TOWARDS PROGRAMMER KNOWLEDGE PROFILE GENERATION

Emília PIETRIKOVÁ, Sergej CHODAREV
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, Slovak Republic, Tel.: +421 55 602 2554, E-mail: {sergej.chodarev, emilia.pietrikova}@tuke.sk

ABSTRACT
The following article deals with static analysis of source code in Java and it is intended for readers interested in techniques focused on evaluation of programming abilities of students or potential job candidates. The main objective of the static analysis is to collect the most relevant and significant data about programmers. If such data is properly visualized, it can result in knowledge profile which further determines programmer's real programming abilities as well as his habits. This can be useful mainly for impartial observer who does not know the code author. In the following article we present our first attempts to create and visualize knowledge profiles through static analysis and statistics regarding frequency of language elements. In perspective, the conclusion combines advanced techniques towards creation of more precise profiles as the future work.

Keywords: *knowledge profile, programmer's profile, static code analysis, descriptive statistics, language element frequency*

1. INTRODUCTION

In many disciplines, the level of knowledge or skills is a stumbling-block, creating a competitive environment. Similar issue is visible in programming, though the variety of one's skills evaluation are quite limited. In this article, we introduce prototype of knowledge profile generator through (yet) static source code analysis. The main interest resides in the source code exploration with an objective evaluation of one's actual knowledge and programming abilities, individual progress compared to the past, or possible weaknesses to be addressed.

Both beginners and experienced programmers can benefit from such a knowledge profile. Moreover, profiles can be helpful for lecturers throughout overall student assessment or while identifying potential shortcomings of the course. Other benefiting areas are in labor market, offering an adequate evaluation of job candidates. I.e. we devote this article to researchers focusing on source code analysis and the code author(s).

There is a number of tools, both automated and semi-automated, dealing with source code analysis. Mostly, the main objective is to evaluate software security, quality or design, and the main result includes a report which includes various metrics or graphs. Since such tools deal with code regarding the final product, they do not focus on its author (programmer).

In the area of software security, some studies detect bugs, defects and other vulnerabilities, e.g. [1] and [2], both performing static analysis of C/C++ source code. Other studies explore static code and identify various bugs as well as bad programming practice [3].

Modern compilers include static analysis tools, usually referring to methods of automated determination of program behavior during compile time. Since traditional tools identify only simple errors, some studies are dedicated to identification of deadlock presence [4], others deal with breaking of mutual exclusion in concurrent applications [5].

A technique of program assembling, comparison, and combining, known as abstract interpretation, has been successfully used to derive run time properties of a program, used in source code optimalization. Other goals of static analysis mostly include code transformation [6, 7], concept location [8, 9] or reverse engineering [10].

A method presented in [11] introduced location of computational units via execution profiles, typical for a set of related features. The authors of this study performed concept analysis resulting in detection of the most feature-specific computational units. Combination of these units with static analysis resulted in detection of additional units along with the dependency graph. Moreover, static code analysis has been also the subject of several surveys, e.g. [12] or [13].

Exploration and examination of software repositories formed the research area of mining software repositories (MSR). In the past, MSR examination was focused on industrial systems [14]. However, the popularity of open-source software led to challenges of clearer understanding of tool development, methods, processes and software evolution [15].

Depending on particular exploration objectives and software repositories, analysis of metadata is always different. The main issues include [16]: Detection of change patterns, prediction of changes, detection of bugs, analysis of bug-fixing change, source code exploration, or identification of software developers.

All the mentioned issues have one common objective: To enhance traditional techniques of software engineering towards processes of guide decision in modern software projects [17]. While MSR researchers deal with programming targets (programming result – software), our research is dedicated to the source (software author). Our aims include assessment of the code author, so the source code exploration and developer identification, approached in [18], are the most related issues.

In this article we describe creation of knowledge profiles from programmers' source codes where every profile can be compared with other profiles. In our experiment, we compare actual profile with older profiles, indicating programmer's improvement. Moreover, we compare a group of different programmers in order to highlight differences in their skills. In our vision, it should be possible to compare profiles to specific levels of knowledge as well, e.g. necessary to perform a specific task.

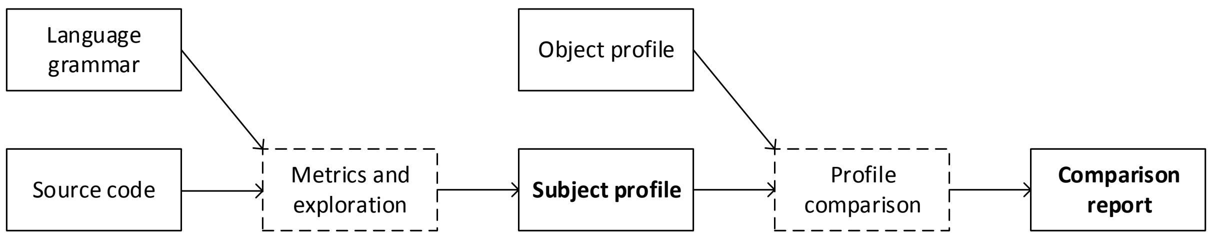


Fig. 1 Main idea of knowledge profile generator

We believe that comparison of source codes in the form of knowledge profiles is the main scientific contribution. We perceive knowledge as an option to perform a better analysis and filter any irrelevant data. According to the literature overview and to our best knowledge, such a profile-creating tool has not yet been developed.

In the following sections, we define the concept of knowledge profile, sec. 2, and we introduce a prototype of profile generator, sec. 3. The generator is based on static analysis and yet it deals partially with the presented task. It analyses the use of language constructs of Java and creates profiles (including visualization) through various metrics and statistics. In sec. 4, we discuss results achieved by the prototype within an experiment performed on student assignments. Conclusion remarks deal mainly with the future version of the profile generator, sec. 5.

2. PROGRAMMER KNOWLEDGE PROFILE

Knowledge profile delineates skills, abilities and bindings among their elements which are required to perform some task. In general, we admit various abstraction degrees of a profile definition (knowledge/skills). E.g. (outside the area of programming) to know how to saw, to know how to saw by a chainsaw, to know how to saw by a chainsaw if the wood is of a thinner diameter.

We suppose that it is hard to differentiate knowledge of similar concepts. If so, we rely on understanding the issue in most usual cases. E.g. (in the area of programming) if a programmer has proved he knows how to work with conditional expressions within `if`, we can assume that he knows how to work with conditional expressions within `while` or `for`.

In our perception, we are able to formally define knowledge profile and create it implicitly provided there is sufficient input information. In the area of programming, input information is represented by source code and a profile is formally set over particular programming language (Java in our case). We differentiate two profiles:

- *Subject profile* – Expresses what the author of the code (the subject) understands and the range of tasks he is able to solve. In our case, the programmer should know, e.g. how to declare or call a method, and even how to use an annotation [19].
- *Object profile* – Expresses what is required to solve a task or tasks. We can define object profile for an educational course or a programming book, consisting of prerequisites, i.e. what one should already understand before attending the course or reading the book. We can even create a distinguished object profile expressing what one should understand after attending the course or reading the book. In other words, object profile represents an expected knowledge profile while the reality may be different. However, if the subject profile is supposed to be general, object profile becomes optional.

One knowledge (subject) profile is expected to be generated after processing (analyzing) a finite number of source code files. Object profile can be constructed both manually or automatically, based on completely or partially solved tasks.

Fig. 1 illustrates the main idea. The object profile is optional, however, language definition and source code are mandatory. If both subject and object profiles are created, we can generate a comparison report.

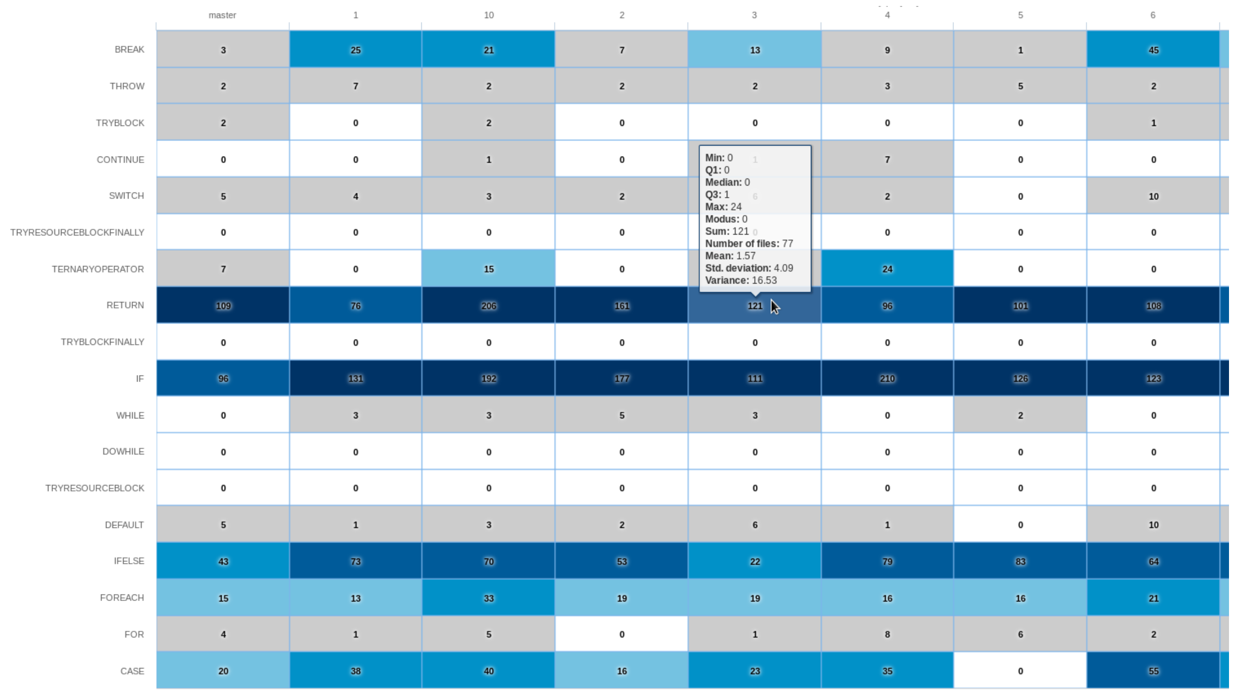
By creating a subject profile, we can determine whether the programmer has enough knowledge to handle some task as well as we can identify any missing knowledge. Since each programming task or its solution is structured, the profile is required to be structured as well.

Currently, results of the profile generator are visualized within a table of data. In later stages of the research we plan its transformation to a graph or a tree containing annotated edges or nodes [19].

In order to analyze the source code, is it possible to use language parser. Since rules of the language grammar define concepts, we assume that if a programmer (subject) uses particular rules, then he understands constructs which describe and define the programming language. Source code exploration (Fig. 1) does not require to use complete language syntax but rules necessary to create a profile. However, an appropriate form of rules should be human-interpretable, i.e. Eq. 1 is better than Eq. 2, expressing that in order to understand *while*, one should understand both expressions and statements.

$$While \rightarrow \text{"}\texttt{while}\text{"}\ \text{"("}\ Expression\ \text{")"}\ Statement \tag{1}$$

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



	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	[illegible]	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 Heat map: Comparison of student assignments

Moreover, when generating a complex profile, we cannot rely on a fact that the programmer understands something after one occurance. That is, we need to define metrics including both facts and empirical observation. E.g. if one class contains 20 methods, then the *understanding* may be derived as: `10 + 4 × number_of_used_methods`, i.e. if the subject has used every method (out of 20) at least once, then `10 + 4 × 20 = 100`, so he fully understands the class. We may also assume multilicity: The more is something used, the more the programmer understands it, or complexity: The longer is the code (or documentation), the more the programmer understands it. Regarding profile generation, such metrics definition is a separate part of our research.

3. PROFILE GENERATOR

Prototype of the proposed profile generator allows to process Java code, counting particular language constructs and generating profile as a table with summary data. Then, the data is visualized in various forms (currently four), so it is possible to further examine and compare the data:

- *Detailed table* – For every source code, it contains usage frequency of language constructs, all divided to logical groups in separate tables, e.g. of arithmetic operators.
- *Summary table* – Contains summary data for all source code files regarding distribution of language constructs, e.g. arithmetic mean, modus, median, or standard deviation.
- *Heat map* – For every language construct, this matrix consists of cells colored by occurrence (the darker the color, the higher the frequency). Additional tooltip window contains additional statistical data.
- *Whisker plot* – Graph illustrating summary data and its distribution [20].

In order to process Java source codes, we use ANTLR (parser generator, [21]), creating tables serialized in JSON format. Results are visualized through web interface, based on AngularJS framework and HighCharts library.

Currently, the profile generator creates simple profiles containing data assembled for some group of source codes created by one programmer (subject profile) or data of a single project (object profile). If comparing various data or programmers, *heat maps* have proved to be the most useful, since they allow to display a lot of data at one place.

4. RESULTS

In order to verify the proposed method of knowledge profile generation, we have measured subject abilities based on his profile and determined whether he is suitable to solve a specific task. A correct determination of what the subject knows or not may be influenced by the following:

- Subject evaluates himself (through a questionnaire),
- Subject is issued a task and his experience is assessed (question, program fragment, program synthesis),
- Subject profile is generated.

In our case, we decided to verify the proposed approach within the educational process by tracking changes in a student profiles. Student assignments are programming projects of similar size within the same domain. We collected assignments of the OOP course, introducing Java laguage as well as object-oriented paradigm. Except that students were supposed to work on the same problem, within the experiment we assumed that students (subjects) had similar dispose of knowledge. To be more precise, student profiles were compared with lecturer profiles as well.

Fig. 2 displays comparison of several profiles (part of the overall results). Rows represent different language elements, e.g. *break* or *try*. Columns correspond to different programming projects (assignments). Students are labeled by numbers while teacher is labeled as *master*. For every source code, the table contains language element occurrence complemented by statistics (displayed as a tooltip window after pointing to a table cell).

Although profiles of students and the master are rather similar, a careful reader may have noticed some notable differences. E.g. student 3 used the highest number of various language elements (also those not used by the master) while student 7 may have encountered difficulties with understanding the principles of object-oriented paradigm as the *static* modifier was used much more often than in other profiles. Some students did not use language elements frequent in other profiles. E.g. student 5 missed *switch* while student 1 missed *float* and *long*. These students could both not understand these types or constructs, or they just inclined to a different way of solution. That is, in some cases, further exploration of the source codes is necessary. On the other hand, some data can clearly indicate weaknesses, e.g. student 6 did not use *final* modifier, i.e. he does not understand the importance of immutability in programs.

5. CONCLUSION REMARKS

We introduced an approach towards creation of programmer knowledge profiles through profile generator, exploring Java source codes. The aim is to implicitly create such profile. Despite this topic is relatively extensive, the analysis revealed it is little explored. There exists a large variety of potential methods, yet we focused on static analysis, language element frequency and descriptive statistics [22]. Authors of [13] claim that tools based on static analysis create a lot of data. This is why there are three relevant research topics: methods of profile generation, usability of profiles and visualization of profiles.

We also described profile generator tool and experimentally explored student assignments. The results showed the tool counts frequency of particular language elements using descriptive statistics [22] and visualizes assembled data in various tables, heat maps and whisker plots (available also in JSON meta-form). The statistics can show the subject is familiar with particular elements, yet it does not implicitly mean he/she is using them correctly. The same applies to unused elements. The fact that the subject did not use particular element does not mean he/she is not familiar with it. Thus, more appropriate metrics should be proposed, otherwise manual code exploration will always be necessary.

Nevertheless, presented approach can be applied in the following areas: Course or book profile (object profile based on subject profile, selection of the most useful course or book), candidate profile (subject profile based on object profile, what is required to be solved), skills profile and student assessment [23] (object profile based on subject profile, consisting of abilities necessary to solve some task), statistics (subject profile, frequency evaluation of a language construct indicating its difficulty), or complexity profile (complexity evaluation of a language construct or a library). Obviously, comparison of student profiles with each other may reveal plagiarism.

Yet, the profile generator has proved to be an interesting tool to assess programmers and it is ready to be extended towards treating more comprehensive programming projects (e.g. model-driven software development [24]) and applying advanced metrics. Further research iterations will enhance the tool, so it will be able to identify various patterns in programmer behavior [25], or to detect and assess advanced language usage, e.g. programming idioms [26] or nested loops, or to deal with security issues [27]. In addition to language elements, we also plan to track the usage of library classes and methods. Comparison of student profiles will involve multiple master profiles. Future work will also include model-based assessment similar to [5] or reference processing in a programming language [28].

ACKNOWLEDGEMENT

This work was supported by project KEGA 047TUKE-4/2016 "Integrating software processes into the teaching of programming."

REFERENCES

[1] HUUCK, R.: *Technology transfer: Formal analysis, engineering, and business value*, Science of Computer Programming **103**, No. 1 (2015) 3–12.

[2] IVANNIKOV, V. – BELEVANTSEV, A. – BORODIN, A. – IGNATIEV, V. – ZHURIKHIN, D. – AVETISYAN, A.: *Static analyzer Svace for finding defects in a source program code*, Programming and Computer Software **40**, No. 5 (2014) 265–275.

[3] HANAM, Q. – TAN, L. – HOLMES, R. – LAM, P.: Finding Patterns in Static Analysis Alerts: Improving Actionable Alert Ranking, ACM Working Conference on Mining Software Repositories, 2014, pp. 152–161.

[4] BREUER, P. – PICKING, S.: *Reliable Software Technologies – Ada-Europe*, Lecture Notes in Computer Science: One Million (LOC) and Counting: Static Analysis for Errors and Vulnerabilities in the Linux Kernel Source Code **4006**, No. 1 (2006) 56–70 Springer.

[5] ZHENG, L. – MUKHOPADHYAY, S.: Model-Based Static Source Code Analysis of Java Programs with Applications to Android Security, Computer Software and Applications Conference (COMPSAC), 2012, pp. 322–327, IEEE Computer Society.

[6] CATTHOOR, F. – DANCKAERT, K. – WUYTACK, S. – DUTT, N.: *Code transformations for data transfer and storage exploration preprocessing in multimedia processors*, Design Test of Computers **18**, No. 3 (2001) 70–82 IEEE Computer Society.

[7] MURRAY, A. – BENNETT, R. – FRANKE, B. – TOPHAM, N.: *Code Transformation and Instruction Set Extension*, ACM Transactions on Embedded Computer Systems (TECS) **8**, No. 4 (2009) 26:1–26:31.

[8] POSHYVANYK, D. – GETHERS, M. – MARCUS, A.: *Concept Location Using Formal Concept Analysis and Information Retrieval*, ACM Transactions on Software Engineering Methodology (TOSEM) **21**, No. 4 (2013) 23:1–23:34.

[9] MARCUS, A. – RAJLICH, V. – BUCHTA, J. – PETRENKO, M. – SERGEYEV, A.: Static Techniques for Concept Location in Object-Oriented Code, International Workshop on Program Comprehension, 2005, pp. 33–42, IEEE Computer Society.

[10] KIENLE, H. – MÜLLER, H.: *Rigi — An Environment for Software Reverse Engineering, Exploration, Visualization, and Redocumentation*, Science of Computer Programming **75**, No. 4 (2010) 247–263 Elsevier.

[11] EISENBARTH, T. – KOSCHKE, R. – SIMON, D.: *Locating features in source code*, IEEE Transactions on Software Engineering **29**, No. 3 (2003) 210–224.

[12] EMANUELSSON, P. – NILSSON, U.: *A Comparative Study of Industrial Static Analysis Tools*, Electronic Notes in Theoretical Computer Science **217**, No. 1 (2008) 5–21 Elsevier.

[13] HECKMAN, S. – WILLIAMS, L.: *A Systematic Literature Review of Actionable Alert Identification Techniques for Automated Static Code Analysis*, Information and Software Technology **53**, No. 4 (2011) 363–387 Butterworth-Heinemann.

[14] KAGDI, H. – COLLARD, M. – MALETIC, J.: *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 **19**, No. 2 (2007) 77–131 Wiley & Son.

[15] KOLLÁR, J. – FORGÁČ, M.: *Combined approach to program and language evolution*, Computing and Informatics **29**, No. 6 (2010) 1103–1116.

[16] CHATURVEDI, K. – SING, V. – SINGH, P.: Tools in Mining Software Repositories, Computational Science and Its Applications (ICCSA), 2013, pp. 89–98.

[17] HASSAN, A.: The road ahead for Mining Software Repositories, Frontiers of Software Maintenance, 2008, pp. 48–57.

[18] KOCH, S. – SCHNEIDER, G.: *Effort, cooperation and coordination in an open source software project: Gnome*, Information Systems Journal **12**, No. 1 (2002) 27–42.

[19] NOSÁL', M. – PORUBÄN, J.: *XML to Annotations Mapping Definition with Patterns*, Computer Science and Information Systems **11**, No. 4 (2014) 1455–1477.

[20] TUKEY, J.: Exploratory data analysis, 1977, ISBN 978-0201076165, Addison-Wesley.

[21] PARR, T. – FISHER, K.: *LL(*): The Foundation of the ANTLR Parser Generator*, ACM SIGPLAN Notices **46**, No. 6 (2011) 425–436.

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

[23] BIŇAS, M. – PIETRIKOVÁ, E.: Useful recommendations for successful implementation of programming courses, IEEE International Conference on Emerging eLearning Technologies and Applications (ICETA), 2014, pp. 397–401.

[24] PORUBÄN, J. – BAČÍKOVÁ, M. – CHODAREV, S. – NOSÁL', M.: Pragmatic model-driven software development from the viewpoint of a programmer: Teaching experience, IEEE Federated Conference on Computer Science and Information Systems (FedCSIS), 2014, pp. 1647–1656.

[25] KOLLÁR, J. – CHODAREV, S. – PIETRIKOVÁ, E. – WASSERMANN, L'.: Identification of patterns through Haskell programs analysis, IEEE Federated Conference on Computer Science and Information Systems (FedCSIS), 2011, pp. 891–894.

[26] SUTTON, A. – HOLEMAN, R. – MALETIC, J.: Identification of idiom usage in C++ generic libraries, International Conference on Program Comprehension, 2010, 160–169, IEEE Computer Society.

[27] BALÁŽ, A.: Computer systems security, 2015, Ed. 2, ISBN 978-8055319483.

[28] LAKATOŠ, D. – PORUBÄN, J. – BAČÍKOVÁ, M.: Declarative specification of references in DSLs, IEEE Federated Conference on Computer Science and Information Systems (FedCSIS), 2013, pp. 1527–1534.

Received October 27, 2015, accepted April 26, 2016

BIOGRAPHIES

Emília Pietriková is Assistant Professor at the Department of Computers and Informatics, Technical University of Košice, Slovakia. She received her MSc in 2010 and her PhD in 2014 in Informatics from Technical University of Košice. In 2010 she spent 1 month at the Department of Telematics at Norwegian University of Science and Technology, Norway. In 2011 she spent 1 semester at the Department of Computer Architecture at University of Málaga, Spain. The subject of her research is abstraction and generation in programming languages, and quality of education.

Sergej Chodarev is Assistant Professor at the Department of Computers and Informatics, Technical University of Košice, Slovakia. He received his MSc in Computer Science in 2009 and his PhD in Informatics in 2012 from Technical University of Košice. In 2011 he spent 1 semester at the Department of Computer Architecture at University of Málaga, Spain. The subject of his research is domain-specific languages, metaprogramming and programming paradigms.