On development of a framework for massive source code analysis using static code analyzers

Alexander Chistyakov
ITMO University
Saint-Petersburg, Russia
alexclear@gmail.com

Artem Pripadchev
ITMO University
Saint-Petersburg, Russia
artem.pripadchev@outlook.com

Irina Radchenko
ITMO University
Saint-Petersburg, Russia
iradche@gmail.com

ABSTRACT

Authors describe architecture and implementation of an automated source code analyzing system which uses pluggable static code analyzers. The paper presents a module for gathering and analyzing the source code massively in a detailed manner. Authors also compare existing static code analyzers for Python programming language. A common format of storing results of code analysis for subsequent processing is introduced. Also, authors discuss methods of statistical processing and visualizing of raw analysis data.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

code analysis, open source, static analyzers

ACM Reference Format:
Alexander Chistyakov, Artem Pripadchev, and Irina Radchenko. 2017. On development of a framework for massive source code analysis using static code analyzers. In *Proceedings of SECrus 2017 (SECRUS'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Informational technology field is one of the fastest growing industries today. Without doubt, using various automated systems to replace human labor especially when doing repeatable operations is very useful. It increases effectiveness of work and removes the possibility of accidental human-induced errors. However, automated systems are not error-free per se. A risk of deterministic error in program code induced not by a worker on a conveyor but by a program creator arises [20].

Global industrial community is very concerned about a possibility of appearance of various types of defects in program code. To address this a number of international standards covering software development process were developed (ISO/IEC 90003:2014, CMM/CMMI) [8]. Moreover, a lot of various methodologies and approaches to development of different types of computer systems were created in past decades. Every such methodology aims for getting a working software product in timely fashion. Some methodologies emphasize sequence of steps of software development process, others make development process simple and agile. Yet every such methodology targets creating a product of good quality.

Software engineers need to define a set of measurable parameters to control the overall process of developing a product. This can be expanded to quality of code too. It's hard to make any project management decisions in lack of quantitative characteristics. Thus a problem of measuring code quality is actual nowadays.

The term "quality" is quite complex and multi-dimensional. "Quality" usually means compliance of object properties to a set of predefined requirements [9]. Quality of program code implies thoroughly designed architecture, clear division of code into functional submodules, defining strict structure and so on. Software engineers use various methodologies and techniques to improve code quality, such as using design patterns, utilizing existing libraries and algorithms for solving typical tasks and so on [5].

But do we have to care about code quality if end users demand another thing - the overall quality of a product? Yes, we obviously do, because every complex information system is a subject to evolution and modification. In this case important metrics are number of defects in program code and a cost of modification of code. If adding new functionality introduces a critical number of errors the product is not able to fulfill customer needs anymore. In the same way, if a cost of adding new functionality to the product is too high it will affect users negatively too. Therefore, code quality is not directly related to functionality but nonetheless important parameter which indirectly relates to the overall program product quality.

Since code quality affects overall computer program product quality severely we have got an idea to develop an automated code analysis system based on existing static code analyzers.

Our idea is to use Github [7] as a provider of Python code repositories of different size and code quality. We are going to build a system suitable for performing massive code analysis using a predefined set of existing static code analyzers. We chose Python mainly for two reasons: firstly, because it's fourth most popular programming language on Github and secondly, because number of already developed static code analyzers for Python is relatively big. Also, we can program in Python and will be able not only fix errors in existing Python-based tools but develop our own implementation of a static code analyzer if needed.

Our goal is to gather some raw data in result of processing code repositories from Github using static analyzers and to normalize this raw data using a developed common data format. Next step is to perform statistical analysis of this set of normalized raw data

SECRUS'17, October 2017, St.Petersburg, Russia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

using classification and clusterization and other data processing algorithms.

2 RELATED WORKS

Static code analysis is analysis of code conducted without real program execution [12]. Result of such analysis in this paper is a certain analytics which can be used to get a representation of code quality. Static code analysis is also being used for other purposes. So, authors of [18] classify android applications into two types: utilities and games using machine learning. Many papers are related to finding possible vulnerabilities in programs during development stage [1], [2], [19]. Authors of [13] extract code characteristics to find defects subsequently. Another problem domain of static code analysis is automated detection of malicious code [15].

3 DESCRIPTION OF EXPERIMENT

In order to determine qualitative characteristics of program code, we should define a set of measurable parameters first. Metrics of program code can be used as these parameters. Essentially, this method analyzes source code to get various numeric metrics. Usually these metrics are defined based on analyzing either a control flow graph or a structure of program code [3].

There are a big number of metrics representing various program code aspects as of today. Most common metrics are number of SLOC, cyclomatic complexity, number of warnings and errors and so on. A benefit of using metrics of program code is absence of human factor. These metrics are measured by a computer. This fact guarantees precision and repeatability of measurements for every metric. Moreover, it becomes possible to measure these metrics automatedly and to create various analytic reports based on these automated measurements.

Proposed method of measuring code quality metrics is presented in a form of block diagram on Fig. 1.

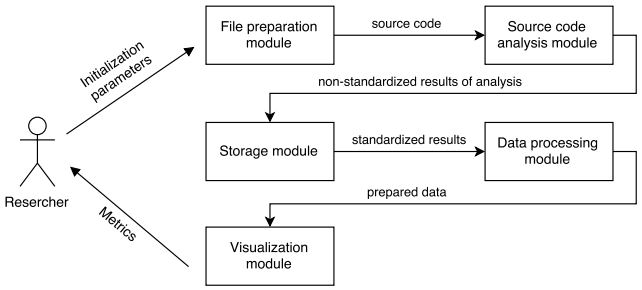

Figure 1: Block diagram of a framework for massive source code analysis using static code analyzers

The file preparation module is an entry point for a user. The user defines all necessary environment for subsequent analysis. This environment consists of a folder with project source code to be analyzed, a set of analyzers to utilize, a set of excluded analyzer rules and so on.

After setting up the environment and starting the application system begins to perform source code analysis. It's worth to mention that we chose Python as a reference programming language, mainly, because Python has a lot of scientific libraries for data processing, visualization and so on. We decided to start off with a preexisting set of source code analyzers and then write our own custom solution if it becomes needed. Thus, our first task was to compare existing open source implementations of static source code analyzing tools for Python language.

4 TOOLS

There is only a limited number of code analysis frameworks for Python with an ability to plug different static code analyzers. Namely, they are Coala [4], Pylama [17] and Flake8 [6].

We defined the following set of parameters to compare these frameworks: popularity among Github [7] users, CPU and RAM utilization, ability to parallelize process of analysis and time required to process the same set of projects.

Since Flake8 is designed to perform style guide enforcements only and does not have an ability to disable standard plugins easily we excluded it from further comparison.

Coala provides a uniform CLI interface for code style checking and code improvement. Coala uses a set of plugins (called "bears") for various programming languages. It's also possible to extend a standard set of plugins with a custom plugin.

Pylama is a code auditing tool for Python and JavaScript programming languages. It is not as feature rich as Coala due to lesser popularity on Github and lower number of active contributors and code commits.

We used a virtual machine with 3Gb of RAM, 3 CPU cores and Ubuntu 14.04 installed as a test host.

5 RESULTS

We chose a relatively small sample project (consisted of roughly 130 files) and configured two plugins in both Coala and Pylama. Processing time took 112 seconds for Coala and 183 seconds for Pylama.

But it turned out that the most important metric was RAM utilization. We created a graphical representation of OS memory usage on Fig. 2.

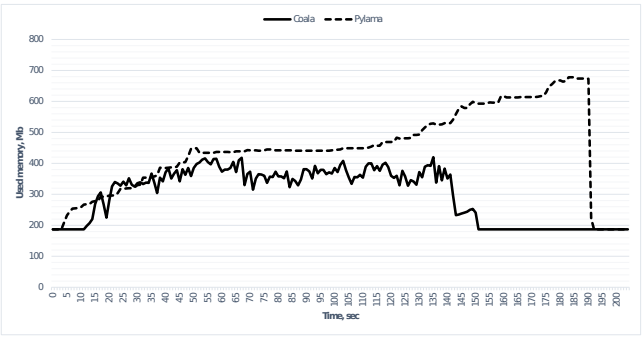

Figure 2: OS memory usage when analyzing a single project by Coala and Pylama

Pylama constantly accumulates information about errors so needed RAM grows almost linearly. This leads to a critical defect that reveals as total RAM exhaustion. Operating system forcibly terminates Pylama then. Coala does not accumulate errors and

Table 1: Comparison of Coala and Pylama

	Coala	Pylama
Popularity among github users	+	-
CPU usage	+	+
RAM usage	+	-
Parallel processing	+	-
Analysis time (sec)	112	183

flushes information on errors to stdout periodically, so the RAM does not exhaust.

Combined comparison results are presented in Table 1. Based on this results we decided to use Coala as a base framework for future work. Another possible option is to patch Pylama.

Next planned step is to process results of analysis by a parsing module and to standardize them. We are going to store standardized analysis results to a database. We plan to evaluate a number of relational (e.g. PostgreSQL) and non-relational databases (e.g. HBase and Cassandra).

We designed a data model and represented it as an ER diagram (Fig. 3).

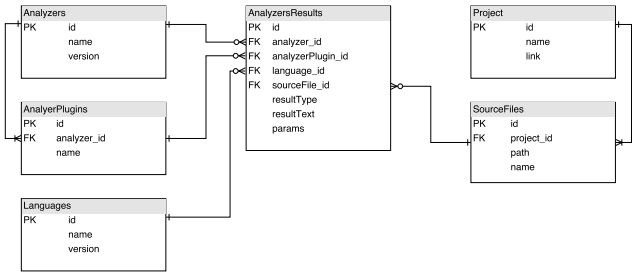

Figure 3: The database schema for storing source code analysis results

6 FUTURE WORK

We defined an overall structure of the system so the next step is to build a working prototype suitable for massive code analysis. Also we are going to define a more strict set of metrics and to start visualizing them. Some of methods of visualizing results of measuring quality of program products are described in [11], [10], [14].

Analysis results are going to be represented as numerical metrics stored in tables. Next step is to create a separate module to interpret these raw standartizes results. The last step is to visualize results by a request of end user initiated code analysis. Thus, different types of comparisons can be done on visualization step [16]. For example, comparison can be based on chronological characteristics of objects or we can perform per component comparison.

Resulting infographics can be presented in various forms, e.g. matrixes, maps, figures, graphs and diagrams.

7 CONCLUSION

A problem of controlling, measuring and predicting code quality is actual at the moment and will become even more actual in the future. Software developers working on big projects need measurable code quality-related metrics to improve their process. End users and other third parties need these metrics to choose a product of best possible quality and for various other needs. Solving this problem requires a proper instrument which scales well and produces well-defined and predictable results. Authors of this paper proposed an approach of creating this instrument, described its architecture and chose a set of tools as a base to implement it.

REFERENCES

[1] N. Antunes and M. Vieira. 2009. Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services. In *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*. 301–306. https://doi.org/10.1109/PRDC.2009.54

[2] D. Baca, K. Petersen, B. Carlsson, and L. Lundberg. 2009. Static Code Analysis to Detect Software Security Vulnerabilities - Does Experience Matter?. In *2009 International Conference on Availability, Reliability and Security*. 804–810. https://doi.org/10.1109/ARES.2009.163

[3] S.K. Chernonozhkin. 1997. *Methods and tools of metric support for the development of quality programs.* NGTU Publishing, Novosibirsk.

[4] Coala. [n. d.]. Coala - Linting and fixing code for all languages. ([n. d.]). Retrieved November 3, 2017 from http://coala.io

[5] T. Sorgente E.B. Fernandez, M.M. Larrondo-Petrie and M. VanHilst. 2007. A methodology to develop secure systems using patterns. (2007), 107–126.

[6] Flake8. [n. d.]. Flake8 - your tool for style guide enforcement. ([n. d.]). Retrieved November 3, 2017 from http://flake8.pycqa.org

[7] GitHub. [n. d.]. GitHub: Social coding. ([n. d.]). Retrieved November 3, 2017 from https://github.com

[8] Analia Irigoyen Ferreiro Ferreira, Gleison Santos, Roberta Cerqueira, Mariano Montoni, Ahilton Barreto, Andrea O. Soares Barreto, and Ana Regina Rocha. 2007. Applying ISO 9001: 2000, MPS.BR and CMMI to Achieve Software Process Maturity: BL Informatica's Pathway. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 642–651. https://doi.org/10.1109/ICSE.2007.15

[9] G.I. Kaigorodtsev. 2011. *Introduction to measurement theory and metrology of programs.* NGTU Publishing, Novosibirsk.

[10] M. Lanza and S. Ducasse. 2003. Polymetric views - a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering* 29, 9 (Sept 2003), 782–795. https://doi.org/10.1109/TSE.2003.1232284

[11] Michele Lanza and Radu Marinescu. 2006. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-39538-5

[12] Panagiotis Louridas. 2006. Static Code Analysis. *IEEE Softw.* 23, 4 (July 2006), 58–61. https://doi.org/10.1109/MS.2006.114

[13] T. Menzies, J. Greenwald, and A. Frank. 2007. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering* 33, 1 (Jan 2007), 2–13. https://doi.org/10.1109/TSE.2007.256941

[14] Lanza Michele and StÃŀphane Ducasse. 2002. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of LMO 2002 (Langages et ModÃĺles Ãă Objets*. 135–149.

[15] A. Moser, C. Kruegel, and E. Kirda. 2007. Limits of Static Analysis for Malware Detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 421–430. https://doi.org/10.1109/ACSAC.2007.21

[16] K.V. Nefedyeva. 2013. Infographics and data visualization in analytics. In *Proceedings of Saint-Petersburg State University of Culture and Arts*, Vol. 2. 89–93.

[17] Pylama. [n. d.]. Pylama - code audit tool for Python and JavaScript. ([n. d.]). Retrieved November 3, 2017 from https://pylama.readthedocs.io

[18] A. Shabtai, Y. Fledel, and Y. Elovici. 2010. Automated Static Code Analysis for Classifying Android Applications Using Machine Learning. In *2010 International Conference on Computational Intelligence and Security*. 329–333. https://doi.org/10.1109/CIS.2010.77

[19] Misha Zitser, Richard Lippmann, and Tim Leek. 2004. Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code. *SIGSOFT Softw. Eng. Notes* 29, 6 (Oct. 2004), 97–106. https://doi.org/10.1145/1041685.1029911

[20] S.V. Zwezdin. 2010. Problems of measuring program code quality. *SUSU Vestnik* 2 (2010), 62–66.