Identification of strategies over tools for static code analysis

Identification of strategies over tools for static code analysis

Darko Stefanović[1], Danilo Nikolić[2], Sara Havzi[3], Teodora Lolić[4], Dušanka Dakić[5]

[1,2,3,4,5] Department of Industrial Engineering and Engineering Management, Faculty of Techincal Sciences, University of Novi Sad, Novi Sad, 21000, Serbia

E-mail: nikolic.danilo@uns.ac.rs

Abstract. Static code analysis tools are being increasingly used to improve code quality. The source code's quality is a key factor in any software product and requires constant inspection and supervision. Static code analysis is a valid way to infer the behavior of a program without executing it. Many tools allow static analysis in different frameworks, different programming languages, and detecting different defects in the source code. Different strategies of using static code analysis tools are often used, and these strategies are not classified. In this paper, an experiment was conducted on different tools and their use in relation to the standard code review cycle. The identified strategies for using static code analysis tools and the steps required to implement them are presented. When using the tool, users should choose one of the identified strategies to implement following the defined steps for successful implementation.

1. Introduction

Since the beginning of software development as a scientific discipline, software developers have lacked a source code quality assessment method [1].

The source code's quality is a key factor in any software product and requires constant verification and monitoring. Static analysis is used to maintain and improve the quality of the source code. Static analysis of program code has been used since the early 1960s to optimize the operation of compilers [2]. Later, it proved useful for debugging tools, as well as for software development frameworks. A growing number of tools allow static code analysis, many of which allow an analysis of different programming languages [3].

Static analysis tools are used to generate reports and point out certain deviations from the prescribed code quality standards. However, these tools do not allow automatic modifications to the source code. The decision to change the structure of previously written code is up to the software developers. Static code analysis tool helps software developers by, in addition to generating a report, stating the cause of a particular defect, as well as how this defect can be corrected.

As the first step in this research, a systematic literature review in the field of static code analysis was conducted [4], and various tools for static code analysis, programming languages for which they have support, and the types of defects they detect were presented and analyzed. This systematic review of the literature is presented in detail in Section 3 of this paper.

Static code analysis tools are applied directly to the source code of the program [1]. Software developers and other users of these tools use different ways to apply these tools over program source code. This paper aims to identify different ways (strategies) of applying tools for static code analysis over program source code. Furthermore, the paper aims to identify the standard steps of implementing different strategies over static code analysis tools.

The remainder of the paper is organized as follows. Section 2 presents the basic concepts of static code analysis tools; Section 3 describes related work based on a conducted systematic literature review; Section 4 describes the methodology used to conduct the research; Section 5 presents the results, strategies, and implementation steps defined strategies; Section 6 concludes the paper and suggests future research.

2. Theoretical foundations

This section will present how static code analysis is performed, which tools help with this process, and an introduction to domain-specific programming language (DSL) and general-purpose programming language (GPL).

2.1. Static code analysis

The process of static code analysis is useful not only for optimizing the compiler's operation (which was the original purpose) but also for detecting defects and possible shortcomings. In this way, it is possible to create tools that will help developers understand the program's behavior and identify various shortcomings of the program without its execution. The tools used for static code analysis are programs that explain the behavior of other programs [2].

At the outset, it is necessary to consider the types of defects that static analysis can detect. Problems can also arise due to a lack of understanding of what secure programming entails. It is not uncommon for developers to be unaware of how attackers will try to exploit a piece of code.

With this in mind, static analysis is suitable for identifying problems for several reasons [5]:

Static analysis tools apply checks thoroughly and consistently, without any prejudice that the programmer may have about which parts of the code are "interesting" from a security perspective or which parts of the code can easily be performed by dynamic testing. The code review must be done without the subjective feeling of the programmer. Therefore, unbiased analysis can be valuable, which static code analysis tools provide.

By examining the code itself, static analysis tools can often point to the root cause of a security problem, not just one of its symptoms. This is especially important to ensure that defects are corrected. Static analysis can find defects in early development, even before the program is run for the first time. Finding a defect early reduces the cost of fixing the defect, but a cycle of quick feedback can help guide the programmer's work: The programmer has the opportunity to correct defects that he was not previously aware could even occur. Attack scenarios and code construct information used by the static analysis tool act to transmit knowledge.

When analysis detects a new type of attack, static analysis tools make it easy to recheck a large portion of the code to detect where a new attack might occur. Some security bugs have been in the software for years before they were discovered, making the assessment of outdated code of newly discovered types of defect invaluable.

However, despite all of the above, static code analysis tools are not used often. Software developers do not often decide to improve their source code's quality using these tools [1].

The standard code review cycle includes four main phases [5]:

1. defining the goal,
2. launching a tool for static code analysis,
3. review of the source code (based on the obtained result of static analysis),
4. source code reengineering.

In addition to the standard cycle, Figure 1 shows several potential feedbacks or minor iterations between standard cycle steps, making the cycle more complex than the standard one. The number of iterations of this cycle and the need for feedback within the cycle depends on applying the static code analysis report. This paper aims to identify strategies for applying static source analysis tools based on feedback within the standard cycle and iteration between steps. These strategies are described in Section 5 of this paper.

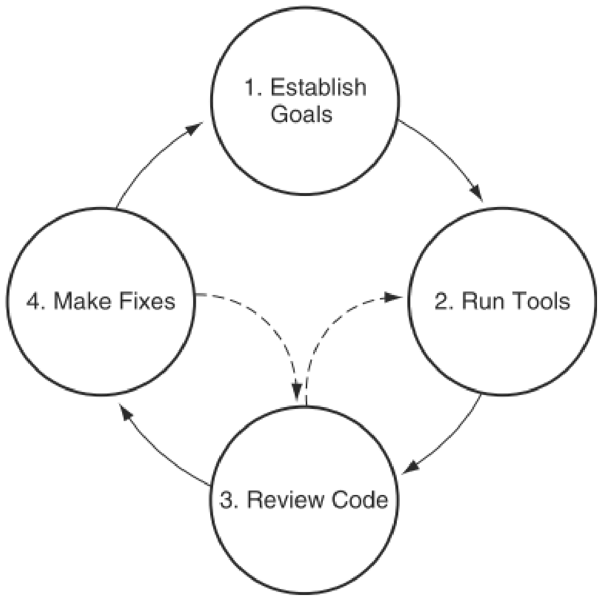

Figure 1. The code review cycle [5].

2.2. GPL and DSL
The GPL is a computer language that is widely applicable in application domains and does not have a special function for a specific domain. Static analysis tools and quality control tools, in general, are ready for GPL source code analysis.

In contrast, DSL specializes in a specific application domain. The difference is not always clear because the programming language may have specialized characteristics for a particular domain but be applicable more widely or capable of wide application, but in practice is used primarily for a particular domain [6]. Professionals in many different fields can develop and describe solutions to problems that take the form of source code artifacts using DSLs that are specifically defined for their field [7]. Static code analysis is a big challenge when it comes to DSL.

The experiment described in this paper refers to a single GPL, and the strategies identified relate to the GPL. Additional analyzes are needed to conclude the implementation of strategies over DSL.

3. Related work
Systematic literature review in the field of static analysis of code and its tools was conducted. A paper describing the process of the systematic literature review was published in [4]. The studies included in this review are published in the previous ten years. The paper review was performed based on guidelines for a systematic review of software engineering literature by Barbara Kitchenham [8], focusing on presenting the most commonly used tools divided by programming languages for which they have provided support and detecting defects.

To review the literature, two databases were searched:
1. SCOPUS and
2. Google Scholar.

The search terms for the mentioned databases are as follows: "Static code analysis" AND "tools" AND "detection" and pubyear> = 2010. The search resulted in 346 initial papers.

Based on the set inclusion criteria, by reviewing abstracts and titles, 22 primary studies were included in the literature review. Some of the representative examples of the included studies are [9], [10], [11],

[12], [13]. A detailed procedure of this systematic review of the literature step by step is presented in the research [4].

The results obtained show that many more papers represent results on tools that have secured support for the GPL than papers representing tools that have secured DSL support. The percentage is shown in Table 1.

Table 1. Percentage of primary studies by type of programming language [4].

Programming language	Number of studies	%
DSL	3	14%
GPL	19	86%

As part of this review, the tools for static code analysis that are most often used in research on different programming languages and detecting various defects were also analyzed.

Many of the static code analysis tools presented in different studies support several GPLs, while other tools have support for only one programming language. Most static code analysis tools support those programming languages that are most commonly used in the industry.

This systematic review presents the static code analysis tools most commonly used in research and which tools are analyzed based on the type of defects they detect and the programming languages they support.

Six tools were presented, which were most often analyzed and presented in primary studies. These are tools that have been presented three or more times in different studies. In comparison, other tools presented in the studies covered by this review were analyzed in two or one studies. Graphically presented results are shown in Figure 2.

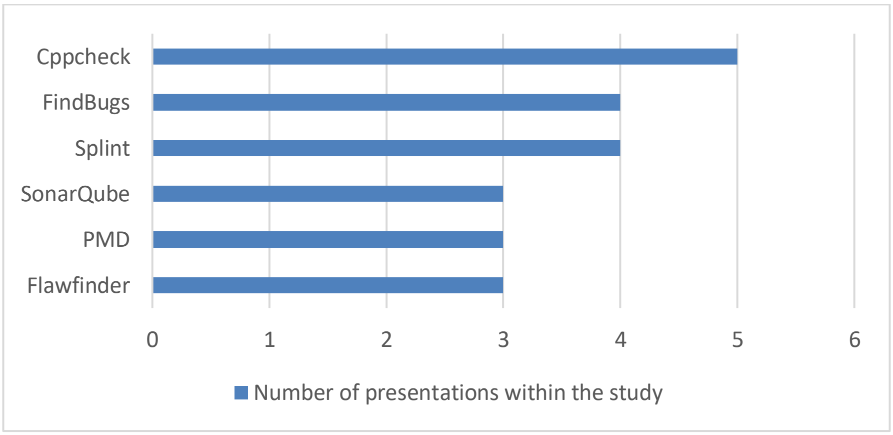

Figure 2. The most common static code analysis tools presented in primary studies [4].

Static code analysis tools are increasingly finding their application and use in the work of programmers in various positions, so it is important that these tools are constantly improved, but also to develop new tools that will offer their customers a better service by building confidence in the results of static analysis and allowing them to avoid certain defects in the source code and improve their software solution.

However, static code analysis tools only allow the analysis of a certain number of programming languages. The range of programming languages for which they have support is constantly expanding. In comparison, it is expected that work will be done on creating and improving tools that will have support for DSL in the coming period. For now, only a few tools have DSL support with a specification of the rules built into the tool using the plugin.

The results show that certain tools are used more often in studies than others. Based on the results obtained from research [4], it is concluded that research mainly analyzes the results obtained using a certain tool for static code analysis, under certain conditions, for certain types of defects, and over certain programming languages.

No paper attempts or proposes specific strategies for conducting static code analysis using some of the presented tools.

4. Methodology

To successfully set up the research methodology, it is first necessary to identify the tools of static code analysis that will be used for the realization of the research. An experiment conducted on tools was used as a research model to identify strategies for conducting static code analysis. In this regard, the tasks of this research are:

1. identify the tools that will be used to conduct the experiment,
2. identify strategies for conducting static code analysis.

The tools used to conduct this research have been identified based on a literature review conducted within [4], already described in Section 2 of this paper. The literature review identified the most commonly used tools for static code analysis in the primary studies covered by this literature review. Also, the selected tools must have java programming language support and a plug-in for the Eclipse framework.

Once the tools that will be used to conduct the experiment have been identified. An analysis was conducted on the selected tools, their characteristics and the features it offers. The experiment was conducted using selected tools on different software products and analysis of the implementation in relation to the standard cycle of code review.

As the first phase of conducting the experiment, it is necessary to establish the initial configuration of the selected tools on the device from which the experiment was conducted. Next, it is necessary to analyze the features of each tool and the ways in which it is possible to apply the tool over the source code of the software product. After the analysis, it is necessary to conduct an experiment, which means conducting a code review procedure on each selected tool and identifying different strategies. Above the identified strategies, it is necessary to detect standard steps for their implementation. The experiment is considered complete when all ways to perform static code analysis in relation to the standard code review cycle are detected and described.

A description of the strategy identification process is described in Section 5 of this paper.

4.1. Research limitations

In the research, the authors limit themselves to the possibility of additional strategies that can be identified in the process of conducting static analysis over other tools not covered by this research or over tools that have provided support for other programming languages not used in this research. The experiment was performed on a single GPL, and it is not possible to draw conclusions about DSL based on this experiment. Also, the experiment was conducted on one project, and there is a possibility of identifying other strategies on different types of projects that are not the subject of this research.

5. Results

This section describes the selected tools, identified strategies, sub-strategies of the static code analysis tools, and the steps of implementing each strategy concerning the standard code review cycle.

The selected tools that met the set requirements for conducting the experiment are::

1. SonarQube,
2. FindBugs,
3. PWD,
4. Checkstyle.

5.1. Strategies over static code analysis tools

The basic division into two major classes of strategies for using static code analysis tools is [14]:

1. Running a static analysis tool over a completed software project – **S1**.
2. Running a static analysis tool during software project development – **S2**.

In this paper, a completed software product will be considered a software product in which development has been completed and which should be tested before deployment. While the application during development is considered to be the use of tools for static analysis during development, ie the inclusion of tools from the beginning of software product development.

If a static analysis startup strategy is applied during software product development, it is essential that the static code analysis tool be included in the integrated development environment (IDE) being developed by the program or external startup at specified time intervals (S2). In this way, the development environment, with the help of a plugin, independently generates static code analysis reports, after which the software developer can check the detected defects and correct them before continuing with further development. Many static code analysis tools provide the ability to integrate into integrated development environments. However, static code analysis is often used only after development is complete, and it is necessary to apply a strategy to run analysis over the completed software product (S1).

Strategies for using static analysis tools will be described in detail, and their sub-strategies and steps within these strategies.

5.1.1 Running a static analysis tool over a completed software project – S1

For this research, a static code analysis was performed on a completed project done in an Eclipse development environment. The source code of the project is written in the Java programming language. The project is a backend part of a multi-layered application for monitoring students at faculties, divided by departments. Over the completed software project, a static analysis of the code was started using tools, and a report was generated which marked the defects in the project. Static code analysis tools independently classify defects that they detect according to the adopted classification scheme.

The tool's initial report detected a total of 18 defects at different levels in the project. After correcting all defects detected by the tool within the initial report, restarting the tool is expected to result in a total of 0 defects. However, after reengineering the source code and resolving all defects, restarting the tool over the source code generates a report that detects 8 new defects in the project's source code.

Based on the obtained report, it can be concluded that the source code corrections caused new defects marked within the new report, generated by restarting the tool on the same project.

After correcting newly created defects, another run of the tool over the source code, the number of all defects is equal to 0.

By conducting this experiment, which involves correcting all defects observed in the initial report and then restarting the tool to check with the second report, it is possible to identify the first subtype of the strategy, listed in section 5.1. of this paper, which is entitled "Starting the static analysis tool after correcting all defects "– S1.1.

If the identified strategy is compared with the presented standard code review cycle based on static analysis, described in section 2.1. it would be seen that the identified strategy fully follows the standard procedure without frequent feedback or iterations between standard steps within the cycle.

Finally, the standard steps for implementing this sub-strategy, relative to the standard code review cycle, are:

1. defining the goal,
2. launching a tool for static code analysis,
3. review of source code,
4. source code reengineering,
5. launching a tool for static code analysis,
6. review of source code,

7. source code reengineering.

However, if the possibility of more frequent use of feedback and smaller iterations within the standard procedure is set, another sub-strategy can be identified.

If the software developer starts correcting defects in the program's source code and, after eliminating one defect, restarts the static analysis over the source code, the report is expected to identify one error less than the initial report.

In this way, the software developer, every time he corrects one of the detected defects, checks whether that defect has been eliminated. On the other hand, there is a possibility that the correction of one defect will cause additional defects in the source code of the project. Frequent starting of the tool, i.e. starting after each correction, enables earlier detection of newly formed defects. In this way, even if new defects occur, the software developer can react and eliminate new defects promptly or otherwise correct the source code not to cause new defects.

Setting up this way of applying static analysis, by running the tool after correcting each defect, another subtype of the strategy listed in section 5.1 of this paper is identified, entitled "Running the static analysis tool after correcting any defect" – S1.2.

If the identified strategy is compared with the presented standard cycle of code review based on static analysis, described in section 2.1., It can be concluded that this sub-strategy requires more feedback, i.e. more iterations between standard steps within the cycle.

Finally, the steps of the second sub-strategy are:

1. defining the goal,
2. launching a static analysis tool,
3. review of the source code (based on the obtained result of static analysis),
4. correction of a defect.

Steps two, three, and four are repeated as many times as there are total defects in the project.

5.1.2 Running a static analysis tool during software project development – S2

If a different static analysis strategy is applied to the static code analysis process during software development, in this case, two sub-strategies can be separated. These sub-strategies differ depending on whether the static code analysis tool is included in the development environment in which the software solution is being developed or is used externally during development.

The first sub-strategy, "Starting static analysis tools by external tool calling during software project "– S2.1 involves the external use of static code analysis tools and the occasional running of tools over a project during development. Occasional running represents running the tool at different levels of the scope of written code, there are a number of levels such as: running after a single class is written, a specific block of code, a logical part, or after each line of code in a project during development.

It is also possible to start based on a time determinant. In defining this sub-strategy and its steps, there will be no more detailed sub-strategies depending on the written code's volume levels.

The steps of this sub-strategy concerning the standard code review cycle are:

1. defining the goal,
2. development of a part of the software project,
3. launching a static analysis tool,
4. review of the source code (based on the report of the obtained result of static analysis),
5. correction of detected defects,
6. continued development of the software project.

Steps two, three, four, five, and six are repeated until the software solution development is complete.

The second sub-strategy, "Running static analysis tools by including tools in the development environment in which the software project is developed," – S2.2 involves the integrated use of static code analysis tools. Integrated use is possible only for those static analysis tools, which have provided support for the appropriate environment, ie, it is possible to place them as an adjunct in the development environment in which the project is developed. In this way, during the development of the project, a tool for static analysis is launched, which, while the project is being developed, performs analysis on the source code and detects the observed defects.

The advantage of this method is that the software developer finds out the cause that caused the defect and eliminates it at the moment of the defect's occurrence.

The steps of this sub-strategy concerning the standard code review cycle are:

1. defining the goal,
2. inclusion of tools for static analysis in the development environment of the project,
3. software project development,
4. the static analysis tool detects a defect,
5. review of the source code (based on the detected defect),
6. correction of detected defect,
7. continued development of the software project.

Steps three, four, five, six, and seven are repeated until the software solution's development is completed.

6. Conclusion

Finally, strategies for applying static analysis tools can be classified as presented in table 2.

Table 2. Identified strategies over tools for static code analysis

Strategy label	Strategy	Substrategy label	Substrategy
S1	Running a tool for static analysis over a completed software project	S1.1	Starting the static analysis tool after correcting all defects
		S1.2	Running the static analysis tool after correcting any defect
S2	Running a static analysis tool during software project development	S2.1	Starting static analysis tools by external tool calling during software project development
		S2.2	Running static analysis tools by including tools in the development environment in which the software project is developed

The strategies identified by this research relate to the type of project used to conduct the experiment, and the tools used in the experiment. The authors limit themselves to the possibility of identifying other strategies over other tools and projects.

Further research on identified strategies should include a comparative analysis of strategies across different tools and projects. It is necessary to conduct different experiments by applying the identified strategies over different static code analysis tools. During the analysis, it is necessary to measure the speed of strategy implementation to compare the strategies concerning the time required for their implementation. It is also necessary to compare the "accuracy" of the strategy, i.e., how many erroneous results the tool generates using the identified strategies.

The development and application of tools for static code analysis enable better software solutions and represent a step forward in software development.

References

[1] E. Penttila, "Improving C++ Software Quality with Static Code Analysis," Aalto, Aalto University, 2014.
[2] A. Moller and I. Schwartzbach, "Static program analysis," 2019.
[3] M. Beller, R. Bholanath and M. S., "Analyzing the state of static analysis: A large-scale evaluation in open source software," 2016.
[4] D. Stefanović, D. Nikolić, D. Dakić, I. Spasojević and S. Ristić, "Static code analysis tools: A systematic Literature Review," in Proceeding of of 31st DAAAM International Symposium, Vienna, 2020.
[5] J. West and B. Chess, "Secure programming with static code analysis," Pearson education, 2007.
[6] M. Fowler, "Domain-Specific Languages," in Pearson Education, London, 2010.
[7] I. Ruiz-Rube, T. Person, J. M. Dodero, J. M. Mota and J. M. Sánchez-Jara, "Applying static code analysis for domain-specific languages," Software & Systems Modeling, vol. 19, no. 1, pp. 95-110, 2020.
[8] B. Kitchenham, "Procedure for Undertaking Systematic Reviews," Keele University, Computer Science Department, 2004.
[9] A. Kaur and R. Nayyar, "A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code," Procedia Computer Science, vol. 171, no. 2019, pp. 2023-2029, 2020.
[10] D. Marcilio, C. Furia, R. Bonifacio and P. Gustavo, "SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings," The Journal of Systems & Software, vol. 168, 2020.
[11] P. Nunes, I. Medeiros, J. Fonseca, N. Neves, M. Correia and M. Vieira, "An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios," Computing, vol. 101, no. 2, pp. 161-185, 2019.
[12] N. Meghanathan, "Identification and removal of software security vulnerabilities using source code analysis: a case study on a java file writer program with password validation features," Journal of Software, vol. 8, no. 10, 2013.
[13] K. Goseva-Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," Information and Software Technology, pp. 18-33, 2015.
[14] J. Xie, H. R. Lipford and B. Chu, "Evaluating interactive support for secure programming," Conference on Human Factors in Computing Systems, pp. 2707-2716, 2012.