DOI: 10.2507/34th.daaam.proceedings.xxx

CHALLENGES IN INTEGRATING STATIC CODE ANALYSIS INTO DOMAIN-SPECIFIC LANGUAGE

Danilo Nikolic, Andjela Todoric, Dusanka Dakic, Teodora Vuckovic & Darko Stefanovic

This Publication has to be referred as: Nikolic, D[anilo]; Todoric, A[ndjela]; Dakic, D[usanka]; Vuckovic, T[eodora] & Stefanovic, D[arko] (2023). Challenges in Integrating Static Code Analysis into Domain-Specific Language, Proceedings of the 34th 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: 10.2507/34th.daaam.proceedings.xxx

Abstract

Domain-specific languages (DSLs) have become increasingly popular for addressing specific domain problems in industry and academia. However, implementing static code analysis tools for DSLs presents unique challenges due to the evolving nature of the field. This paper explores major hurdles in implementing static code analysis tools for DSLs, including customization needs, limited tooling support, and complexity and expertise requirements. It emphasizes the significance of overcoming these hurdles to successfully integrate static code analysis tools into DSLs, enhancing code analysis and improving quality in DSL-based projects.

Keywords: domain-specific languages; static code analysis; integration; implementation challenges;

1. Introduction

In computer science and software engineering, domain-specific languages (DSLs) represent an important field of research. A DSL is a language adapted to a specific application domain and offers appropriate notations and abstractions [1]. Compared to general-purpose languages (GPLs), DSLs are more expressive and easier to use, increasing productivity and decreasing maintenance costs [1].

Providing the ability to assess the quality of DSL source code artifacts using well-known static analysis techniques can be an additional factor contributing to the successful adoption of DSLs [2]. Static program analysis enables the evaluation of software quality characteristics and the detection of various code smells and potential errors by using automatic tools for source code analysis [2]. GPLs can be used to build computer software with high-quality features and flexibility. Furthermore, DSLs offer ease of use and statement writing, with users not being required to have extensive domain knowledge.

Compared to GPLs, DSLs have a small community of software developers as well as domain experts. By using a DSL, domain experts can be more involved in developing a solution to a problem. They can more easily understand, check the code and what impact a particular change will have [3].

Usability is one of the crucial attributes in language evaluation. DSL usability evaluation can be used in the language design process, which can significantly affect user productivity [4].

This paper is concerned with identifying challenges that may arise when integrating static code analysis into domain-specific languages. In order to improve the quality of DSLs, it is important to identify potential problems, thus enabling more efficient static analysis of the code. Solving the challenges leads to more reliable software solutions that rely on DSLs.

Understanding the challenges of integrating static code analysis into DSLs is crucial for efficient development and quality assurance. It enables developers to anticipate obstacles, allocate resources effectively, and ensure compliance with domain-specific rules and standards. By identifying these challenges, tool developers and researchers can develop specialized analysis tools and frameworks, driving the adoption and usage of DSLs in various domains. Stakeholders, including developers, quality assurance teams, tool developers, researchers, and organizations, benefit from this understanding, leading to improved development processes, code quality, and tooling support in DSL-based projects.

Previous research in the field has explored the implementation and evaluation of various static code analysis tools on GPLs. These studies have investigated strategies for effective deployment, aiming to optimize code quality and identify vulnerabilities in GPL-based projects [5], [6], [8]. As a first step in this research, a systematic review of the literature in the field of DSL, as well as a static analysis of the code, was organized. In this way, relying on the literature in this field, various challenges in using static code analysis in integration with domain-specific languages are presented and analysed. The descriptive analysis also includes general purpose language, in order to adequately present the difference between GPL and DSL. The aim of this paper is to show aspects of the integration of static code analysis in domain-specific languages, and thus cover all the differences between GPL and DSL.

The remainder of the paper is structured as follows. The theoretical foundations of DSL and static code analysis are presented in Section 2. Section 3 reviews related works in this area. The following are the challenges of integrating static code analysis into DSL, presented in Section 4. Finally, Section 5 concludes the paper.

2. Theoretical foundations

This section provides a theoretical background on domain-specific languages and static code analysis.

2.1. Domain-specific languages (DSL)

The concept of DSL is based on the idea that a programming language can be designed specifically for a specific domain [9]. The use of DSLs is diverse. In addition to documenting the requirements and behaviour of a single domain, it can also be used to generate programs for the addressed domain [10]. By focusing on particular domain needs, DSLs can be more efficient and easier to use than GPLs.

A DSL development process includes an assessment of the effectiveness, usability, and suitability of the language for its purpose [4]. By using DSL, it is possible to hide implementation details and improve code readability [9]. Compared to GPLs, DSLs can offer significant gains in expressiveness and ease of use [9].

In software development, DSLs can significantly increase productivity and quality [11]. Also, since DSLs are explicitly designed for a specific domain, they can provide higher precision and accuracy than general-purpose languages. On the other hand, GPLs are designed to be applicable to a wide range of domains and applications [12]. For this reason, GPLs can be more flexible and adaptable to changing demands.

2.2. Static code analysis

With static code analysis, the source code can be analysed, and in this way, various defects in the program can be identified without its execution [8]. Static code analysis aims to detect problems early in development before the code is deployed. One significant advantage is their ability to identify the root cause of security issues rather than merely indicating symptoms [7]. This is particularly vital for ensuring that defects are effectively resolved.

Among the features offered by static code analysis, the most valuable one is the ability to identify common programming defects. Similar to how antivirus software detects viruses, static code analysis tools scan source code based on a specific set of patterns or rules [6]. Practitioners can use more advanced tools to add new rules to a set of predefined ones. However, it is important to note that if the patterns or rules do not specify a particular behaviour, the tools may not be able to detect corresponding errors.

During the analysis process, false positive and false negative errors may occur [13], [14]. A false positive error occurs if a static code analysis tool detects a problem that is not actually a problem in the code [14]. A large number of false positive errors can cause operational difficulties. In the analysis process, false positive errors are undesirable, but from a security perspective, false negative errors are a bigger problem. A false negative error occurs when it is impossible to detect a real problem in the code [14]. It implies that the source code still contains defects, even after the static analysis procedure is completed. As these errors can be problematic, more tools and methods should be used to cover more potential problems.

3. Related work

In recent years, DSL has become increasingly popular among professionals in various sectors and has more interest in the industry and literature. Also, research on the integration of static code analysis into DSL is increasingly mentioned in multiple works [2], [13], [14], [15], [16], [17], [18]. Despite the large number of existing DSLs, a lot of research is based on defining a new way of approaching and building DSLs [13], [15], [20].

In the paper [9], the authors believe that using DSL instead of GPL makes the code more readable, the mapping between the model and the code becomes more apparent, and thus the development time is reduced. Also, another advantage over the GPL is that DSLs can be updated and extended more easily than entire programming languages, such as Java, C, C++, etc. [9]. Specifically, the authors in the paper [18] showed that using DSL for robot systems is better than GPL. The authors [1] explain the difference between understanding DSL and the GPL, as well as the advantages of DSL over GPL.

In [15], the authors presented that static code analysis tools usually combine different types of analysis and look for a specific set of patterns or rules in the source code. The analyses performed by these tools consist of several independent rules that users can select each time the tool is run. These tools cannot find errors if patterns or rules do not specify such behaviour.

Given that patterns are significant both in software engineering and domain engineering, the paper [10] presented that DSL development can be integrated with application development. Also, with the help of forms, a solution can be provided for some common domain engineering problems. To start using DSL agilely, developers need to apply the patterns in the context of their existing programming knowledge.

Since GPLs are not always the most suitable for solving problems, DSLs have a significant advantage. DSLs can work independently but can also be designed from scratch or by extending some basic language [19].

The integration of static code analysis into DSL has gained traction in recent years, as evidenced by the growing interest in the industry and literature. The advantages of DSL over GPL, such as improved code readability, clear model-code mapping, and easier extensibility, make it a promising avenue for implementing static code analysis tools in the context of DSLs.

4. Aspects in the integration of static code analysis in domain-specific languages

Domain-specific languages are used to reduce development time and thus allow developers to work at a higher level of abstraction. Static code analysis is used to analyse the source code without executing it, thus identifying potential bugs, security vulnerabilities, and other problems [2].

Based on a descriptive review of the literature in this area and previous work, the conclusions are presented in this section. Several authors addressed some of the presented problems [9], [10], [11], [15], [18], [4].

Understanding the distinction between GPLs and DSLs is essential because it helps developers and tooling experts recognize the unique characteristics and challenges associated with each type of language. When integrating static code analysis into DSLs, it becomes crucial to tailor the analysis techniques and tooling to the specific constructs, abstractions, and the constraints of DSL, rather than relying solely on existing tools built for GPLs.

To integrate static code analysis into domain-specific languages in the best possible way, it is necessary to see the differences between general-purpose and domain-specific languages, as shown in Table 1.

	Domain-specific languages	**General-purpose languages**	**Source**
Development time	Save time during development	Long development time	[11]
Complexity	DSL syntax and semantics aid ease of parsing	A large, diverse codebase can make analysis difficult	[4], [10]
Identification of potential errors	The declarative approach describes the desired result	Lack of specialized functions for a specific domain	[10]
Community of experts	Lack of expertise among developers due to specificity	The larger community of developers and domain experts to develop tools	[2], [4]
Standardization	Lack of standardization can hinder the development of analysis tools	Established standardization for easier development of analysis tools	[9], [18]
Specificity	Precisely identifies problems due to specificity	Wide range of applications designed for flexibility and versatility	[13], [15], [18]

Table 1. Limitations and differences between DSL and GPL

Significant benefits can be seen by integrating static code analysis with domain-specific languages. One advantage is saving time during program development using DSL because errors and potential problems can be noticed much earlier in the process itself [11]. Therefore, removing them is much simpler. Integrating static code analysis with DSL can improve the quality of DSL by maintaining good design practices and principles. On the other hand, the time during program development using the GPL is significantly longer. As a result, it takes more time to debug a program developed using GPL.

Defining the syntax and semantics of the language facilitates the analysis of DSL itself. When designing the syntax and semantics, programmers can make DSL relatively easy to interpret, thus facilitating the identification of potential errors and problems through static code analysis [4]. For example, a language could be designed to facilitate parsing or to ensure that certain types of errors are impossible to express in the language itself. This could mean designing the language to facilitate the analysis or using a type system that catches common compile-time errors. The language could be designed to ensure that only valid operations and data types are used to help catch bugs early in development, thereby reducing the effort and time needed to fix those same bugs later. From Table 1 one can see the complexity of the language during its analysis. If the language has a large and diverse code base, the complexity may be greater, which is a characteristic of the GPL.

The declarative approach is a programming paradigm that emphasizes the logic and rules of the problem domain. It focuses on the desired result rather than the steps required to achieve it. With a declarative approach, a DSL can be easily analysed. In this way, identifying potential errors can be facilitated. Certainly, with the integration of static code analysis in DSL, a language is created that is more reliable, easy to use and maintain.

Another approach is to provide tools and frameworks that can enable the analysis of DSL code [10]. In this way, tools can be included to visualize and explore the structure of code and analyse the code for some common patterns and problems that may arise. Providing such tools and frameworks allows potential errors and issues to be more easily identified. In the case of complex DSLs that have many interdependent components, visualization tools can help to understand the structure and flow of DSL code and thus enable the identification of potential inefficiencies. Likewise, code analysis tools can identify common patterns and problems in DSL code, such as code smells or potential performance issues. Using tools and frameworks that support static code analysis can effectively improve the quality and reliability of DSLs, reducing the time and effort needed to debug and solve problems and making working with them more accessible. Combining these tools with a well-designed DSL can create a powerful and efficient tool for solving problems within a specific domain or application area.

The expertise and knowledge of programmers and researchers in that field are needed to analyse domain-specific languages effectively. Given that the area of static code analysis is still relatively new, a certain lack of expertise and knowledge can manifest itself in different ways. Because of language specificity, it can be difficult to develop practical analysis tools that can accurately identify problems.

In Table 1, the difference in standardization between DSL and GPL can be noticed. Established standardization can facilitate the development of analysis tools. But on the other hand, the lack of standardization or best practices for parsing domain-specific languages can challenge the development and implementation of efficient parsing algorithms that can be adopted and used [9].

General-purpose languages can be used for a wide variety of applications or tasks and are designed to be flexible and versatile. For this reason, they also lack some unique domain-specific features and constructs often found in domain-specific languages. These features may include domain-specific vocabulary, syntax, and semantics that reflect the specific needs and requirements of a particular application domain. This would be one of the challenges when analysing DSLs because the unique features of DSLs can make it difficult for static code analysis tools to accurately and reliably analyse code written in these languages. In the event that a DSL has domain-specific constructs and semantics that are not supported by existing analysis tools, then it can be quite challenging to identify potential problems and errors through static code analysis alone. It can lead to false positives or negatives, making it difficult to effectively use static code analysis to identify potential problems in DSL code. The best solution to these challenges is to develop custom analysis tools and frameworks designed to handle a particular DSL's unique features and constructs.

This may also include the development of custom parsers and parsing algorithms optimized for the specific syntax and grammar of the language, as well as incorporating domain-specific knowledge and heuristics into the parsing process [18]. To more easily identify the specific features and constructions of the language that are most important for analysis and to develop customized analysis tools that adapt to these needs, working with domain experts can be of significant importance. Aiming to improve the quality and reliability of DSL code, the static analysis of the code must be used efficiently [2]. Effective use can be achieved through tools for static code analysis that are designed for DSL, more precisely by integrating them into the development process itself. This can help to detect potential problems at an early stage of development.

Many existing static code analysis tools are designed and optimized for general-purpose languages such as C++, Java or Python. Adapting existing tools to work with domain-specific languages faces various challenges and may require a deep understanding of both DSLs and analysis tools. This can require significant expertise in language design, decomposition, and static analysis techniques. But it is also necessary to understand the domain and the specific requirements of DSL. The best solution is to develop new analysis tools for working with DSLs, but also existing analytical tools for working with DSLs can be modified or extended, which is also discussed in papers [13], [15]. Adapting

existing tools can be much more challenging than creating a new one because it requires the appropriate expertise and approach. Still, in this way, the quality and reliability of DSL code can be obtained.

Overall, based on the literature review and ongoing research, integrating static code analysis into domain-specific languages can improve the quality and reliability of DSLs and make it easier for developers to work with them. Previously mentioned is an area of ongoing research and development, and much remains to be explored regarding how best to integrate analysis techniques into DSLs.

5. Conclusion

This paper presents static code analysis in domain-specific languages to improve their quality and enable more efficient static code analysis. Various challenges and obstacles are associated with implementing static code analysis tools in domain-specific languages. One of the identified problems that can arise when integrating static code analysis into domain-specific languages is that they are not originally designed to support static code analysis and lack the basic features and constructs needed for efficient analysis. Consequently, existing static code analysis tools may be difficult or impossible to apply to such languages.

In addition, a lack of knowledge and expertise in static code analysis for domain-specific languages may exist since this field is still relatively new. The unique features and constructs of domain-specific languages can also present challenges because they may not be found in general-purpose languages, thus making it difficult for static code analysis tools to accurately and reliably analyse code written in these languages.

On the other hand, adapting existing static code analysis tools to work with domain-specific languages is another challenge since these tools are generally optimized for general-purpose languages. Therefore, modifying or extending tools to support domain-specific languages can require considerable effort and expertise.

Finally, implementing static code analysis tools in domain-specific languages requires overcoming various technical, knowledge and innovation challenges. Despite these difficulties, exploring the potential for incorporating static code analysis into domain-specific languages is crucial to improving their quality and supporting more efficient static analysis.

Future works could explore a concrete tool that would enable a static code analysis into domain-specific languages and thus overcome all the challenges identified in this paper. Additionally, future work could involve identifying ways of designing new and current DSLs to include the implementation of static code analysis tools easily. Through continued innovation and exploration, we can unlock the full potential of DSLs and enable their effective use in diverse application domains.

6. References

[1] Kosar, T.; Oliveira, N.; Mernik, M.; Varanda-Pereira, M. J., Črepinšek, M.; Cruz, D. & Henriques, P. R. (2010). Comparing General-Purpose and Domain-Specific Language: An Empirical Study. ComSIS Vol. 7, No.2, Special Issue, April 2010, pp. 247-264. https://doi.org/10.2298/CSIS1002247K

[2] Ruiz-Rube, I.; Person, T.; Dodero, J. M.; Mota, J. M. & Sanchez-Jara, J. M. (2019). Applying static code analysis for domain-specific languages. Springer-Verlag GmbH Germany 2019. https://doi.org/10.1007/s10270-019-00729-w

[3] Erazo-Garzon, L.; Cedillo, P.; Rossi, G. & Moyano, J. (2022). A Domain-Specific Language for Modeling IoT System Architectures That Support Monitoring. IEEE Access, Volume 10. https://doi.org/10.1109/ACCESS.2022.3181166

[4] Barišić, A.; Amaral, V.; Goulao, M. & Barroca, B. (2011). Quality in Use of Domain-Specific Languages: a Case Study. Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools, 65-67. https://doi.org/10.1145/2089155.2089170

[5] Stefanović, D.; Nikolić, D.; Havzi, S.; Lolić, T. & Dakić, D. (2021). Identification of strategies over tools for static code analysis. 9th International Conference on Engineering and Technology (ICET-2021), May 2021, Thailand. https://doi.org/10.1088/1757-899X/1163/1/012012

[6] Nikolić, D.; Havzi, S.; Dakić, D.; Lolić, T. & Stefanović, D. (2021). Evaluation of Strategies over Static Code Analysis Tools. Proceedings of the 32nd DAAAM International Symposium, Vienna, Austria. https://doi.org/10.2507/32nd.daaam.proceedings.069

[7] Nikolić, D.; Stefanović, D.; Dakić, D.; Sladojević, S. & Ristić, S. (2021). Analysis of the Tools for Static Code Analysis. 20th International Symposium INFOTEH-JAHORINA, March 2021, Jahorina. https://doi.org/10.1109/INFOTEH51037.2021.9400688

[8] Stefanović, D.; Nikolić, D.; Dakić, D.; Spasojević, I. & Ristić, S. (2020). Static Code Analysis Tools: A Systematic Literature Review. Proceedings of the 31st DAAAM International Symposium, Vienna, Austria. https://doi.org/10.2507/31st.daaam.proceedings.078

[9] Miller, A.; Han, J.,& Hybinette, M. (2010). Using domain-specific language for modeling and simulation: Scalation as a case study. Proceedings of the 2010 Winter Simulation Conference. https://doi.org/10.1109/WSC.2010.5679113

[10] Gunther, S.; Haupt, M. & Splieth, M. (2010). Agile Engineering of Internal Domain-Specific Languages with Dynamic Programming Languages. 2010 Fifth International Conference on Software Engineering Advances. https://doi.org/10.1109/ICSEA.2010.32

[11] Van den Bos, J. & van der Storm, T. (2013). A Case Study in Evidence-Based DSL Evolution. ECMFA 2013, Lecture Notes in Computer Science, vol 7949. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-39013-5_15

[12] Hrnčić, D.; Mernik, M. & Bryant, B. R. (2011). Embedding DSLs into GPLs: A grammatical inference approach. ISSN 1392-124X Information Technology and Control, 2011, Vol. 40, No. 4. http://dx.doi.org/10.5755/j01.itc.40.4.980

[13] Marcilio, D.; Furia, C. A.; Bonifacio, R. & Pinto, G. (2019). Automatically Generating Fix Suggestions in Response to Static Code Analysis Warnings. 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM). https://doi.org/10.1109/SCAM.2019.00013

[14] Schiewe, M.; Curtis, J.; Bushong, V. & Cerny, T. (2022). Advancing Static Code Analysis With Language-Agnostic Component Identification. IEEE Access, Volume 10. https://doi.org/10.1109/ACCESS.2022.3160485

[15] Bodden, E. (2018). Self-adaptive static analysis. 2018 ACM/IEEE 40th International Conference on Software Engineering: New Ideas and Emerging Results. https://doi.org/10.1145/3183399.3183401

[16] Palomba, F.; Bavota, G.; Di Penta, M.; Oliveto, R.; De Lucia, A. & Poshyvanyk, D. (2013). Detecting bad smells in source code using change history information. 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). https://doi.org/10.1109/ASE.2013.6693086

[17] Kuruppu, T.; Tharmaseelan, J.; Silva, C.; Samaratunge Arachchillage, U. S.; Manathunga, K.; Reyal, S.; Kodagoda, N. & Jayalath, T. (2021). Source Code based Approaches to Automate Marking in Programming Assignments. Proceedings of the 13th International Conference on Computer Supported Education, Volume 1, 291-298. https://doi.org/10.5220/0010400502910298

[18] Mandal, A.; Mohan, D.; Jetley, R.; Nair, S. & D'Souza, M. (2018). A Generic Static Analysis Framework for Domain-specific Languages. 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA). https://doi.org/10.1109/ETFA.2018.8502576

[19] Mohagheghi, P. & Haugen, Ø. (2010). Evaluating Domain-Specific Modelling Solutions. Advances in Conceptual Modeling – Applications and Challenges, ER 2010, Lecture Notes in Computer Science, vol 6413, Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-16385-2_27

[20] Ton, L. P. & Truong, T. M. (2015). Linking Rules and Conceptual Model in a Domain Specific Language. 2015 International Conference on Advanced Computing and Applications (ACOMP). https://doi.org/10.1109/ACOMP.2015.25