DOI: 10.2507/31st.daaam.proceedings.xxx

STATIC CODE ANALYSIS TOOLS: A SYSTEMATIC LITERATURE REVIEW

Darko Stefanović, Danilo Nikolić, Dušanka Dakić, Ivana Spasojević, Sonja Ristić

This Publication has to be referred as: Stefanović, D[arko], Nikolić, D[anilo], Dakić, D[usanka], Spasojević, I[vana], Ristić, S[onja]; (2020). Static code analysis tools: a systematic literature review, Proceedings of the 31st DAAAM International Symposium, pp.xxxx-xxxx, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-xx-x, ISSN 1726-9679, Vienna, Austria
DOI: 10.2507/31st.daaam.proceedings.xxx

Abstract

Static code analysis tools are being increasingly used to improve code quality. Such tools can statically analyze the code to find bugs, security vulnerabilities, security spots, duplications, and code smell. The quality of the source code 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, for different programming languages, and for detecting different defects in the source code. Still, a small number of tools provide support for domain-specific languages. This paper aims to present a systematic literature review focusing on the most frequently used static code analysis tools and on classifying the presented tools according to the supported both general-purpose and domain-specific programming languages and the types of defects a specific tool can detect.

Keywords: static code analysis; tools; defects; reengineering; literature review;

1. Introduction

At the Faculty of Technical Sciences, various forms of application of information technologies are studied, such as [1] [2] [3] [4] [5], in addition, information technologies find application in areas such as [6] [7], in this paper, the application of information technologies and tools to static code analysis will be described.

Since the beginning of software development as a scientific discipline, software developers have lacked a method for source code quality assessment. Most software developers rely on the compiler and the fact that the software product works successfully until an error is caused by poor code quality or a bug that disrupts the operation of the software solution occurs.

The quality of the source code 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 [8]. Later, it proved useful for debugging tools, as well as for software development frameworks. There is a growing number of tools that allow static code analysis, many of which are open-source tools and allow an analysis of several different programming languages [9].

Static analysis tools are used to generate reports and to 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 way the previously written code is structured remains in the hands of 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.

However, the question remains on how to approach the defects, to eliminate the highlighted shortcomings, and reengineer the source code.

When it comes to static code analysis tools, a big problem is a fact that there are a large number of tools that provide this type of analysis. The number of tools is constantly increasing, and it is necessary to classify existing tools according to the programming language for which they have the support and the types of defects they detect.

Static code analysis is performed on the source code of the program and therefore the programming languages in which the program is written are very important. From the side of static code analysis, it is necessary to separate the types of programming languages into general-purpose programming languages (GPL) and domain-specific programming languages (DSL). In this literature review, there is a classification of tools that provide support for these two groups of programming languages.

Studies that are included in the literature review are set for a certain period of time. This paper reviews published studies in the previous ten years, conducted based on the guidelines for performing systematic literature reviews in software engineering by Barbara Kitchenham [10], with a focus on displaying the most commonly used tools divided by the programming languages for which they have support and defects they detect.

The remainder of the paper is organized as follows. Section 2 presents the basic concepts of static code analysis tools. Section 3 describes the systematic literature review methodology, presents conducted planning the review phase of the systematic literature review, and presents conducting the review phase of the systematic literature review, where primary studies were selected and the review results presented. Section 4 discusses the systematic literature review results in detail, and Section 5 concludes the paper and suggests future research.

This literature review will be used as a basis for conducting further experiments on static code analysis tools.

2. Theoretical foundations

This section will present what manual code review is, how it is performed, how static code analysis and its tools help this process, and what are domain-specific and general-purpose programming languages.

2.1. Manual code review

Code review is used to improve the maintenance process of a software product, increasing its reliability and security. Code review activity can reduce the number of required updates in the source code, and can be easily integrated into the software development lifecycle so that more experienced software developers do each code review. However, a manual review of the source code takes a very long time [11]. To conduct an efficient manual code review, reviewers must know all possible defects and inaccuracies before carefully checking all source code. As a substitute for a lengthy manual source code review, the use of a static code analysis tool is recommended. These tools are faster and contain the knowledge to perform the scan. However, these tools cannot completely replace the manual review, but they facilitate this process.

2.2. Static code analysis

The static code analysis process is useful not only for optimizing the operation of the compiler (which was the original purpose) but also for detecting irregularities and possible defects. In this way, it is possible to create tools that will help developers to understand the behaviour of a program and to identify various defects in a program without its execution. The tools used for static code analysis are programs that explain the behaviour of other programs [8].

Static code analysis is significantly faster than conventional testing and can detect any defect visible in the program's source code. If the tools for static code analysis are compared with the manual code review by software developers, it can be concluded that the code review using the tool is also much faster and more efficient [11]. However, for static code analysis to detect any defect, it must be visible in the source code.

In addition to the amount of context required to identify a defect, many defects can only be found in a particular representation of the program. Figure 1 shows a matrix formed according to the type and visibility of the defect. High-level problems are often only visible in program design, while implementation errors can often only be spotted by examining program source code. Object-oriented languages such as Java have a large number of libraries, which makes it easier to understand the design by examining the source code.

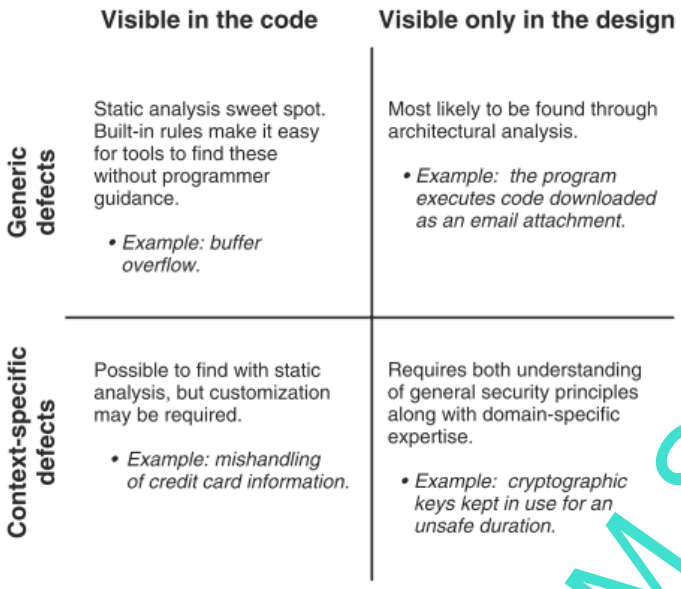

Fig. 1. Generic defects and context-specific defects [11]

Static code analysis tools look for a specific set of patterns or rules in the source code, very similar to the way antivirus programs detect viruses. More advanced tools allow practitioners to add new rules to a set of predefined ones. These tools cannot find an error if patterns or rules do not specify such behaviour. The most significant value of a static code analysis tool is the ability to identify common programming defects.

The standard code review cycle includes four main phases [11]:

1. establish goals,
2. run the static analysis tool,
3. review code (using the output of the tool),
4. make fixes.

In addition to the standard cycle, Figure 2 shows several potential feedbacks or minor iterations between standard cycle steps, making the cycle more complex than the standard procedure.

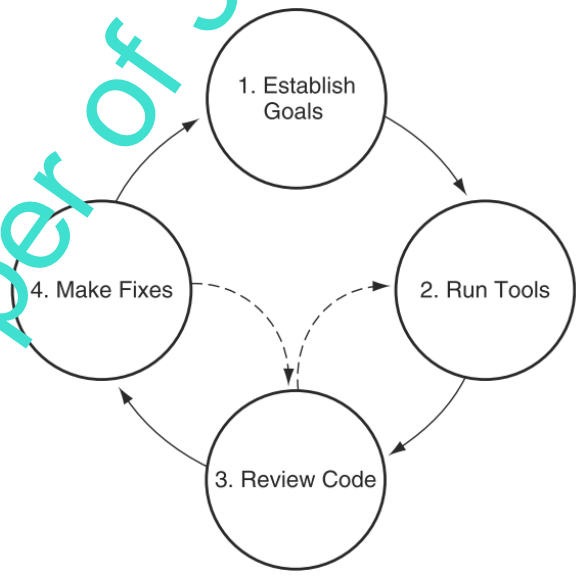

Fig. 2. Code review cycle [11]

2.3. General-purpose (GPL) and domain-specific programming languages (DSL)

A GPL is a computer language that is broadly applicable across application domains and lacks specialized features for a particular domain. Static analysis tools and quality inspection tools, in general, are prepared to analyse the source code of GPLs.

Contrary, a DSL is specialized in a particular application domain. The line is not always sharp, as a language may have specialized features for a particular domain but be applicable more broadly, or conversely, may be capable of broad application but in practice is used primarily for a specific domain [12]. Professionals in many different areas can develop and describe problem solutions that take the shape of source code artefacts using DSLs that are specially defined for their field [13].

Static code analysis is a great challenge when it comes to DSLs, so this paper will highlight the tools that can contribute to static analysis in this group of programming languages.

2.4. A Taxonomy of Software Security Errors

In order to successfully classify all defects presented in the studies included in this literature review, it is necessary to introduce a unique taxonomy according to which all presented defects will be classified.

A simple, intuitive taxonomy of software security is proposed in [14], called Seven Pernicious Kingdoms. This classification scheme is amenable to automatic identification using static analysis tools for detecting real-world security vulnerabilities in software. In this way, this taxonomy encompasses all security defects that static code analysis can detect and for this reason, this taxonomy is used in research.

The classification divides the defects into seven (plus one) high-level kingdoms, seven of these kingdoms are dedicated to errors in source code, and one is related to configuration and environment issues. It is presented in order of importance to software security [14]:

1. input validation and representation,
2. API abuse,
3. security features,
4. time and state,
5. errors,
6. code quality,
7. encapsulation,
8. environment.

All studies included in this literature review use different taxonomies to classify the defects that the presented tools detect. For this reason, a uniform taxonomy is used in this paper and all defects were classified according to this presented taxonomy.

3. Methodology

To conduct this systematic literature review, a procedure for systematic reviews developed by Barbara Kitchenham [10] is followed. According to Kitchenham, a systematic literature review can be summarized into three main phases: Planning the Review, Conducting the Review, and Reporting the Review.

3.1. Planning the Review

The first activity of Planning the Review phase should be an elaboration of the need for a systematic literature review, by reviewing existing literature review of the subject [10]. However, there are no explicit, systematic literature reviews on the
subject of static code analysis tools. There is only threat analysis in software systems in general [15]. Other sources of information about static code analysis tools could be the related work chapters of studies that are presenting a novelty on the subject. However, these studies only mention previous research in the field of static code analysis relevant to their particular topic of interest. A systematic review of the literature presents the static code analysis tools most commonly used in studies and the supported programming languages. Studies describing a static code analysis tool or tools and analysing their defect detection in one or more programming languages have been searched and included in the literature review.

Based on the guidelines from Kitchenham [10], the following research questions are formulated:

RQ1. Are there static code analysis tools applicable to any GPL?
RQ2. Is there a static code analysis tool that applies to a DSL?
RQ3. Which tools are most commonly mentioned in the primary studies?
RQ4. Is there a tool that detects all types of defects?

For this literature review, the following databases where searched:

1. SCOPUS and

2. Google scholar.

Search terms defined for search in these databases is: *"Static code analysis" and "tools" and "detection"* and pubyear>=2010. The search resulted in 346 papers.

Inclusion criteria applied to papers are:

IC1: Papers have to be written in English.
IC2: Paper has to present tools based on static code analysis.
IC3: Paper has to present supported programming languages by static code analysis tool(s).

Exclusion criteria applied to papers are:

EC1: Duplicate papers should be removed.
EC2: If one author had more than one paper regarding the same tool, only one paper should be included in the review.

For this literature review, the data extraction strategy is developed. For each primary study, the following features will be extracted in order to answer the research questions:

1. publication year and source type,
2. proposed tool(s),
3. type(s) of defect(s) they detect,
4. supported programming languages (DSL/GPL).

3.2. Conducting the Review

After applying defined inclusion and exclusion criteria on available papers, papers were critically appraised based on their relevancy and type of information they contained. Finally, there were 22 primary studies available for data extraction.

The flow diagram of the systematic literature review process is presented in Fig. 3.

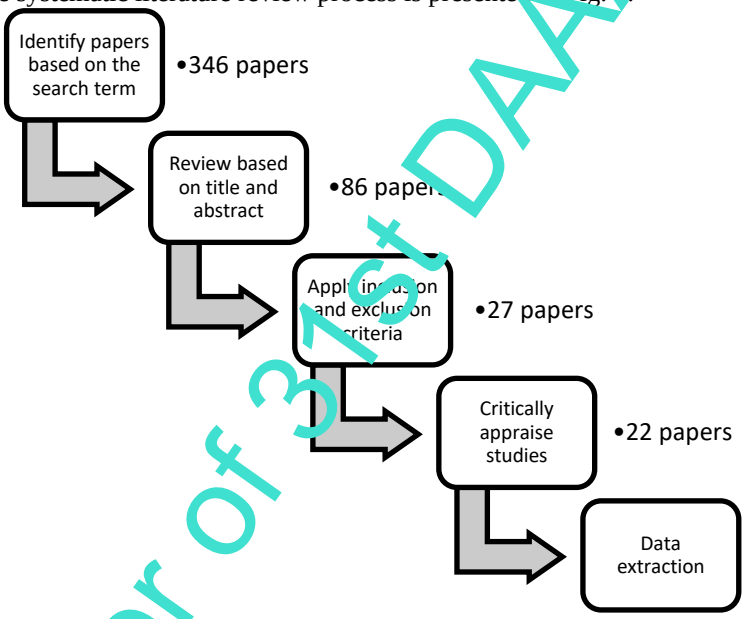

Fig. 3. Systematic literature review flow diagram

Search through data sources with previously defined search terms resulted in the identification of 346 primary studies. Based on the title and abstract, 260 papers were excluded from further research. A review based on inclusion and exclusion criteria is conducted on the paper and resulted in the inclusion of 27 primary studies. The review based on a critical assessment of the study resulted in the exclusion of 5 studies from further research. Finally, 22 primary studies are included in the literature review.

3.3. Data extraction and summarization

In this chapter, the main characteristics of primary studies are summarized and presented, following with the tables that present extracted data. Interpretation and discussion of the presented results will be described in the next chapter.

Fig. 4 presents primary studies that present static code analysis tools by year of publication. The line diagram shows that two peaks in the number of published studies occurred, one in 2012 and one in 2016.

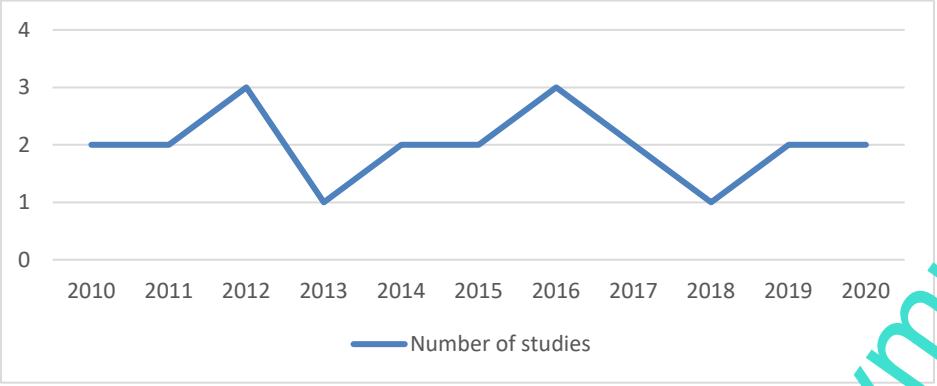

Fig. 4. Summary of studies that present static code analysis tool(s) by year

Table 1. presents source types of primary studies, showing that 37% of the papers were published as journal articles, 59% of conference proceedings, and one master thesis (4%).

Source type	Studies	%
Journal article	[16], [17], [18], [19], [20], [21], [22], [13]	37%
Conference proceedings	[23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35]	59%
Master thesis	[36]	4%

Table 1. Source types of primary studies

In the review protocol, three research questions were formulated. Figure 5 presents extracted data that answers the first research question-RQ1: Are there static code analysis tools applicable to any GPL?

Some of the presented static code analysis tools have support for several GPLs, while other tools have support for only one programming language. Most static code analysis tools have support for those programming languages that are most commonly used in the industry. The reason for this is that in this way, the creators of the static code analysis tool provide a higher number of users of their tool.

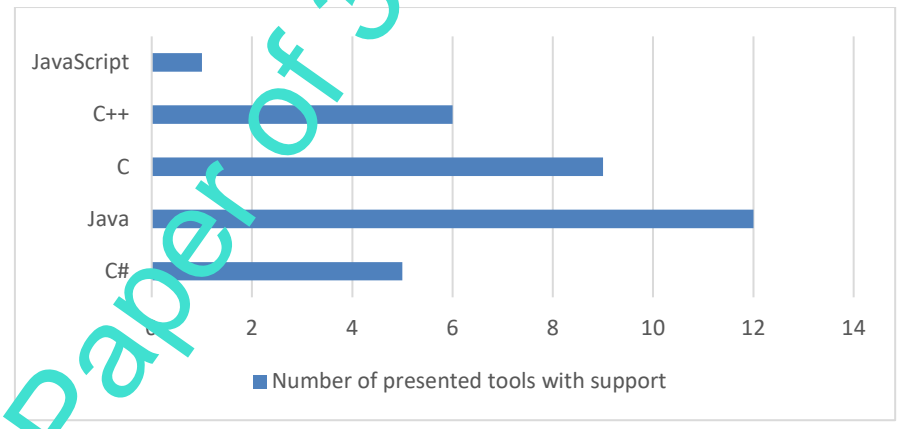

Fig. 5. Supported programming languages by the tool(s) presented in papers

Table 2. presents extracted data that answers the second research question-RQ2: Is there a static code analysis tool that applies to a DSL?

Within the selected studies in the literature review, only three studies (14%) represent tools and support for DSLs. In contrast, other tools (86%) in the studies analyze the accuracy and efficiency of static code analysis tools that have support for one or more GPLs.

Programming language	Studies	%
DSL	[13], [29], [35]	14%
GPL	[16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [30], [31], [32], [33], [34], [36]	86%

Table 2. Primary studies classified on DSL and GPL

Figure 6 presents extracted data that answer the third research question-RQ3: Which tools are most commonly mentioned in the primary studies?

Six tools were presented, which were most often analysed and presented in primary studies. These are tools that have been presented three or more times in different researches. In comparison, other tools presented within the studies included in this literature review were analysed in two or one studies.

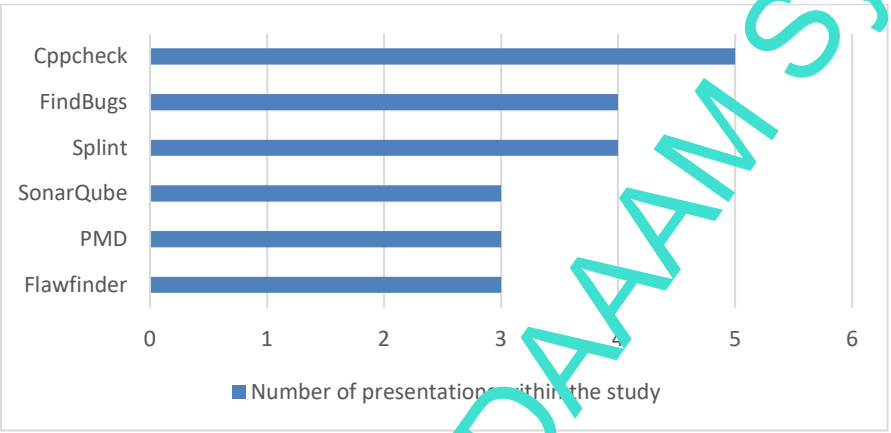

Fig. 6. The most common static code analysis tools presented in primary studies

In Table 3 are given the extracted data that answer the research question-RQ4: Is there a tool that detects all types of defects?

The problem observed in the extraction of data related to the types of defects to be detected is that the studies do not clearly and precisely highlight the types of defects that the proposed tools detect. Some studies do not clearly (or at all) highlight the types of defects that the proposed tools detect, and such studies [13] [26] [29] [30] [33] are excluded from this classification by type of defect.

Another problem observed is that studies use different taxonomies to classify defects. In this research, a unique taxonomy has been established, which is called Seven pernicious kingdoms [14], and all types of defects that various studies point out have been classified according to it.

Table 3 shows the classification of studies according to the types of defects detected within the presented tools.

Seven Pernicious Kingdoms	Studies
Input validation and representation	[16], [17], [18], [19], [20], [21], [23], [24], [25], [28], [32], [34], [35], [36]
API abuse	[20], [28], [31], [36]
Security features	[16], [20], [21], [25], [27], [28], [36]
Time and state	[20], [21], [31], [32]
Errors	[16], [19], [20], [21], [23], [28], [32]
Code quality	[16], [20], [21], [23], [28], [31], [32], [35]
Encapsulation	[20]
Environment	[28]

Table 3. Primary studies classified by type of defects that presented tools detect

4. Discussion

The results are commented in this section of the conducted systematic literature review, presented in section 3.3.

The highest number of the reviewed studies were published in the years 2012 and 2016. However, the number of papers on an annual basis is uniform, and research related to static code analysis tools is becoming more common, and more importance is attached to these tools in order to increase the quality of the source code of the software solution.

Special mention should be made of the Master thesis derived from Alto University, School of Science, as a study that describes in the most detail the presented tools and the experiment on those tools, which makes a great contribution in this area.

In order to answer the research question RQ1, the tools presented in the studies were analysed, it is concluded that each of the proposed tools has support for at least one GPL. Some of the tools, such as the tool presented in [35], have support for a number of GPLs, while the tool presented in [22] has support for just a single GPL. In the context of the question RQ1 it is important to consider the answer to the question RQ2. Lessons learned from the development of tools that can support DSL can be applied to the development of tools for the GPL.

The second research question RQ2 is answered positively. Creating a static code analysis tool that will provide support for DSLs is of great importance in this area. Of the studies presented in this literature review, three studies highlight tools for static code analysis of programs written by means of DSL. The framework within the study [35] is presented, and the way of using and efficiency of using the tool over three static code analysis of programs written by means of DSL is explained. On the other hand, the tool, which is otherwise intended for GPL, is presented in the study [13] as a possible solution for conducting static code analysis over DSLs. In this study, they highlight a plug-in that defines source code quality control rules for any DSL Also, within [13], an experiment is performed on a DSL, and the obtained results were presented. Based on the presented, it is concluded that within the already developed tool for static code analysis, there is a way to provide code overview and detect defects over any language specific for the domain.

To answer the research question RQ3, the tools presented in the studies were analysed, it is concluded that some tools for static analysis are more often used in research than others. It mainly depends on which GPLs the proposed tools support. Thus, the Cppcheck tool is used in five different studies [16] [21] [28] [32] [36], FindBugs in four [17] [21] [26] [31], as well as Splint studies [16] [21] [28] [32], followed by the SonarQube tool presented in three studies [13][17] [33], as well as the PMD tool studies [16] [21] [26] and the static code analysis tool Findbugfinder is also presented in three studies [16] [21] [28]. The reason for this lies in the fact that these tools are free and are most often used for static code analysis, so their accuracy and metrics offered within the programming languages for which they are supported are often examined.

The research question RQ4 is answered in the negative, i.e., no tool or study includes all types of defects according to the presented taxonomy. The study [20] presents seven of the eight main types of defects identified by the classification scheme. Other studies included in this review detect only certain types of defects. The biggest problem in distinguishing types of defects from tools and studies and their classification is that most studies use different schemes for defect classification.

5. Conclusion

Static code analysis tools are increasingly finding their application and use in the work of software developers in different positions. New tools are being developed, and existing tools are being improved every day, offering their customers an increasingly high-quality service by creating confidence in the results of static analysis and enabling them to avoid certain code defects and make their software solution better.

Still, static code analysis tools allow the analysis of only a certain number of programming languages (RQ1). However, the range of programming languages for which they have support is continuously expanding. In comparison, it is expected that in some future period, it will work on creating and improving several tools that will have support for DSLs (RQ2). For now, only a few tools have support for DSLs with the specification of rules embedded in the tool via a plug-in.

One of the research questions in this paper is which tools are most often used in research (RQ3). The results show that certain tools are used more often in studies than others. It is concluded that the research is focused mainly on tools that are more often used in practice and that have support for certain GPL. In this way, the research compares the results of the accuracy and efficiency of the presented tools.

Based on the obtained results that provide an answer to the research question RQ4, it is concluded that no tool can detect all defects defined by the presented classification, while a combination of certain tools can solve the presented problem. Also, the same classification is used in the paper for all studies, which leads to an easier understanding of the difference between the tools presented in the studies.

Further research involves performing experiments on the presented tools to compare their advantages and disadvantages in terms of static code analysis, as well as expanding the scope of studies and tools for static code analysis and their classification according to the supported programming languages and the defects they detect.

The use of tools for static code analysis and continuous work on improving and maintaining the quality of source code provides higher quality software solutions and represents a step forward in software development.

6. References

[1] Asenovic, M.; Sladojevic S.; Anderla A. & Stefanovic, D. (2018) “Deep neural network ensemble architecture for eye movements classification” in 2018 17th International Symposium on Inoteh-Jahorina, Infoteh 2018 - Proceedings, Jahorina

[2] Beric D.; Havzi, S.; Lolic, T.; Simeunovic, N. & Stefanovic, D. (2020) “Development of the MES software and integration with an existing ERP Software in Industrial Enterprise,” in 2020 19th International Symposium Infoteh-Jahorina, Infoteh - Proceedings, Jahorina.

[3] Beric, D.; Stefanovic, D; Cosic, I. & Lalic, B. (2018) “The implementation of ERP and MES Systems as a support to industrial management systems,” International Journal of Industrial Engineering and Management, vol. 9, no. 2, pp. 77-86
[4] Dakic D.; Stefanovic D.; Cosic, I; Lolic, T. & Medojevic, M. (2018) “ Business Process Mining Application: A Literature Review,” in Proceedings of the 29th DAAAM International Symposium, Vienna
[5] Jokic, M.; Gracanin, D. & Lalic, B. (2019) “Sustainable Low Cost and High-Quality Supply Chain Assurance-A Systematic Literature Review,” in Proceedings of the 30th DAAAM International Symposium, Vienna, Austria
[6] Blahova, M.; Mach V.; Pavlik, L.; Hormada, M.; & Ficek, M. (2019) “ The Information Security to Software of Crisis Management,” in Proceedings of the 30th DAAAM, Vienna
[7] Suleykin, A. & Panfilov P. (2019) “Implementing Big Data Processing Workflows Using Open Source Technologies,” in Proceedings of the 30th DAAAM International Symposium, Vienna
[8] Moller, A. & Schwartzbach, I.(2019) "Static program analysis"
[9] Beller, M.; Bholanath, R.; & S. M. (2016) "Analyzing the state of static analysis: A large-scale evaluation in open source software"
[10] Kitchenham, B. (2004) Procedures for Undertaking Systematic Reviews, Computer Science Department, Keele University (TR/SE-0401) and National ICT Australia Ltd. (0400011T.1)
[11] West, J. & Chess, B. (2007) Secure programming with static code analysis, Pearson Education
[12] Fowler, M. (2010) "Domain-Specific Languages," in Pearson Education, London
[13] Ruiz-Rube, I.; Peson, T.; Dodero, M. J.; Mota, M. J.; & Sanchez-Jara, J. (2020) "Applying static code analysis for domain-specific languages," Software & Systems Modeling, vol. 19, no. 1, pp. 95-110
[14] Tsipenyuk, K.; Chess, B. & McGraw, G. (2005) “Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors,” IEEE Security and Privacy, vol. 3, no. 6, pp. 81-84
[15] Tuma, K.; Calikli, G. & Scandariato, R. (2018) "Threat analysis of software systems: A systematic literature review," Journal of Systems and Software, vol. 144, pp. 275-294
[16] Kaur, A. & Nayyar, R. (2019) 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, pp. 2023-2029
[17] Marcilio, D.; Furia, C.; Bonifacio, R. & Gustavo, P. (2020) "SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings," The Journal of Systems & Software, vol. 168
[18] Nunes, P.; Medeiros, I.; Fonseca, J.; Neves, N.; Correia, M. & Vieira, M. (2019) "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
[19] Meghanathan, N. (2013) "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
[20] Goseva-Popstojanova, K. & Perhinschi, A. (2015) "On the capability of static code analysis to detect security vulnerabilities," Information and Software Technology, pp. 18-33
[21] Chahar, C.; Chauchan, V. & Das, M. (2012) "Code Analysis for Software and System Security Using Open Source Tools," Infromation security journal, vol. 21, no. 6, pp. 346-352
[22] Seshagiri, P.; Vazhayil, A. & Sriram, V. (2016) "AMA: Static Code Analysis of Web Page for the Detection of Malicious Scripts," Procedia Computer Science, vol. 93, pp. 769-773
[23] Vert, T.; Krikun, T. & Glukhikh, M. (2013) "Detection of incorrect pointer dereferences for C/C++ programs using static code analysis and logical inference," in Tools & Methods of Program Analysis
[24] Liu, X. & Cai, W. (2011) "A program vulnerabilities detection frame by static code analysis and model checking," in 2011 IEEE 3rd International Conference on Communication Software & Networks (ICCSN)
[25] Antunes, N. & Vieira, M. (2010) "Benchmarking Vulnerability Detection Tools for Web Services," in 2010 IEEE International Conference on Web Services (ICWS)
[26] Fiorella, Z.; Simone, S.; Rocco, O.; Gerardo, C. & Massimiliano D. P. (2017) "How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines," in 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)
[27] Zhioua, Z.; Short, S. & Roudier, Y. (2014) “Static Code Analysis for Software Security Verification: Problems and Approaches,” in 2014 IEEE 38th International Computer Software & Applications Conference Workshops
[28] Arusoaie, A.; Ciobâca, S.; Craciun, V.; Gavrilut, D. & Lucanu, D. (2017) "A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code," in 2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)
[29] Madurapperuma, M.; Mahagamage, D.; Madhavi, P.; Madushan, J. & Wijesiriwardana, C. (2016) "Domain specific infrastructure for code smell detection in large-scale software systems," in International Research Symposium on Engineering Advancements 2016 (IRSEA)
[30] Manzoor, N.; Munir, H. & Mayyed, M. (2012) "Comparison of Static Analysis Tools for Finding Concurrency Bugs," in IEEE--Institute of Electrical and Electronics Engineers
[31] Mamun, M.; Khanam, A.; Grahn, H. & Feldt, R. (2010) "Comparing Four Static Analysis Tools for Java Concurrency Bugs," Göteborg
[32] Chatzieleftheriou, G.; Chatzopoulos, A. & Katsaros, P. (2011) "Test-Driving Static Analysis Tools in Search of C Code Vulnerabilities," in Lecture Notes In Computer Science, (8803), Heidelberg

[33] Javier, G.; Marisol, G. & Escribano-Barreno, J. (2016) "Improved Metrics Handling in SonarQube for Software Quality Monitoring," in Distributed Computing and Artificial Intelligence, 13th International Conference
[34] Xie, J.; Lipford, H. & Chu, B. (2012) “Evaluating Interactive Support for Secure Programming,” in CHI - Conference, New York
[35] Mandal, A.; Mohan, D.; Jetley, R.; Nair, S. & D'Souza, M. (2018) "A Generic Static Analysis Framework for Domain-specific Languages," in IEEE International Conference on Emerging Technologies and Factory Automation, ETFA
[36] Penttila, E. (2014) "Improving C++ Software Quality with Static Code Analysis"