The Role of Open Source Software in Program Analysis for Reverse Engineering

Taher Ahmed Ghaleb
Information and Computer Science Department
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia
Email: g201106210@kfupm.edu.sa

***Abstract*—Program analysis is the process of statically or dynamically retrieving the structure and behavior of software systems. Static analysis solely relies on the availability of source code of computer programs, while dynamic analysis captures program information using execution traces during program runtime. The entire process is called software reverse engineering, where the extracted information could eventually be visualized to facilitate program comprehension for the sake of learning, maintenance, etc. Open source software, in this context, plays a vital role in developing, enriching, and validating program analysis techniques. In this paper, we show and discuss how open source software projects significantly contribute to the development, evolution, and validation of program analysis techniques as well as enriching reverse-engineered diagrams with useful and meaningful information, even for those techniques that rely on dynamic analysis.**

Keywords*—*Reverse engineering; Program analysis; Open Source Software; Software Validation and Evaluation

I. Introduction

To retrieve the structure or understand the behavior of software systems, it is important to employ some kind of program analysis [1]. If software source code is available, then static analysis could be carried out [2], while dynamic analysis can work even if the source code does not exist [3]. However, some dynamic techniques do require source code to be present at the time of analysis in order to conduct some instrumentation that could assist in collecting information about program behavior during runtime [4]. This means that the availability of program source code is essential for most program analysis techniques, especially when it comes to validating how such techniques perform if applied to real-world systems.

On the other hand, open source software plays an important part in the development and evolution of program analysis tools, and this is maily expressed by the high reliance on open source compiler frameworks [5]. In the literature, various program analysis techniques have been developed on top of open source compilers, which help in gathering Abstract Syntax Trees (ASTs) of programs that hold their entire design. After doing so, software structure and behavior may be deduced from such ASTs using specialized techniques for generating useful artifact representations.

What motivated us to write this paper is the noticed heavy demand for open source projects in almost all program analysis tools, even by dynamic and closed-source techniques. This implies that closed-source software systems are no longer useful in this matter. Therefore, this paper explores the need for and the reasons behind using open source software in the area of program analysis for reverse engineering. We study various program analysis approaches of the three main types, namely static, dynamic, and hybrid, and then we discuss their dependence on open source software, either in the development or evaluation.

The rest of this paper is organized as follows. Section II introduces and analyzes different techniques of program analysis. In Section III, we present some related concepts about open source technology and discuss how program analysis techniques demand the use of open source projects as a basis for their development, functionality, or validation. Finally, section IV concludes the paper and suggests the possible future work.

II. State of the Art

This section presents preliminaries about reverse engineering and the state-of-the-art techniques that support it. In this contexts, static, dynamic, and hybrid techniques that facilitate the reverse engineering of software structure and/or behavior are presented.

A. Reverse Engineering

Reverse engineering of software artifacts is the process of recovering the structure or behavior of software systems and exporting it in formats that can be used for maintenance or learning. Reverse engineering is mainly composed of a set of processes, including the static or dynamic analysis of programs to collect relevant structural or behavioral information, and the transformation of such gathered information into higher-level models [6]. The resulting representations can actually be used for various purposes, such model checking, program comprehension, performance monitoring, etc.

Usually, reverse engineering is conducted through heuristics, which may, subsequently, result in generating imprecise representations of program control flow of behavior. Although it is important to provide models reflecting programs precisely, users may sometimes need to focus on understanding a specific aspect of the program. Therefore, hiding trivial information

978-1-5090-4580-8/16/$31.00

from the produced output can sometimes reduce their complexity, which can offer better understanding of software artifacts

Performance, usability, and scalability are of the main characteristics of reverse engineering techniques, where trade-offs can occur among them. Producing thorough models, for example, gives a comprehensive overview of the software objects and interactions, but, at the same time, requires higher memory usage and may even negatively affect program understating. In addition, reverse engineering techniques might not be effectively applicable to large and complex systems as a whole, due primarily to a large number of infeasible paths present throughout their interprocedural control flows, which are known to be hard to identify in general [7].

B. Program Analysis Techniques

Program analysis is the set of processes concerned with analyzing different aspects of software programs for various purpose. In this paper, we focus on program analysis techniques that aim to reverse engineer program behavior and interactions. These techniques are divided into three categories of analysis. First, static analysis, which relies solely on the source code of the software system needed to be analyzed. Second, dynamic analysis, in which execution traces of the program are captured and analyzed during program runtime. Third, hybrid analysis, which combines both kinds of analysis within the same technique, which allows producing more useful information about program behavior.

1) Static Analysis: Static analysis is an exploration of software artifacts relying on the source code of the program in order to extract the structure and behavior that software systems including the control flow and all interactions between system objects [2]. Static analysis actually works without requiring the intended software systems to be executed [8]. It essentially works by parsing program source files and logging all information about interactions between internal system components. It firstly heads to the main entry point of the program (e.g., `main` method), and then start analyzing and keeping track of all object construction operations and their method invocations, taking into account all constructs that control the flow (e.g., `if`, `for`, etc.). The gathered information about program behavior is normally logged into memory as interaction traces to be visualized at a later stage.

Although static analysis is efficient, it does require to have open source software systems to functions, which may be the case for many software systems. On the other hand, it can only provide an approximation of the real program runtime, which may be caused by missing the dynamic features of programs, which itself may lead to producing less expressive models. Dynamic features of programs written in object-oriented languages may include dynamic class loading, multi-users, and multi-threading. However, static analysis can sometimes show an expectation of how such kind of interactions would be executed in the program at runtime. Thus, it was highly recommended in [9] and [10] to incorporate behavioral information through the static analysis of source code due to its importance and efficiency in covering different aspects of program control flow.

There have been various static techniques of program analysis in the literature. each technique was developed to achieve a specific goal of program understanding. For example, Lu *et al.* [11] studied the conformance and relationships the different sequence diagrams generated by their technique. On the other hand, Korshunova *et al.* [12] presented a tool that can statically reverse engineer class, sequence, and activity diagrams of any given C++ system by parsing its source code and then extracting its Abstract Syntax Tree (AST) in XMI files [13], which represent UML elements in an eXtensible Markup Language (XML) format. Rountev *et al.* [14] could introduce two simple extensions to the UML sequence diagram whose purpose was to precisely capture the intraprocedural control flow of programs. They designed a generalized break fragment that enables exiting from a certain surrounding fragment, in addition to an approach for mapping the control flow graph (CFG) to UML and handling any reducible exception-free CFG.

2) Dynamic Analysis: Unlike static analysis, dynamic analysis can work with the absence of program source code but requires programs to be executed instead [3]. Having source code is optional in dynamic analysis, but having it can facilitate injecting tracing functionality into programs by some techniques. Commonly, dynamic analysis-based depend on (customized) debuggers and profilers that can supply them with the required information about program behavior [15]. Although it can recover precise information about the actual behavior of a system, it is challenged by the coverage problem. A single run of the program can only explore a sample execution trace, which typically covers a few interactions out of many more possible ones. Augmenting coverage requires multiple executions of the program with varying parameters representing different execution scenarios, which will produce many diagrams causing another problem concerning how to combine them [4].

Moreover, control flow is not captured during the dynamic tracing [16]. It, therefore, lacks the ability to capture information about whether these interactions passed through conditional alternatives, repetitions, or recursive calls. Notice that such information can simply be collected if static analysis is employed. To allow dynamic analysis to identify control flow, it is important to instrument source code (or sometimes bytecode) to inject certain scripts for tracing the changes in the flow of the program. Such scripts can do their job while the program execution and hand-by-hand work collaborate with dynamic analysis [4]. Another issue with dynamic analysis is that it can only work with complete software systems, rather than software fragments, components, or subsystems. This issue can be resolved by instrumenting subsystems by automatically employing program stubs (e.g. a specially crafted main class) to enable tracing executions of a certain program component [17].

To minimize the complexity of diagrams that are generated by dynamic analysis, Jayaraman *et al.* [18] proposed

a method for summarizing the resulting sequence diagrams using state diagrams after multiple runs, but user participation was required in the summarization process. Briand *et al.* [4] introduced a dynamic analysis technique that addresses interactions of distributed systems where messages between different objects at different network nodes can be identified. Oechsle *et al.* [19] targeted students by proposing JAVAVIS to help them understand the interactions executed at runtime of their small-sized programs. In [7], a dynamic monitoring tool called Kieker was introduced to continuously (or on-demand) observe the behavior of Java programs, and its performance was evaluated in [20]. In [21], reducing sequence diagram size was the main goal, and that was achieved by removing the less important details and methods from them with the help of their previous tool Amida [22]. Ziadi *et al.* [23] also introduced a technique that performs dynamic analysis of Java programs based on their bytecode, and they applied the k-tail algorithm [16] to merge interactions that relate to the same fragment.

3) Hybrid Analysis: We have noticed that recent techniques started to focus on combining both aforementioned types of analysis together to produce a new compound program analysis called hybrid analysis. This kind of analysis is relatively effective and precise as the results produced from one analysis is complemented by the other's results [24], [25]. In other words, hybrid techniques exploit the power of both kinds of analysis while alleviating their weaknesses. Nevertheless, this kind of analysis is generally considered to be time-consuming as the instrumentation overhead increases with the implementation of the two types of analysis [24]. In addition, merging the multiple diagrams generated by both analyses into a single comprehensive diagram to represent the entire program behavior is a challenge as well [9].

Labiche *et al.* [24] presented a reverse engineering technique that combines both static and dynamic analyses aiming to reduce the instrumentation overhead required by the dynamic analysis, by collecting only a small amount of runtime information that cannot be derived from static analysis, like threads. Myers *et al.* [9] introduced a hybrid technique to improve the visual appearance of the generated sequence diagrams by introducing an algorithm that compacts a large amount of information of call/message interactions between system objects. Srinivasan *et al.* [26] also introduced a combined technique that merges and optimizes diagrams generated by both kinds of analysis according to object signatures and line numbers.

Another trend of reverse engineering techniques focuses on web-based applications [27], [28], [29]. In web-based applications, techniques were usually used to represent web pages as objects in the resulting diagrams, while the transactions between them represent the interaction messages. From a different perspective, modeling the behavior of mobile applications was also investigated in the literature, either using the UML standard or relevant extensions [30], [31]. Huang *et al.* [32] proposed a tool for modeling the stealthy behavior of Android application by detecting the semantic mismatch between the program behavior and user interface. Another work was proposed in [33] to enhance the understandability of the behavior of mobile applications by combining different artifacts, from different sources, to obtain a thorough and accurate model.

III. The Role Open Source Software

Open Source Software (OSS) development has gained a massive importance in software technology nowadays. Basically, open source projects are developed by teams of persons or by individuals. At present, several reputed software companies like Microsoft and Sun have been attracted to this trend of development due to its power to improve products' usability and extendability of the products [34], and to solve various problems of the companies, like the acceleration of development and evolution [35] and the enhancement of reliability [36]. In 1984, Stallman proposed the idea of Free Software Foundation, which then was confirmed in the Open Source Definition in 1997 [1] along with open source licenses, such as GNU General Public License (GPL), Common Development and Distribution License (CDDL), etc. Bonaccorsi *et al.* [37] presented the initiatives behind the success of open source software.

Web-based repositories of open source software projects play a vital role in the distribution of the source code of open source projects. GitHub[2], SourceForge[3] and CodePlex[4] are among the top and most popular alternatives of such a service. It is common to have them on more than one website to increase the availability and usability. Users through these websites can store, manage, maintain and deploy source code of their projects online easily. In addition, distributed revision control, bug tracking, and shared development are examples of the features that can be provided by such services. In relevance to open source compilers, most providers of deploy their open source compilers in these web-based hosting.

A. In the development of program analysis techniques

Building a parser to parse and analyze programs of a certain programming language is somewhat complicated and can lead to having wrong representations of some of the syntactic rules of that language, especially if that was based on textual analysis. For instance, Java grammar is well-known and there exist many open source parsers and corresponding compilers that can assist in correctly and efficiently reading Java programs. Therefore, such parsers can be utilized to build program analysis tools to retrieve the required information about analyzed programs.

Having an open source parser or compiler does not guarantee the constructions of a program analysis tool easily. Program analysis tools have different objectives and so have open source parsers and compilers. It may happen that some of the available open source parsers are not extensible, or at

[1] opensource.org
[2] github.com
[3] sourceforge.net
[4] codeplex.com

least not easy to be extended. This means that, in order to customize such parsers, you will need to understand its current functionality and then make the proper adjustments that serve your prospective features. This means that you would follow a non-modular way to accomplish your technique since various parts of the base parser/compiler will be modified.

On the other hand, the literature is full of compilers that facilitate the creation of extensions of programming languages' parsers. This means that users can utilize the extending functionality of a certain extensible compiler to add the required functionality of the analysis tools. This includes: determining the set of constructs to be retrieved, injecting appropriate logging operations, the passes at which programs should be analyzed, and specifying the intermediate representation holding the AST. Take into mind that some dynamic program analysis tools also rely on open source software even though they actually target executable software.

The final product after completing the implementation of a program analysis technique would either be open or closed source, which concludes that building a tool on top of another open source tool or framework does not necessarily produce an open source product. It is the owner decision to deploy the constructed tool under a proprietary copyright or an open source license.

As we can see in Table I, every single program analysis depends on another technique. That dependent technique can either be developed by the same authors (e.g., [21], [24], and [38]) or by others. The authors of techniques proposed in [9] and [11] did not provide information about any dependency on other techniques or tools, and thus we assumed that their used techniques are not open source. In addition, the tools that are unclear whether they are open source or not and the ones are not available online are assumed to be closed source. It can be observed that only a third of these techniques were built on top of open source tools, while the rest were built on closed source tools. This can indicate that open source program analysis tools might not be that useful by other tools, which is perhaps due to reliability issues.

TABLE I
PROGRAM ANALYSIS TECHNIQUES WITH THEIR DEPENDENT TECHNIQUES

Ref.	Dependent technique(s)	Open source?
[11]	Unknown	✗
[14]	Their tool: RED	✗
[12]	Columbus/CAN, DOT	✗
[4]	Unknown	✗
[18]	JIVE	✓
[19]	JDI	✗
[7]	UMLGraph	✓
[38]	Their tool: Amida [22]	✗
[21]	Their tool: Amida [22]	✗
[23]	K-tail algorithm [16]	✓
[39]	Oberon [40]	✓
[41]	JVM	✗
[42]	MAS [43]	✗
[44]	Rigi	✓
[15]	JExtractor, Rigi [45], SCED [46]	✓
[24]	Their work in [4]	✗
[9]	Unknown	✗
[26]	Visual Paradigm	✗

B. In retrieving important information about programs

Some important information about program structure and behavior cannot actually be extracted by analyzing the executable files of programs, which gives program source code a high significance in program analysis techniques in general. For example, without open source, it is difficult to capture the control flow of software programs. The best way to extract program control flow is to read all constructs in the program that participate in changing the path of the program counter, and that could only be achieved through parsing program source code. Due to this particular prominence of source code, some dynamic analysis techniques, though they dynamically analyze program execution, were based on program source code. However, some other dynamic techniques could derive different aspects of program control flow, such as conditions and repetitions, by employing certain algorithms whose goal is to compact interactions based on specific behavioral patterns that expose their enclosing parent blocks.

It is common for dynamic program analysis techniques to conduct a source code instrumentation to help in gathering information about program execution during runtime. To this end, source code is required even if the analysis of program will work at runtime. On the other hand, some techniques can instrument bytecode instead of source code [47], [48], which can effectively work with dynamic techniques and with closed source projects.

C. In validating program analysis techniques

To validate a program analysis technique, it is required from its developers to select proper case studies represented by real-world projects and run their technique over them [49]. The selection of such case studies can sometimes be positively or negatively biased to those that can expose good impressions about the developed technique. This bias may be intentional by the developers or by chance. As an example of negative biasing, developers may choose to validate their technique against a small-sized project as a case study just because the technique may fail if applied to larger scale ones. From a different perspective, developers may choose a project that can cover all (or most of) the aspects the technique's functionality. Generally, using projects of different types, scales, and purposes is preferable to show how techniques would perform under real circumstances.

Selecting either an open source or closed source project for technique validation is based on the nature of the analysis technique. Static techniques, for instance, always require nothing other than open source projects for its validation as source code represents its intrinsic input. Dynamic techniques, on the other hand, can either be validated with the presence or absence of source code. If the functionality of a dynamic technique requires extracting some information from source code, then it should necessarily be extant. If the technique

solely works on executables, then it does not matter whether the source code is available or not. Since hybrid techniques combine both kinds of analysis, static and dynamic, they also presuppose source code to be available since static analysis is part of its process.

Table II shows a list of different techniques along with their corresponding case studies: types and names of the projects used. We refer to each type of program analysis by a single character, where S refers to static analysis, D refers to dynamic analysis, and H refers to hybrid analysis. For the static and hybrid analyses, it is ordinary to use open source projects as case studies as they perform a source code analysis. On the other hand, we can see that the majority of dynamic techniques used open source projects as well as case studies, though they can work with executables. This confirms the high usability and usefulness of open source technology.

TABLE II
CASE STUDIES OF SAMPLE PROGRAM ANALYSIS TECHNIQUES

Ref.	Analysis	Case Study	Open source?
[11]	S	JHotDraw and Monetary Access Control	✓
[14]	S	21 Java library components	✓
[12]	S	30K LoCs, 60K LoCs projects	✓
[4]	D	A library system	✓
[18]	D	Dining Philosopher's problem	✓
[19]	D	Small-sized programs	✓
[7]	D	iBATIS JPetStore	✓
[38]	D	jEdit, Gemini, Scheduler, LogCompactor	✓
[21]	D	jEdit, Eclipse	✓
[23]	D	A project of 500+ classes/interfaces with 25k LOC	✓
[39]	D	Kepler + A Compiler Construction Framework	✓
[41]	D	A Technical Report System (TRS)	✓
[42]	D	Not mentioned	?
[44]	D	FUJABA project	✓
[15]	H	FUJABA project	✓
[24]	H	5 large systems and 2 small programs	✓
[9]	H	Eclipse IDE, HSQLDB, Jetty web server platform	✓
[26]	H	Polyshape, Animal, and Task Scheduler systems	✓

For the techniques that were validated and evaluated through controlled experiments, we noticed that although they are dynamic, they were compared with results of static techniques. For example, the work in [50] proposed a dynamic tool for analyzing the interactions within a program while it is running, in comparison with Eclipse that indeed requires source code to be available. Therefore, because of this and to make the results of that paper reproducible, authors selected an open source project. Another experience was with the technique proposed in [51], as they compared their tool with the one in [50] using two open source projects of different sizes, though both techniques rely on dynamic analysis. To summarize, open source projects still have the superiority over closed-source projects, especially when it comes to software experimentation and empirical evaluation.

IV. CONCLUSION

This paper discussed the different kinds of program analysis techniques and how open source software plays a vital role in their development, evolution, and validation. The aim of the paper is to show the importance of open source software from three different aspects. First, the use of open source tools, frameworks, and compilers to build new techniques on top of them. Second, the use of open source software to allow gathering as much useful information as possible from programs. Third, the use of open source projects to validate the outcome and performance of such techniques. Different program analysis techniques have been presented in the paper as examples of stakeholders of open source software, along with a discussion about their close correlation.

For future work, we suggest two possible research directions for the purpose of enhancing the connection between program analysis techniques and open source software. The first direction is concerned with constructing an open source framework for program analysis where techniques can be built as extensions to it. It would be necessary to support both kinds of analysis in such a framework so that new techniques can have an option of employing only one of them or hybridizing them together. The second direction, which can be aligned with the former one, is to introduce an open source benchmark that can be used for evaluating the different program analysis techniques. Such a benchmark would indeed help to gaining fair evaluation and comparative results.

ACKNOLEDGEMENT

The author would like to sincerely thank and appreciate his home institution, Taiz University - Yemen, which donors him a scholarship to pursue his graduate studies abroad.

REFERENCES

[1] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 2015.

[2] A. Gosain and G. Sharma, "Static analysis: A survey of techniques and tools," in *Intelligent Computing and Applications*. Springer, 2015, pp. 581–591.

[3] T. Ball, "The concept of dynamic analysis," in *Software Engineering - ESEC/FSE'99*. Springer, 1999, pp. 216–234.

[4] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the reverse engineering of UML sequence diagrams for distributed Java software," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 642–663, 2006.

[5] T. A. Ghaleb, "Toward open-source compilers in a cloud-based environment: the need and current challenges," in *2015 International Conference on Open Source Software Computing (OSSCOM)*. IEEE, 2015, pp. 1–6.

[6] L. C. Briand, "The experimental paradigm in reverse engineering: Role, challenges, and limitations," in *WCRE'06. 13th Working Conference on Reverse Engineering, 2006.* IEEE, 2006, pp. 3–8.

[7] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stoever, S. Giesecke, and W. Hasselbring, "Kieker: Continuous monitoring and on demand visualization of Java software behavior," in *Proceedings of the IASTED International Conference on Software Engineering*. ACTA Press, 2008.

[8] J. E. Grass, "Object-oriented design archaeology with CIA++," *Computing Systems*, vol. 5, no. 1, pp. 5–67, 1992.

[9] D. Myers, M.-A. Storey, and M. Salois, "Utilizing debug information to compact loops in large program traces," in *14th European Conference on Software Maintenance and Reengineering (CSMR), 2010.* IEEE, 2010, pp. 41–50.

[10] Y.-G. Guéhéneuc and T. Ziadi, "Automated reverse-engineering of UML v2.0 dynamic models," in *proceedings of the 6th ECOOP Workshop on Object-Oriented Reengineering*. Citeseer, 2005.

[11] L. Lu and D.-K. Kim, "Required behavior of sequence diagrams: Semantics and conformance," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 2, p. 15, 2014.

[12] E. Korshunova, M. Petkovic, M. van den Brand, and M. R. Mousavi, "CPP2XMI: reverse engineering of UML class, sequence, and activity diagrams from C++ source code," in *13th Working Conference on Reverse Engineering 2006 (WCRE'06)*. IEEE, 2006, pp. 297–298.

[13] T. J. Grose, G. C. Doney, and S. A. Brodsky, *Mastering XMI: Java Programming with XMI, XML and UML*. John Wiley & Sons, 2002, vol. 21.

[14] A. Rountev, O. Volgin, and M. Reddoch, "Static control-flow analysis for reverse engineering of UML sequence diagrams," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 1, pp. 96–102, 2005.

[15] T. Systä, K. Koskimies, and H. Müller, "Shimba—an environment for reverse engineering Java software systems," *Software: Practice and Experience*, vol. 31, no. 4, pp. 371–394, 2001.

[16] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Transactions on Computers*, vol. 100, no. 6, pp. 592–597, 1972.

[17] J. S. Carr and B. T. Kachmarck, "Generating module stubs," Aug. 25 2015, uS Patent 9,117,177.

[18] S. Jayaraman, B. Jayaraman *et al.*, "Towards program execution summarization: Deriving state diagrams from sequence diagrams," in *Seventh International Conference on Contemporary Computing (IC3), 2014*. IEEE, 2014, pp. 299–305.

[19] R. Oechsle and T. Schmitt, "JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java debug interface (JDI)," in *Software Visualization*. Springer, 2002, pp. 176–190.

[20] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst, "Continuous monitoring of software services: Design and application of the Kieker framework," in *Technical Reports by Department of Computer Science*. Kiel University, Germany, 2009.

[21] Y. Watanabe, T. Ishio, Y. Ito, and K. Inoue, "Visualizing an execution trace as a compact sequence diagram using dominance algorithms," *Program Comprehension through Dynamic Analysis*, p. 1, 2008.

[22] T. Ishio, Y. Watanabe, and K. Inoue, "AMIDA: A sequence diagram extraction toolkit supporting automatic phase detection," in *Companion of the 30th International Conference on Software Engineering*. ACM, 2008, pp. 969–970.

[23] T. Ziadi, M. A. A. Da Silva, L.-M. Hillah, and M. Ziane, "A fully dynamic approach to the reverse engineering of UML sequence diagrams," in *16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), 2011*. IEEE, 2011, pp. 107–116.

[24] Y. Labiche, B. Kolbah, and H. Mehrfard, "Combining Static and Dynamic Analyses to Reverse-Engineer Scenario Diagrams," in *29th IEEE International Conference on Software Maintenance (ICSM), 2013*. IEEE, 2013, pp. 130–139.

[25] S. Lamprier, N. Baskiotis, T. Ziadi, and L.-M. Hillah, "CARE: a platform for reliable Comparison and Analysis of Reverse-Engineering techniques," in *18th International Conference on Engineering of Complex Computer Systems (ICECCS), 2013*. IEEE, 2013, pp. 252–255.

[26] M. Srinivasan, J. Yang, and Y. Lee, "Case studies of optimized sequence diagram for program comprehension," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–4.

[27] M. H. Alalfi, J. R. Cordy, and T. R. Dean, "Automated reverse engineering of UML sequence diagrams for dynamic web applications," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW'09), 2009*. IEEE, 2009, pp. 287–294.

[28] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, "Understanding JavaScript event-based interactions," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 367–377.

[29] D. Amalfitano, A. R. Fasolino, A. Polcaro, and P. Tramontana, "The DynaRIA tool for the comprehension of Ajax web applications by dynamic analysis," *Innovations in Systems and Software Engineering*, vol. 10, no. 1, pp. 41–57, 2014.

[30] V. Dehlen and J. Ø. Aagedal, "A uml profile for modeling mobile information systems," in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2007, pp. 296–308.

[31] F. A. Kraemer, "Engineering android applications based on uml activities," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 183–197.

[32] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1036–1046.

[33] E. Kowalczyk, "Modeling app behavior from multiple artifacts," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 385–386.

[34] M. S. Andreasen, H. V. Nielsen, S. O. Schrøder, and J. Stage, "Usability in open source software development: Opinions and practice," *Information technology and control*, vol. 35, no. 3, 2015.

[35] J. Feller and B. Fitzgerald, "A framework analysis of the open source software development paradigm," in *Proceedings of the twenty first international conference on Information systems*. Association for Information Systems, 2000, pp. 58–69.

[36] Y. Zhou and J. Davis, "Open source software reliability model: an empirical approach," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4. ACM, 2005, pp. 1–6.

[37] A. Bonaccorsi and C. Rossi, "Why open source software can succeed," *Research policy*, vol. 32, no. 7, pp. 1243–1258, 2003.

[38] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue, "Extracting sequence diagram from execution trace of java program," in *Principles of Software Evolution, Eighth International Workshop on*. IEEE, 2005, pp. 148–151.

[39] K. Koskimies and H. Mossenbock, "Scene: Using scenario diagrams and active text for illustrating object-oriented programs," in *Proceedings of the 18th International Conference on Software Engineering, 1996*. IEEE, 1996, pp. 366–375.

[40] H. Mössenböck and N. Wirth, "The programming language Oberon-2," *Structured Programming*, vol. 12, no. 4, pp. 179–196, 1991.

[41] T. Souder, S. Mancoridis, and M. Salah, "Form: A framework for creating views of program executions," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. IEEE Computer Society, 2001, p. 612.

[42] J. Jiang, J. Koskinen, A. Ruokonen, and T. Systa, "Constructing usage scenarios for API redocumentation," in *15th IEEE International Conference on Program Comprehension (ICPC'07), 2007*. IEEE, 2007, pp. 259–264.

[43] E. Mäkinen and T. Systä, "MAS—an interactive synthesizer to support behavioral modelling in UML," in *Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society, 2001, pp. 15–24.

[44] T. Systa, "On the relationships between static and dynamic models in reverse engineering java software," in *Proceedings. Sixth Working Conference on Reverse Engineering, 1999*. IEEE, 1999, pp. 304–313.

[45] H. A. Müller, S. R. Tilley, and K. Wong, "Understanding software systems using reverse engineering technology perspectives from the rigi project," in *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*. IBM Press, 1993, pp. 217–226.

[46] K. Koskimies, T. Systa, J. Tuomi, and T. Mannisto, "Automated support for modeling oo software," *IEEE software*, vol. 15, no. 1, pp. 87–94, 1998.

[47] H. B. Lee and B. G. Zorn, "Bit: A tool for instrumenting java bytecodes." in *USENIX Symposium on Internet technologies and Systems*, 1997, pp. 73–82.

[48] M. Dahm, "Byte code engineering," in *JIT99*. Springer, 1999, pp. 267–277.

[49] R. K. Yin, *Case study research: Design and methods*. Sage publications, 2013.

[50] B. Cornelissen, A. Zaidman, and A. van Deursen, "A controlled experiment for program comprehension through trace visualization," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 341–355, 2011.

[51] F. Fittkau, S. Finke, W. Hasselbring, and J. Waller, "Comparing trace visualizations for program comprehension through controlled experiments," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. IEEE Press, 2015, pp. 266–276.