per source code file

even the ones not used by the teacher. Student 7, on the other hand, probably encountered problems with understanding the principles of object-oriented programming, since he used the *static* modifier much more often than the other subjects.

Some students missed language constructs used by all the others. For example, student 1 did not use *float* and *long* types, student 5 did not use *switch* statement. This may indicate that they did not understand these types or constructs, or they simply selected a different implementation strategy. Therefore, exploration of the source codes themselves is needed in both cases.

On the other hand, the fact that student 6 did not use *final* modifier quite clearly indicates that he does not understand the importance of immutability in programs.

V. Related Work

There is a number of studies dealing with source code analysis. Most of them are focused on software security, detecting bugs, defects and potential vulnerabilities. Two of such studies are [11] and [12], both dedicated to static analysis of C/C++ source code. Static analysis tools which are the most popular usually explore static code and identify a large variety of bugs and bad programming practice [13].

Usually, static analysis refers to methods of automated determination of a program behavior during compile time. Static analysis tools have become part of modern compilers, however, these tools can only identify elementary errors [14]. E.g. traditional tools cannot identify the presence of deadlocks, having their own research branch [15], [16], or breaking mutual exclusion in concurrent applications [17], [18].

A method dedicated to collecting, comparing, and combining program semantics is refered to as abstract interpretation and it has been successfully used to derive run time properties of a program which can be used for program optimization. Other objectives of static analysis tools are mostly concept location [19], [20], code transformation [21], [22], security [23], [24], or reverse engineering [25].

When dealing with techniques of the static analysis, one may refer to [26], published a decade ago but still actual, focused on various approaches in software testing based on automata theory. One may also refer to a newer publication

[27], of which authors claim that empirical code evaluation plays an important role in software analysis.

A technique presented in [28] locates computational units typical for a set of related features through execution profiles. In order to detect the most feature-specific computational units, concept analysis is performed [28]. This is combined with static analysis using the feature-specific computational units to detect additional units along with the dependency graph.

Static software analysis has been also covered by a number of surveys, e.g. [29] or [30].

An interesting source of data for programming profile generation may be software repositories, produced and archived throughout software development [31]. In order to explore and examine software repositories, mining software repositories (MSR) have been created. As stated in [32], MSR exploration used to be subjected on industrial systems in the past. However, with an extensive increase of open-source software, this research has become a new challenge. MSR researchers mostly focus on clearer understanding of software evolution [33], development of tools, methods and processes.

Metadata analysis differs with particular exploration objectives and software repositories. The most common issues addressed by MRS researchers are [34]:

- detection of change patterns,
- prediction of changes,
- detection of bugs,
- analysis of bug-fixing change,
- source code exploration,
- identification of software developers.

All of the mentioned issues have one main objective in mind: To augment traditional software engineering techniques in order to guide decision processes in modern software projects [35]. That is, while MSR researchers focus on programming targets (programming result – software), our attention is paid to the source (software author). Since the aim of this study is to assess quality of the code author, source code exploration and developer identification [36] are the most related issues.

VI. Conclusion

Having been first described as a prototype, this study has dealt with an exploration tool aimed at the Java code of various programmers. The main objective is to automatically generate programmer assessment profile. The analysis indicates the topic is quite extensive and little explored. This is why we described only a few of the potential methods for the profile generation.

The proposed tool has been developed and experimentally evaluated on several code samples including a medium-sized and large software project. The tool is intended to analyze knowledge through counting the language constructs. In order to clarify the results, the method utilizes elements of the descriptive statistics [37]. Within the experiment, various possibilities of source code processing have been implemented. For all the processing types, results are available in JSON meta-form as well as in various types of graphical web-based representation.

Since we are still in early stages, the described code-exploring tool and its continuing maturation will include the capability of treating more complex code solutions and utilization of additional metrics. The future plan is to detect and evaluate more advanced language usage e.g. nested loops, or programming idioms [38]. The tool should also track used library classes and method in addition to built-in language constructs. In order to support this, it would be required to implement processing of references in a programming language [39]. In distinction to general search-based techniques, future work will also involve model-based deductive evaluation, similar to [17]. Moreover, if combined with automated code-functionality evaluation during the educational process [40], knowledge profiles may become a significant contribution to student assessment.

Even with the current implementation it is hard to manually analyze and compare large number of profiles. This means that growing amount of data in the profile would require advanced methods of its visualization and automated analysis.

Apparently, automated knowledge evaluation might not be completely accurate. In order to achieve more precise profile results, it will be necessary to perform a lot of experiments over a large group source code and to combine several types of metrics or statistics. However, interesting results will be visible immediately.

Acknowledgment

This work was supported by project VEGA 1/0341/13 “Principles and methods of automated abstraction of computer languages and software development based on the semantic enrichment caused by communication”.

References

[1] D. Mihályi and V. Novitzká, “Towards the Knowledge in Coalgebraic model of IDS,” *Computing and Informatics*, vol. 33, no. 1, pp. 61–78, 2-14.

[2] J. Paralič, F. Babič, and M. Paralič, “Process-driven Approaches to Knowledge Transformation,” *Acta Polytechnica Hungarica*, vol. 10, no. 5, pp. 125–143, 2013.

[3] R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock, “A Comparative Study of Language Support for Generic Programming,” *SIGPLAN Notices*, vol. 38, no. 11, pp. 115–134, 2003. doi: 10.1145/949343.949317

[4] J. Kollár and P. V. Jaroslav Porubän, “Separating concerns in programming: Data, control and actions,” *Computing and Informatics*, vol. 24, no. 5, pp. 441–462, 2005.

[5] M. Nosál’ and J. Porubän, “XML to Annotations Mapping Definition with Patterns,” *Computer Science and Information Systems*, vol. 11, no. 4, pp. 1455–1477, 2014. doi: 10.2298/CSIS130920049N

[6] M. Nosál’, J. Porubän, and M. Nosál’, “Concern-oriented Source Code Projections,” in *Federated Conference on Computer Science and Information Systems (FEDCSIS)*. IEEE, 2013. ISBN 978-1-4673-4471-5 pp. 1541–1544.

[7] S. Heckman and L. Williams, “A Comparative Evaluation of Static Analysis Actionable Alert Identification Techniques,” in *International Conference on Predictive Models in Software Engineering*. ACM, 2013. doi: 10.1145/2499393.2499399 pp. 4:1–4:10.

[8] J. W. Tukey, *Exploratory Data Analysis*. Addison-Wesley, 1977.

[9] T. Parr and K. Fisher, “LL(*): the foundation of the ANTLR parser generator,” *SIGPLAN Notices*, vol. 46, no. 6, pp. 425–436, 2011. doi: 10.1145/1993316.1993548

[10] J. Porubän, M. Forgáč, M. Sabo, and M. Běhálek, "Annotation based parser generator," *Computer Science and Information Systems*, vol. 7, no. 2, pp. 291–307, 2010. doi: 10.2298/CSIS1002291P

[11] R. Huuck, "Technology transfer: Formal analysis, engineering, and business value," *Science of Computer Programming*, vol. 103, pp. 3–12, 2015. doi: 10.1016/j.scico.2014.11.003

[12] V. Ivannikov, A. Belevantsev, A. Borodin, V. Ignatiev, D. Zhurikhin, and A. Avetisyan, "Static analyzer Svace for finding defects in a source program code," *Programming and Computer Software*, vol. 40, no. 5, pp. 265–275, 2014. doi: 10.1134/S0361768814050041

[13] Q. Hanam, L. Tan, R. Holmes, and P. Lam, "Finding Patterns in Static Analysis Alerts: Improving Actionable Alert Ranking," in *Working Conference on Mining Software Repositories*. ACM, 2014. doi: 10.1145/2597073.2597100 pp. 152–161.

[14] V. Djukić, I. Luković, A. Popović, and V. Ivančević, "Model Execution: An Approach based on extending Domain-Specific Modeling with Action Reports," *Computer Science and Information Systems*, vol. 10, no. 4, pp. 1585–1620, 2013. doi: 10.2298/CSIS121228059D

[15] P. T. Breuer and S. Pickin, "One Million (LOC) and Counting: Static Analysis for Errors and Vulnerabilities in the Linux Kernel Source Code," in *Reliable Software Technologies – Ada-Europe*, ser. Lecture Notes in Computer Science. Springer, 2006, vol. 4006, pp. 56–70.

[16] M. Tomášek, "Language for a Distributed System of Mobile Agents," *Acta Polytechnica Hungarica*, vol. 8, no. 2, pp. 61–79, 2011.

[17] Z. Lu and S. Mukhopadhyay, "Model-Based Static Source Code Analysis of Java Programs with Applications to Android Security," in *Computer Software and Applications Conference (COMPSAC)*. IEEE Computer Society, 2012. doi: 10.1109/COMPSAC.2012.43 pp. 322–327.

[18] S. Šimoňák, "Verification of Communication Protocols Based on Formal Methods Integration," *Acta Polytechnica Hungarica*, vol. 9, no. 4, pp. 117–128, 2012.

[19] D. Poshyvanyk, M. Gethers, and A. Marcus, "Concept Location Using Formal Concept Analysis and Information Retrieval," *ACM Transactions on Software Engineering Methodology (TOSEM)*, vol. 21, no. 4, pp. 23:1–23:34, 2013. doi: 10.1145/2377656.2377660

[20] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev, "Static Techniques for Concept Location in Object-Oriented Code," in *International Workshop on Program Comprehension*. IEEE Computer Society, 2005. doi: 10.1109/WPC.2005.33 pp. 33–42.

[21] F. Catthoor, K. Danckaert, S. Wuytack, and N. Dutt, "Code transformations for data transfer and storage exploration preprocessing in multimedia processors," *Design Test of Computers*, vol. 18, no. 3, pp. 70–82, 2001. doi: 10.1109/WPC.2005.33

[22] A. C. Murray, R. V. Bennett, B. Franke, and N. Topham, "Code Transformation and Instruction Set Extension," *ACM Transactions on Embedded Computer Systems (TECS)*, vol. 8, no. 4, pp. 26:1–26:31, 2009. doi: 10.1145/1550987.1550989

[23] A. Baláž, *Computer Systems Security*, 2nd ed., 2015. ISBN 978-80-553-1948-3

[24] L. Vokorokos, A. Baláž, and N. Ádám, "Secure web server system resources utilization," *Acta Polytechnica Hungarica*, vol. 12, no. 2, pp. 5–19, 2015.

[25] H. M. Kienle and H. A. Müller, "Rigi — An Environment for Software Reverse Engineering, Exploration, Visualization, and Redocumentation," *Science of Computer Programming*, vol. 75, no. 4, pp. 247–263, 2010. doi: doi:10.1016/j.scico.2009.10.007

[26] G. J. Holzmann, "Software Analysis and Model Checking," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2404, pp. 1–16.

[27] M. B. Dwyer, J. Hatcliff, R. Robby, C. S. Pasareanu, and W. Visser, "Formal Software Analysis Emerging Trends in Software Model Checking," in *Future of Software Engineering*. IEEE Computer Society, 2007. doi: 10.1109/FOSE.2007.6 pp. 120–136.

[28] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 210–224, 2003. doi: 10.1109/TSE.2003.1183929

[29] P. Emanuelsson and U. Nilsson, "A Comparative Study of Industrial Static Analysis Tools," *Electronic Notes in Theoretical Computer Science*, vol. 217, pp. 5–21, 2008. doi: 10.1016/j.entcs.2008.06.039

[30] S. Heckman and L. Williams, "A Systematic Literature Review of Actionable Alert Identification Techniques for Automated Static Code Analysis," *Information and Software Technology*, vol. 53, no. 4, pp. 363–387, 2011. doi: 10.1016/j.infsof.2010.12.007

[31] S. O. Olatunji, Y. S. Al-Ghamdi, and J. S. A. Al-Ghamdi, "Mining Software Repositories – A Comparative Analysis," *International Journal of Computer Science and Network Security*, vol. 10, no. 8, pp. 161–174, 2010.

[32] H. Kagdi, M. L. Collard, and J. I. Maletic, "A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution," *Journal of Software Maintenance and Evolution: Research and Practice*.

[33] J. Kollár and M. Forgáč, "Combined approach to program and language evolution," *Computing and Informatics*, vol. 29, no. 6+, pp. 1103–1116, 2010.

[34] K. Chaturvedi, V. Sing, and P. Singh, "Tools in Mining Software Repositories," in *Computational Science and Its Applications (ICCSA)*, 2013. doi: 10.1109/ICCSA.2013.22 pp. 89–98.

[35] A. Hassan, "The road ahead for Mining Software Repositories," in *Frontiers of Software Maintenance*, 2008. doi: 10.1109/FOSM.2008.4659248 pp. 48–57.

[36] S. Koch and G. Schneider, "Effort, cooperation and coordination in an open source software project: Gnome," *Information Systems Journal*, vol. 12, no. 1, pp. 27–42, 2002. doi: 10.1046/j.1365-2575.2002.00110.x

[37] W. Trochim, "Research methods knowledge base: Descriptive statistics," http://www.socialresearchmethods.net/kb/statdesc.php, 2006, Accessed: 2015-04-30.

[38] A. Sutton, R. Holeman, and J. I. Maletic, "Identification of idiom usage in C++ generic libraries," *International Conference on Program Comprehension*, pp. 160–169, 2010. doi: 10.1109/ICPC.2010.37

[39] D. Lakatoš, J. Porubän, and M. Bačíková, "Declarative specification of references in DSLs," in *Federated Conference on Computer Science and Information Systems (FedCSIS 2013)*. IEEE, 2013. ISBN 978-1-4673-4471-5 pp. 1527–1534.

[40] M. Biňas, "Improving reliability of arena platform for automated assessments," in *Electrical Engineering and Informatics 5: Proceedings of FEEI*, 2014, pp. 115–118.