A case study comparing static analysis tools for evaluating SwiftUI projects

A case study comparing static analysis tools for evaluating SwiftUI projects

Gerald Birgen Imbugwa, Luiz Jonatã Pires de Araújo, Mansur Khazeev, Ewane Enombe, Harrif Saliu, Manuel Mazzara

Innopolis University, Universtikaya St. 1, Tartastan, Russia

E-mail: {g.imbugwa,l.araujo,m.khazeev,e.ewane,h.saliu}@innopolis.university
m.mazzara@innopolis.ru

Abstract. Declarative programming languages such as SwiftUI have gained increasing relevance for user interface implementation in mobile applications. A tool for evaluating and improving the quality of such projects is static analysis (SA). This study compares the usefulness of two of the most popular SA tools (SonarQube and Codacy) for evaluating real-world SwiftUI projects. Moreover, it recommends setup and adjustments to promote SA tools for SwiftUI projects that can be extended to other languages.

1. Introduction

Declarative programming is a paradigm in which a high-level computer program is used to specify the task to be done rather than a sequence of steps of how to do it. The program is then executed by an "inference engine" that investigates the written set of logic relations to infer the answer to a given query. Examples of such languages include Prolog and Swift, which have been introduced in the late 1960s and have a direct correspondence to mathematical logic. In this paradigm, the purpose is to allow developers to focus on the problem domain, leading to reduced development time and increased maintainability.

In the last five years, declarative programming has been used for implementing user interfaces in mobile applications. The list of benefits of its adoption includes the following features: a simplified coding convention, a smoother learning curve, the ability to run on different mobile platforms, a live preview of changes during the implementation without having to rebuild the entire application among a host of others, among other aspects. Moreover, recently introduced frameworks such as SwiftUI, Flutter [20] and React Native [29] contribute to a more efficient development process when using a declarative style of programming. However, such tools do not address code quality, a critical factor in the software development process.

The need for high-quality mobile applications requires tools and techniques to aid developers in identifying critical portions of the source code and measuring software quality. Quality assurance methods include testing [15], static analysis, prototyping or simulation, code inspection, and review among a host, to mention a few. These techniques address different aspects of the development process to guarantee that the application meets user requirements and performs as expected. This study focuses on static analysis, which is a verification conducted before the execution of software to determine whether given coding conventions are followed [14]. It is noteworthy that static analysis has been firstly designed and employed for imperative programming. Hence, there has been limited literature on using this quality inspection method for declarative programming, especially for mobile applications written in SwiftUI, for example.

This study aims to compare the effectiveness of some of the industry's most common tools for software quality assessment when evaluating real-world open-source projects written in Swift. Such quality ensuring methods and tools are expected to automatically detect anomalies and violations within the source code and enable one to verify whether the quality measures confirm the framework's official documentation. The results allow the community to gain valuable insights into the advantages and requirements for integrating such tools into the development process. In this study, the authors focus on SwiftUI for mobile applications, a modern declarative programming language with limited academic literature.

The remainder of this paper is organized as follows. Section 2 reviews the existing literature on static analysis and its use for declarative programming in the last decades. Section 3 presents the methodology for selecting the automatic static analysis tools, as well the real-world projects written in SwiftUI used as a benchmark. It also presents additional tasks necessary to address a common issue in the static analysis: false positives. A comparison of the results obtained by the selected tools is shown in Section 4 and discussed in Section 5. Lastly, Section summarizes the main findings, suggestions for the adoption of static analysis into the development cycle when using declarative programming and aspects to be approached by future research.

2. Literature Review

This section reviews the existing literature on static analysis and its use for declarative programming.

2.1. Static analysis for evaluating projects

Static analysis is the automated analysis of a program before its execution. It aims to aid development teams to verify software quality quickly. Static analysis is often used to detect security vulnerabilities, low performance, non-standard or outdated programming constructs, and noncompliance with enterprise standards. Static analysis tools can be classified according to the employed technique: parsing the source code into an abstract syntax tree (AST), regular expression matching, and combining both. While regular expression matching is easier to implement and allows more flexible rules for identifying errors, AST enables more specific, contextual matching and can reduce the number of false positives errors reported in the code.

There has been a considerable amount of research demonstrating the benefits of static analysis for imperative programming. For example, Wichmann et al. [28] pointed its usefulness in industrial settings. Examples of its adoption include major companies such as eBay [16], Google [6] and Microsoft [19]. However, the authors recognize that the benefits are often hard to quantify, diminishing its impact. Chess et al. [8] which stresses the benefits of performing code security analysis with static analysis tools such as BOON, CQual, MOPS, Splint, amongst others. For example, it can find security vulnerabilities in the early development stages. Prähofer et al. [25] stresses that static code analysis complements the work that compilers execute regarding the assessment of self-contained program quality by identifying bad code, violations of programming conventions and potential defects. One major drawback of the static analysis tools identified by the previous studies is the possibility to produce false positives errors.

2.2. Static analysis for declarative programming

Since its introduction, static analysis has been employed mostly for imperative programming languages like C++ and Java. There has been a limited number of studies using this technique for declarative programming. For example, Peralta et al. [23] demonstrated the use of SA for coding in an imperative language written in the form of logic constructs. In other words, a declarative program is derived from the original imperative before being analysed by the

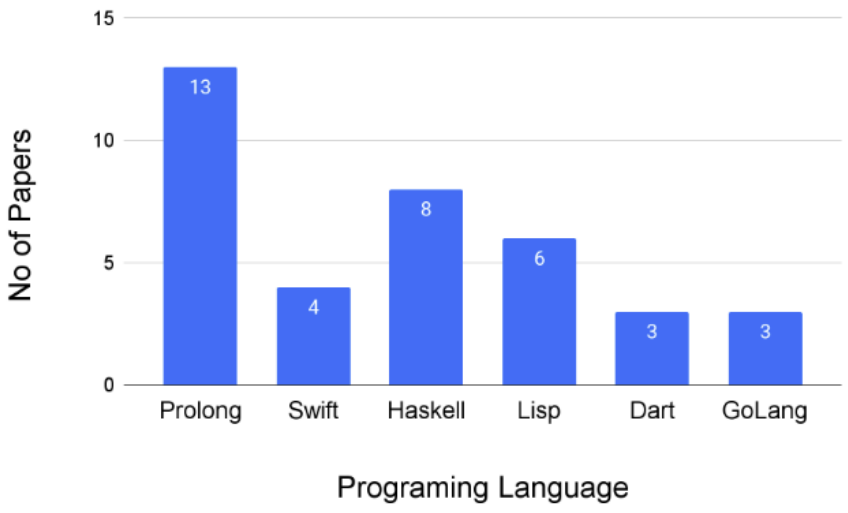

Figure 1. Number of papers published between 2000 and 2021 on static analysis per declarative programming language.

automatic tool. A limitation of this method is the requirement for parsing the code from imperative to declarative programming, which might introduce noise into the code and give limited insight into the semantics of the imperative program.

Most of the existing literature on static analysis for declarative programming targets Prolog applications with the purpose to evaluate and complement compiler techniques [17, 10, 12, 11]. Interestingly, some static analysis frameworks for specific declarative languages have been implemented in the last decades, including Lisp [21], Java [22], and Swift [27]. However, there is still a need for a more comprehensive body of research on the effectiveness of SA for declarative programming used in real-world projects. Furthermore, the number of papers published in the last decades on this topic is limited, as shown in Figure 1.

3. Methodology

This study assesses the usefulness of static analysis tools for evaluating the quality of real-world projects written in SwiftUI, a declarative programming language developed by Apple [3]. This section presents the criteria for selecting the automatic tools and the real-world projects used as a benchmark.

3.1. Requirements for the static analysis tools

The first step of the conducted research was to select static analysis tools that report some of the most relevant quality metrics [9]: number of issues, complexity, code duplication, code smell, hotsport, number of vulnerabilities, number of and code size. Specifically, for a number of issues, the results reported by the tool were validated by manual inspection of the code and the use of the existing Apple documentation [4]. In addition to the quality metrics mentioned above, the static analysis tools should present the following features:

- **SwiftUI support:** Most of the imperative programming languages have several static analysis tools available. SwiftUI, however, has a limited number of options.
- **Report on the number of code standard violations:** Code standards are rules and guidelines provided either by the programming language or the organization, leading to improved communication among teams, reduced program errors, and contributing to the overall software quality.

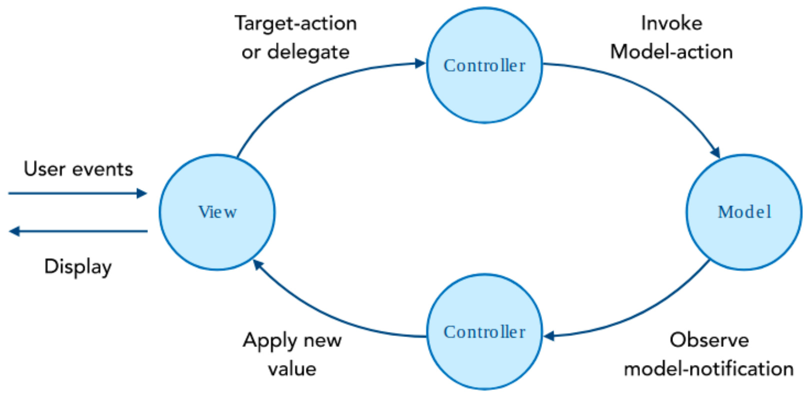

Figure 2. Model View Controller [13].

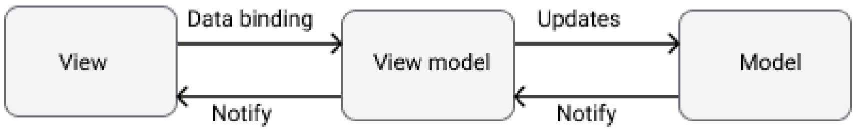

Figure 3. Model View ViewModel Architecture

- **User friendly on software installation:** Such a feature aims to promote the use of software quality control by heterogeneous teams. This also considers the compatibility with modern operating systems, required libraries and plugins, and available online support and documentation.

After an analysis of the existing tools and how they comply with the requirements above, the following two applications were selected: Codacy [9], and SonarQube [26].

3.2. Requirements for software projects
Next, open source real-world projects written by different teams and publicly available on GitHub were selected for analysis. Such an approach allowed the authors to focus on the testing instead of the implementation of possibly homogeneous applications. The second criterion was to select projects which have been implemented using different software architectures from the list:

- **Model-View:** In this architecture, the view layer is able to access the model layer, translate user inputs, and perform actions in the model [24].
- **Model-View-viewModel (MVVM):** View components each have a ViewModel component that provides a scene-specific state to the view [2].
- **Redux:** All views are able to access and view specific information in specific view structs using a global app state [24].

After filtering a preliminary list of SwiftUI projects available on GitHub, a subset of repositories was selected using the requirements as mentioned earlier. The projects selected for further investigation using static analysis tools are shown in Table 1.

As mentioned in Section 2.1, false positives are among the most critical issues of the results reported by static analysis tools. Therefore, the authors manually validated a sample of the issues identified by each SA tool. Lastly, the results from the tools are compared to enable one to gain insights about their effectiveness when assessing a declarative programming code.

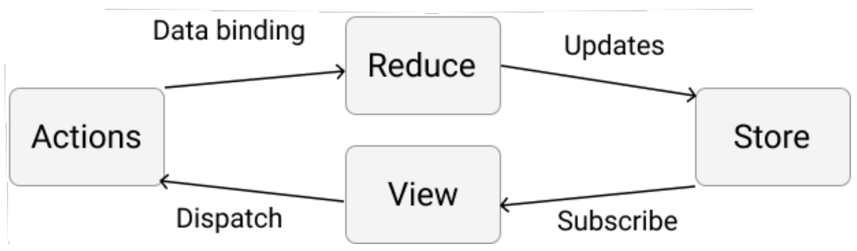

Figure 4. Redux Architecture

Table 1. Projects Metrics

Repository name	Description	Architecture	LoC
CoronaVirus-TrackerSwiftUI	It tracks the number of positive cases of COVID-19 around the globe.	Redux	1,464
DesignCode-SwiftUI	An app for converting design to code.	MVVM	926
FireTodo	An TODO application using Google Firestore.	Redux	1,379
FlipClock-SwiftUI	An app that shows the current time with a flip animation.	MV	263
MovieSwiftUI	A movie application.	Redux	7,468
NewsApp-With-SwiftUI-And-Combine	An application using an API from Apple to get news using the combine framework.	MVVM	14,034
SwiftUI-Chat	A chat application.	MVVM	309
SwiftUI-WeChat	A clone application of WeChat.	MVVM	2,600
SwiftUISignin	A library of authentication screens for different user interfaces.	MV	960
SwiftUITodo	An TODO application.	MVVM	319
Weather	A weather application.	MVVM	1,016

LoC: Lines of Code; MV: Model-View; MVVM: Model-View-viewModel.

4. Results

Firstly, the following steps were performed to run Codacy and SonarQube(sonarcloud) to assess the previously selected projects (see Table 1):

(i) Connect Codacy and SonaQube to GitHub
(ii) User authentication in the SA tool
(iii) Add the target repository from the GitHub
(iv) The project is forked into the SA tool
(v) The SA tool calculates the corresponding software quality metrics

The results obtained by Codacy are shown in Table 2. The following quality metrics are reported:

- **Repository name**
- **Grade:** Overall quality of each project, which ranges from A (high quality) to F (low quality) according to the average number of bugs per LOC [18, 9].
- **Issues:** Number of occurrences of bad or potentially harmful code, which falls into one of the following categories [9]:
  - Security: potential vulnerabilities and unsafe dependencies
  - Error-Prone: bad practices/patterns that cause the code to fail/prone to bugs

Table 2. Software quality metrics reported by Codacy.

Repository name	Grade	Issues	Complexity	Duplication
CoronaVirus-TrackerSwiftUI	B	19%	-	0%
DesignCode-SwiftUI	A	8%	-	0%
FireTodo	B	36%	0%	0%
FlipClock-SwiftUI	A	0%	-	0%
MovieSwiftUI	B	11%	-	2%
NewsApp-With-SwiftUI-And-Combine	B	19%	-	0%
SwiftUI-Chat	B	17%	-	0%
SwiftUI-WeChat	B	13%	-	0%
SwiftUISignin	B	27%	-	0%
SwiftUITodo	B	13%	-	0%
Weather	B	23%	-	0%

  – Code Style: related to the style of the code including line length, tabulation instead of space, etc
  – Compatibility: code that has no support for legacy systems or cross-platform components
  – Unused code: unnecessary code
  – Performance: inefficient code regarding memory utilization and speed

- **Complexity:** A measure of the difficulty of understanding and maintaining the source code.
- **Duplication:** The percentage of code which is implemented more than once within the repository.

It is possible to notice from Table 2 that most of the projects are graded as having overall quality B. One reason is the apparent correlation between the number of issues and the grade. For example, two projects with the issue metric below 10% have grade A. It is also possible to observe that, for all projects but FireTodo, Codacy shows a dashed character for complexity. For FireTodo, however, Codacy shows 0%. The tool fails to indicate if the metric is nonapplicable or an error in the report presentation. Lastly, only MovieSwiftUI had a small percentage of duplicated code.

Next, SonarQube was executed against the selected projects. The quality metrics reported by the tool use the following letter grades: A (no open issue), B (minor), C (major), D (critical) and E (blocker). The results are shown in Table 3, which contains the following list of quality metrics:

- **Bugs:** The number and grade of errors in the source code. Such errors can be a source of reliability issues in the source code.
- **Vulnerabilities:** Number of issues that enable a malicious code to hack the source code.
- **Security hotspots:** Percentage of sensitive parts of the code based that require attention.
- **Code smell:** Number of portions of the code that are confusing and difficult to maintain.
- **Duplication:** Code that appears more than once within the repository.

It is noteworthy that all repositories had zero bugs except for MovieSwift, which had one bug but grade C instead of B. This indicates that other factors are considered when grading this metric in addition to the number of errors. One reason can be the higher number of code smells

Table 3. Software quality metrics reported by SonarQube.

Project name	**Bugs**	**Vulnerability**	**Hotspot reviewed**	**Code smells**	**Duplication**
CoronaVirus-TrackerSwiftUI	0 (A)	0 (A)	100% (A)	15 (A)	0.0
DesignCode-SwiftU	0 (A)	0 (A)	100% (A)	23 (A)	0.0
FireTodo	0 (A)	0 (A)	100% (A)	28 (A)	0.0
FlipClock-SwiftUI	0 (A)	0 (A)	100% (A)	1 (A)	0.0
MovieSwiftUI	1 (C)	0 (A)	100% (A)	171 (A)	0.4
NewsApp-With-SwiftUI-And-Combine	0 (A)	0 (A)	100% (A)	21 (A)	3.7
SwiftUI-Chat	0 (A)	0 (A)	100% (A)	7 (A)	0.0
SwiftUI-WeChat	0 (A)	0 (A)	100% (A)	49 (A)	6.5
SwiftUISignin	0 (A)	0 (A)	100% (A)	47 (A)	0.0
SwiftUITodo	0 (A)	0 (A)	100% (A)	11(A)	0.0
Weather	0 (A)	0 (A)	100% (A)	18(A)	0.0

of this project. Again, there is an apparent lack of a more transparent evaluation and grading by the SA tool. Lastly, SonarQube was more successful than Codacy in identifying duplicated codes.

An analysis of both tools indicates a need for more transparent grading criteria of the quality metrics. For example, Codacy combines quality metrics, which prevents a better understanding of the aggregate result. Moreover, these tools generate many false positives because of the verification of supporting code (e.g., read me files).

It is noteworthy that most of the issues identified by Codacy and SonarQube were not related to the declarative programming nature of the code, since this type of error is also recurrent in imperative programming languages. This indicates that such tools are as applicable to declarative programming as they are for imperative code styles.

5. Discussion

SA tool is a leap forward to the code quality of any software project. The two tools(SonarQube and Cadacy) used for the experiment proved correctly to identify the coding-style violation despite failing in some areas.

The metrics generated for software quality differed from the two SA tools used during this research. Despite running the same project with different SA tools under the same category with the same steps. For example, duplication in Codacy one project only was flagged(MovieSwiftUI), whereas, in SonarQube, three projects were flagged(MovieSwiftUI, NewsApp-With-SwiftUI-And-Combine, SwiftUIWeChat). Additionally, the duplication metric in Codacy is not part of the overall software quality.

Rules used in analysing a project under test in the AS tools differed immensely. SonaQube Swift package has 172 rules used in analysing the software projects. Whereas, Codacy depends on linter packages from 3rd parties like the tailor, SwiftLint, and Jackson. The generated metrics might not be meaningful if the developer had already used the 3rd party linter library before running the project on Codacy. The extensive database of strict rules offered by SonarQube can be the reason for the effective metrics reported in all projects analysed.

Codacy offers the most desirable setup process with a few steps for configuration regarding ease of use. The Codacy UX/UI offers comfort to view the metrics at a glance and navigation to different subcategories of metric analysis. On the other hand, SonarQube suffers most from the complex setup process that is not friendly to beginners. SonaQube UX/UI is a bit complex as it offers extensive metrics. A new user using SonarQube can lose focus on the task at hand due to the number of charts and figures floating around.

The SA tools are of great benefit as they offer extensive metrics for accessing new paradigms in programming languages. The metrics features shared by both tools include; security, maintainability, duplication, and security. In addition, the SA tools offered rules configurations depending on the analysis required for a project. SonaQube goes further to add other sets of rule quality gates.

Lastly, a significant criticism of the use of the selected quality tools is related to the lack of transparent criteria for the overall grades of a project. One reason might be that such rates result from the aggregation of individual results for all the files in the project without considering their relative importance and role(e.g. source code, setup file) in the project. However, the integration of automatic tools and data-driven approaches in software quality is recommended for integration into the development cycle [7, 1, 5].

6. Conclusion

This study compared the usefulness of two among the most popular static analysis tools in the software development industry (i.e., Codacy and SonarQube) to assess the quality of real-world open-source projects written in SwiftUI. SwiftUI has gained increasing relevance in the software industry and is often used for developing applications in Apple's ecosystem. The tests consisted of running the tools for the projects, a manual validation for false positives, and a comparison of the results to gain insights into the possible incorporation of these tools into the development cycle.

The conducted experiments and results demonstrated that static analysis could correctly identify issues regarding security and especially code-style to some extend. However, the use of the selected tools requires some prior effort in setting up its use. For example, filtering files that do not contain source code are configuration files (e.g. markdown files). Moreover, a manual inspection of the code is also necessary due to false positives, i.e. parts of code that have been mistakenly flagged as containing errors. In SwiftUI projects, one of the reasons is the use of some programming patterns (e.g. closure). This study also identified a limitation on the use of static analysis tools which is the need for more transparent grading criteria for projects and artefacts. For example, there is an apparent correlation between grade and number of issues in the reports provided by Codacy. However, the available references do not provide information on whether the latter feature affects the former.

One of the aims of this study is to strengthen the currently limited amount of literature on static analysis for declarative programming languages. Real-world projects were selected and used as a benchmark for tools that are popular among development teams. The conducted experiments demonstrated their usefulness and indicated which adjustments are necessary for a more productive development cycle. Although there might be some criticism on the format of the reports provided by static analysis tools (e.g. Codacy and SonarQube), they can play a meaningful role in ascertaining software quality as already in imperative software development. Future research should address additional available declarative frameworks such as Flutter, Jetpack, Compose and React Native. Moreover, it might be beneficial to assess how static analysis can be integrated into the development process and activities such as compilation and interpretation without extensive manual code inspection.

References

[1] Akinsanya, B.J., Araújo, L.J., Charikova, M., Gimaeva, S., Grichshenko, A., Khan, A., Mazzara, M., Ozioma Okonicha, N., Shilintsev, D.: Machine learning and value generation in software development: a survey. In: International Conference on Tools and Methods for Program Analysis. pp. 44–55. Springer (2019)

[2] Aljamea, M., Alkandari, M.: Mmvmi: A validation model for mvc and mvvm design patterns in ios applications. IAENG Int. J. Comput. Sci **45**(3), 377–389 (2018)

[3] Apple: Declare the user interface and behavior for your app on every platform., `https://developer.apple.com/documentation/swiftui`

[4] Apple: Xcode and swift, `https://developer.apple.com/swift/resources/`

[5] Atif, F., Rodriguez, M., Araújo, L.J., Amartiwi, U., Akinsanya, B.J., Mazzara, M.: A survey on data science techniques for predicting software defects. In: International Conference on Advanced Information Networking and Applications. pp. 298–309. Springer (2021)

[6] Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., Zhou, Y.: Evaluating static analysis defect warnings on production software. In: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. pp. 1–8 (2007)

[7] Capizzi, A., Distefano, S., Araújo, L.J., Mazzara, M., Ahmad, M., Bobrov, E.: Anomaly detection in devops toolchain. In: International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment. pp. 37–51. Springer (2019)

[8] Chess, B., McGraw, G.: Static analysis for security. IEEE security & privacy **2**(6), 76–79 (2004)

[9] Codacy: Software metrics: A practical guide for the curious developer. pp. 2–29 (11 2020). https://doi.org/10.1109/TELFOR.2018.8612149

[10] De Boeck, P., Le Charlier, B.: Static type analysis of prolog procedures for ensuring correctness. In: International Workshop on Programming Language Implementation and Logic Programming. pp. 222–237. Springer (1990)

[11] Eichberg, M., Kahl, M., Saha, D., Mezini, M., Ostermann, K.: Automatic incrementalization of prolog based static analyses. In: International Symposium on Practical Aspects of Declarative Languages. pp. 109–123. Springer (2007)

[12] Filé, G., Rossi, S.: Static analysis of prolog with cut. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 134–145. Springer (1993)

[13] Gallagher, M.: Model-view-controller without the controller., `https://www.cocoawithlove.com/blog/mvc-without-the-c.html`

[14] Gomes, I., Morgado, P., Gomes, T., Moreira, R.: An overview on the static code analysis approach in software development (11 2020)

[15] Jamil, A., Arif, M., Abubakar, N., Ahmad, A.: Software testing techniques: A literature review. pp. 177–182 (11 2016)

[16] Jaspan, C., Chen, I.C., Sharma, A.: Understanding the value of program analysis tools. In: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion. pp. 963–970 (2007)

[17] Matsumoto, H.: A static analysis of prolog programs. ACM Sigplan Notices **20**(10), 48–59 (1985)

[18] McConnell, S.: Code Complete, Second Edition. Microsoft Press, USA (2004)

[19] Nagappan, N., Ball, T.: Static analysis tools as early indicators of pre-release defect density. In: Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005. pp. 580–586. IEEE (2005)

[20] Napoli, M.L.: Beginning Flutter: A Hands on Guide to App Development. John Wiley & Sons (2019)

[21] Narayanaswamy, K.: Static analysis-based program evolution support in the common lisp framework. In: Proceedings.[1989] 11th International Conference on Software Engineering. pp. 222–223. IEEE Computer Society (1988)

[22] Öqvist, J.: Contributions to Declarative Implementation of Static Program Analysis. Ph.D. thesis, Lund University (2018)

[23] Peralta, J.C., Gallagher, J.P., Sağlam, H.: Analysis of imperative programs through analysis of constraint logic programs. In: International Static Analysis Symposium. pp. 246–261. Springer (1998)

[24] Phan, D.H.: Benchmarking common architectural patterns in ios development (2019)

[25] Prähofer, H., Angerer, F., Ramler, R., Lacheiner, H., Grillenberger, F.: Opportunities and challenges of static code analysis of iec 61131-3 programs. In: Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012). pp. 1–8. IEEE (2012)

[26] SonarQube: Your teammate for code quality and security, `https://www.sonarqube.org`

[27] Tiganov, D., Cho, J., Ali, K., Dolby, J.: Swan: a static analysis framework for swift. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1640–1644 (2020)

[28] Wichmann, B.A., Canning, A., Clutterbuck, D., Winsborrow, L., Ward, N., Marsh, D.W.R.: Industrial perspective on static analysis. Software Engineering Journal **10**(2), 69–75 (1995)

[29] Wu, W.: React native vs flutter, cross-platforms mobile application frameworks (2018)