Journal of Theoretical and Applied Information Technology

30th November 2023. Vol.101. No 22

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

TOWARDS A HYBRID APPROACH TO REVERSE ENGINEER BEHAVIORAL UML DIAGRAMS FROM SOURCE CODE

HAMZA ABDELMALEK[1], ISMAÏL KHRISS[2], AND ABDESLAM JAKIMI[3]

[1,3]GLISI team, FSTE, Moulay Ismail University Meknès, Errachidia, Morocco

[2]Département de Mathématiques, d'Informatique et de Génie, Université du Québec à Rimouski, Rimouski, Canada

E-mail: [1]h.abdelmalek@edu.umi.ac.ma, [2]ismail_khriss@uqar.ca, [3]ajakimi@yahoo.fr

ABSTRACT

Software reverse engineering plays an important role when maintaining legacy systems, enabling understanding of a system by extracting high-level models from its source code. These models can represent the structural or behavioral aspects of the system. Several approaches have been proposed in the literature for recovering structural models, such as the Unified Modeling Language (UML) class diagram. Conversely, there is less work concerning extracting behavioral representations that capture different interactions within a given system. This paper investigates approaches to extracting behavioral UML diagrams, precisely sequence and use case diagrams. We have categorized these approaches into three groups, depending on the type of analysis employed: static, dynamic, or hybrid. Subsequently, we conducted a comparative analysis of these approaches, evaluating them based on various criteria to highlight their strengths and weaknesses. Based on this comparison, we propose an approach that combines static and dynamic analysis techniques to recover behavioral diagrams from source code. This proposed approach can potentially assist software developers in maintenance by providing a higher-level representation of a system that can even be employed in a modernization process to migrate it from a legacy environment to a modern one.

Keywords: *Reverse Engineering, Modernization Process, Behavioral model, UML Sequence Diagram, UML Use Case Diagram.*

1. INTRODUCTION

Software development is a systematic process where each phase builds upon the previous one. Poor choices at any stage can lead to issues later on. The phases in the software development process include requirements analysis, design, implementation, testing, and maintenance. The phases from software requirements to implementation are called forward engineering [1]. Several artifacts, such as documentation and source code, are created throughout the process. These artifacts are valuable for developers to aid in future maintenance tasks.

During the maintenance phase, developers must update a system for different reasons, such as correcting bugs or changes in requirements. Unfortunately, the documentation no longer reflects what is in the source code, as changes are only done in the source code and not in other artifacts. Which greatly complicates the maintenance process as it becomes difficult to understand the system. That is why reverse engineering, one of the main approaches to software comprehension, becomes necessary.

According to Chikofsky et al., reverse engineering involves a two-step process. The first step involves analyzing an existing system to identify its components and interrelationships. The second step consists of creating representations of the system at a higher level of abstraction, relying on the information obtained in the first step [1]. The primary artifacts used in the analysis step are source code and execution traces. Therefore, three types of analysis are used in the literature: static, dynamic, or hybrid. Different modeling languages are used to represent the higher level of abstraction. One of the most used is the Unified Modeling Language (UML) with its structural diagrams, such as class diagrams, or behavioral diagrams, such as use case diagrams and sequence diagrams.

This paper discusses a set of approaches focusing on extracting UML behavioral diagrams. We categorize these approaches based on the type of analysis. First, we present the static analysis approach, which analyzes the software without executing it. Instead, it analyzes artifacts such as source code or byte code. Static analyses are effective in recovering the system's overall structure and extracting details about the software's components, and they don't require modifying the source code. However, it can be challenging to identify certain dynamic aspects like polymorphism using static analysis, and it demands significant processing time when analyzing larger systems. Therefore, static analysis is better suited for simpler systems.

In the second part of the related work, we delve into approaches that leverage dynamic analysis. These approaches require the instrumentation of the software, followed by its execution, during which execution traces are generated. These traces are analyzed to identify dynamic information such as invoked operations and object interactions. This type of analysis extracts the behavior of the systems that can be captured in dynamic UML diagrams like sequence and use case diagrams. Nevertheless, it is important to note that instrumentation can significantly slow down the software [2], and ensuring that the scenario information accurately reflects in the execution traces can be challenging.

Alternatively, some approaches adopt hybrid techniques combining static and dynamic analysis. In these cases, both the source code and the execution traces are analyzed. Hybrid analysis proves beneficial because it allows us to gather information in two modes: before and after running the software, thus providing rich and various data. The precision offered by dynamic analysis and the generalization provided by static analysis complement each other effectively, resulting in improved results [3]. Hybrid analysis serves the purpose of minimizing the need for extensive instrumentation of the source code. It accomplishes this by selectively extracting important data using dynamic analysis and then enhancing it by incorporating static analysis. This analysis enables us to extract maximum helpful information from the software, yielding higher abstraction in our results.

The investigation and comparison of these approaches allow us to propose a complete methodology aimed at reverse engineering sequence and use case diagrams from source code. Our proposed approach uses a combination of static and dynamic analysis. Dynamic analysis is applied first, involving the instrumentation of the source code and the generation of execution traces based on important scenarios. Subsequently, static analysis is employed to enhance these execution traces with additional information. To further refine the results, we apply other operations to the execution traces. The first operation is trace reduction, which filters irrelevant data, such as implementation details. Next, we propose merging multiple traces representing different scenarios for a given use case. The final step in our approach is straightforward, allowing the generation of use case and sequence diagrams from the improved execution traces.

aligning with our ongoing modernization approach following the Model Driven Engineering (MDE) paradigm, this research establishes a systematic methodology for extracting behavioral UML diagrams through a combination of static and dynamic analyses. Building on previous work targeting the reverse engineering of the static aspect of a system, namely the UML class diagram for the problem domain [4] and the Platform Description Model (PDM) for the solution domain [5, 6], this study aims to facilitate the understanding of both static and dynamic aspects of software systems. Furthermore, the extracted models from the reverse engineering task can be employed in generating modern systems for new implementation platforms and architectures through a forward engineering process [7].

The structure of this paper is as follows. Section 2 presents related work regarding reverse engineering behavioral diagrams. Section 3 is dedicated to a comparative analysis of the approaches in this field. In section 4, we introduce our reverse engineering approach. Finally, section 5 concludes the paper.

2. RELATED WORK

The addressed problem in this research revolves around the challenges faced during the maintenance phase of software development, particularly in scenarios where the documentation no longer accurately reflects the changes made in the source code. This problem hinders the comprehension of the system, making maintenance a complex task. The specific problem to be addressed is the need for effective reverse engineering approaches, specifically focusing on the extraction of UML behavioral diagrams, to enhance understanding during the maintenance phase.

The literature screening criteria encompass a targeted selection of studies addressing the challenges in software maintenance, specifically focusing on reverse engineering methodologies for extracting UML behavioral diagrams. Emphasis is placed on the analysis approaches, including static, dynamic, and hybrid methods, with a consideration of their effectiveness in diverse software environments. Additionally, the impact of instrumentation on software performance, the advantages and limitations of each analysis approach. The objective is to assemble a cohesive body of literature that not only addresses the identified problem of differences between documentation and source code during maintenance but also contributes to the formulation of a comprehensive and efficient reverse engineering approach for extracting behavioral diagrams that serve other purposes such as software modernization.

2.1 Static Approaches

Fauzi et al. used Abstract Syntax Tree (AST) to reverse engineer sequence diagrams of a system [8]. They implemented their approach in a tool called RE-VUML, which extracts and illustrates important information within the sequence diagrams. This information includes method calls, loops, conditional statements, class/object types, object creation, package, import statements, and object-oriented concepts like inheritance and static polymorphism. The process begins by extracting the AST from the source code using the JavaParser API. Next, they trace all the AST nodes using the Depth-First Search (DFS) Post Order algorithm. This traversal algorithm evaluates nodes sequentially following the source code. Finally, they use the PlantUML API[1] to create the sequence diagram.

Nanthaamornphong and Leatongkam extended the ForUML tool [9] to recover sequence diagrams from the source code of object-oriented Fortran applications [10]. They begin by parsing the source code into smaller parts and then discovering relationships between them using transformation rules. These transformation rules are based on UML specifications to map the source code into sequence diagram elements. Later, from the derived relationships, an XMI file is created and imported into the Modelio tool[2] to visualize the diagram.

Alvin et al. implemented a tool called StaticGen, which extracts sequence diagrams from source code using static analysis [11]. The first step concerns transforming the source code into a typed Control Flow Graph (CFG). The subsequent step involves the construction of a directed code hypergraph, which captures additional information, including interactions between objects within the code. The final step concerns creating the sequence diagrams. In this stage, the user employs a query-based refinement interface to navigate the hypergraph and extract important interactions present in the source code.

2.2 Dynamic Approaches

Delamare et al. are inspired by the work presented in [12]. They generate basic sequence diagrams from execution traces and combine them into a single sequence diagram while identifying fragments such as loop and alt [13]. The difference between these two approaches lies in the manner of combination. The contribution of Delamare et al. is their approach's ability to capture the program's state both before and after each message within the basic sequence diagrams. This ability enables them to identify iterations and conditional statements within the diagrams.

Li et al. introduced an approach to construct use case diagrams based on execution traces [14]. They leverage call trees derived from execution traces to retrieve the initial operations that present the call tree's root. These root operations represent the basic use cases, which are then arranged into a single sequence. Subsequently, they apply an algorithm to establish relationships between these basic use cases, yielding a composite use case diagram.

Dugerdil and Repond proposed an approach to extract the sequence diagrams from legacy systems to facilitate software understanding [15]. Their process commences by recovering the use cases from the software's users. Then, they instrument the source code to generate execution traces for specific scenarios. Based on dynamic information, they construct clusters of classes where each cluster represents a set of strongly interconnected classes that implement common business logic. To achieve more abstraction, they added two reduction techniques to the execution trace. They eliminate accessor methods and compress repeated events. They performed a bottom-up approach to identify event repetition, systematically replacing repeated events with a special node. This node will later be transformed into a loop fragment within the sequence diagram.

[1] https://plantuml.com/api
[2] https://www.modelio.org

Grati et al. employed interactive visualization as a part of a semi-automatic approach to reverse engineer sequence diagrams from execution traces [16]. The first part of their approach concerns the instrumentation of source code to generate execution traces from the specified scenarios. Each scenario may contain different alternatives, resulting in distinct behaviors and trace generation for each case. To align two execution traces, they are structured as trees where each node corresponds to a method call. Then, the trees are compared node by node. This task is implemented based on the Smith-Waterman algorithm [17]. In the second part, the approach involves visualizing the execution traces through an interactive environment. The interactive visualization application contains two components. The first component facilitates the visualization of the execution traces using geometric shapes. The second component is used to generate the corresponding sequence diagram and provides users with suggestions to enhance the process.

Ziadi et al. introduced a dynamic approach to reverse engineer UML sequence diagrams from multiple execution traces [18]. The initial phase involves the collection of traces based on various scenarios to capture the overall system behavior. In the subsequent step, each trace is represented using a Labeled Transition System (LTS), where each method invocation within the trace corresponds to a transition between two states in the LTS. They merge all the LTSes using the k-tail algorithm [19]. This algorithm combines two states if they share the same path length of method invocations. Finally, the resulting LTS is presented as a regular expression to simplify the conversion into a sequence diagram.

Sarkar and Chaterjee employ dynamic analysis to extract sequence diagrams from Java applications [20]. Initially, they instrument the source code by identifying all classes, objects, and functions. Then, they insert two methods, *begin* and *end* at the start and end of each function. The begin method contains parameters to indicate the origin and destination of the function call. During the system's execution, when a method is invoked, it triggers the *begin* and *end* functions. These functions, in turn, call another function called *WritePicFile* which constructs a file in the .pic format. This file is used to generate the sequence diagram.

Hammad and Al-Hawawreh proposed an approach for generating sequence diagrams and call graphs from execution traces [21]. The process begins by defining the target methods that require instrumentation. Subsequently, a Classes/Objects finder is used to locate the suitable positions for instrumentation within the source code. Instrumenting the source code involves identifying the start and end points of the target methods and inserting a monitoring function at the appropriate locations. This monitoring function includes two parameters: the first parameter specifies the method's name, while the second parameter identifies whether it's the method's beginning or ending. In the second phase, the system is executed to generate execution traces, and finally, these traces are utilized to construct the sequence diagrams and call graphs.

2.3 Hybrid Approaches

Guéhéneuc and Ziadi proposed an approach for reverse engineering UML sequence and state machine diagrams using dynamic and static analysis techniques [12]. This work aims to perform high-level analyses like conformance checking and pattern identification. The first step of the approach is to generate the execution traces from the JAVA program using the Caffeine tool [22]. To ensure that the execution trace contains sufficient information for constructing the sequence diagram, the same scenario is executed multiple times with different inputs. The second step involves creating basic sequence diagrams from the generated traces, followed by their combination using fragments like loop, alt, seq, and par. Finally, they use the approach proposed by [23] to generate the state machine diagram from the generated sequence diagram.

Dugerdil and Jossi introduced a technique to recover complete use cases when those initially provided by system users are incomplete and inaccurate [24]. Their approach involves instrumenting the legacy system's source code. Then, executing user scenarios are executed to generate execution traces, which in turn are analyzed to extract the executed operations. The next step is to analyze the source code AST using the visitor design pattern [25] to identify conditional statements for each method. Any statement found represents an alternative flow, thus an alternative behavior. All the alternatives are found using the backward slicing technique [26] of the AST corresponding to the source code of the operation. If the users confirm that the alternative statement could be executed, an extra step is added to the scenario under study. Therefore, a new execution trace for the scenario is generated. This step is repeated until obtaining a

final trace for the scenario with the different alternatives.

In [27], the authors adopt a similar methodology to that in [24] but with notable enhancements. They introduce a novel execution trace format and employ a dynamic decision tree compression technique. The process is initiated by documenting the main use cases of the system based on user experience, resulting in one scenario per use case. After source code instrumentation, they execute the system according to the initial scenario and capture the generated execution trace. The new execution trace format enables the identification of conditional statements, potentially revealing alternative scenarios within the same use case. Next, the execution trace is analyzed to find conditional and control statements, which are then structured as a decision tree and subsequently reduced. Following this, the reduced tree is analyzed to identify control statements that users could alter via actions on the user interface. Source code analysis aids in identifying the actions leading to various variants of the initial scenario. Lastly, for each valid variant uncovered, a new scenario is designed to generate a new execution trace, and this entire process is repeated as necessary.

Labiche et al. proposed combining static and dynamic analyses to reverse engineer scenario diagrams [2]. The hybrid analysis aims to reduce the instrumentation and avoid affecting the program's behavior. In dynamic analyses, they instrument the source code with Aspects [28] to gather essential information without affecting the program's behavior. In addition, they perform static analysis to extract further information, such as the method and class associated with a particular call and whether the call is located within a conditional statement or loop. Consequently, they obtain execution traces and a Control Flow Graph (CFG) from the dynamic and static analyses. These are transformed into scenario diagrams using a model transformation.

3. COMPARATIVE STUDY

We conduct a comparative analysis based on several criteria to evaluate the presented approaches. These criteria include the type of analysis employed, the target UML diagrams, the supported systems, the supported programming languages, the level of abstraction they achieve, the degree of automation integrated into the process, the instrumentation, and the technique used in each approach. Table 1 presents the comparison of the approaches.

We distinguish between static, dynamic, and hybrid approaches in the analysis type. The target UML diagrams are primarily sequence and use case diagrams, specifically focusing on whether sequence diagrams respect the UML 2.0 standard use of fragments. The proposed approaches can support specific systems such as object-oriented, legacy, or procedural systems. Furthermore, we have added the supported programming languages.

The critical objective of software reverse engineering is to generate abstract representations for the users. Some approaches present the business aspect of the software by eliminating the implementation details to give useful information. We also explore the presence of trace reduction techniques, which augment abstraction and enhance diagram readability. Finally, we provide the degree of automation within these approaches, whether they necessitate user intervention or operate automatically.

4. OVERVIEW OF APPROACH

The complexity of extracting behavioral UML diagrams at a high abstraction level demands the application of several techniques and analyses. Therefore, we combined static and dynamic analysis to extract sequence and use case diagrams. Our approach begins with dynamic analysis, enabling extracting information tied to specific scenarios, which is subsequently enriched through static analysis. Additionally, we introduce two techniques to produce concise and valuable execution traces: trace reduction and merging. Figure 1 presents our approach overview and the techniques employed in each step.

The first step in dynamic analysis is instrumentation, a technique to insert code pieces into an existing program. This technique is valuable when monitoring a software's performance and gaining insights into its runtime behavior. During this phase, we instrument a system's source code or bytecode to generate dynamic information. Reverse engineering sequence or use case diagrams from a given source code requires locating the executed elements when running a scenario such as invocated operations. Source code instrumentation proves straightforward and is often favored when we can access the source code. In contrast, bytecode instrumentation becomes necessary when the source code is unavailable.

Following the instrumentation, the next step involves running the system depending on the desired scenarios. This phase benefits significantly when essential documentation is accessible or when the system's users are actively engaged. Their experience allows us to find the system's use cases and scenarios in such cases. However, without these resources, the task becomes notably more challenging. In such scenarios, identifying the components that require execution to extract the desired diagrams becomes less straightforward.

Table 1: Comparison Of The Approaches.*

Reference	Analysis Type	Target UML diagram	Supported system	Supported language	Abstraction			Other criteria		
					Level	Business logic	Trace reduction	Automation	Instrumentation	Technique used
[2]	H	SD	OO	Java	+-	-	-	F	Aspects	CFG
[12]	H	SD2.0	OO	Java	+-	-	-	F	Caffeine	-
[13]	D	SD2.0	OO	Java	--	-	-	F	JTracor	-
[14]	D	UCD	OO	C++	+-	-	-	F	Reflection	Call tree
[15]	D	SD2.0	OO/L	Java	++	+	+	S	JavaCC	Clustering
[16]	D	SD	OO	Java	++	-	+	S	-	Interactive visualization
[18]	D	SD2.0	OO/L	Java	+-	-	+	F	Customized debugger	Labeled transition system
[20]	D	SD	OO	Java	+-	-	-	F	-	Pic language
[21]	D	SD	OO	Java	--	-	-	F	Monitoring function	-
[8]	S	SD2.0	OO	Java	+-	-	-	F	Java Parser	AST
[10]	S	SD2.0	OO	Fortran	+-	-	-	F	-	Transformation rules
[11]	S	SD	OO	Android	+-	-	-	S	-	CFG
[24]	H	UCD	OO/L	Java	++	+	-	S	JavaCC	AST
[27]	H	UCD	OO/L	Java	++	+	-	S	The code instrumentor	Dynamic decision tree

*: +: yes, presented, or supported / -: no, not presented, or unsupported.
Analysis Type: *S*: static, *D*: dynamic, *H*: hybrid.
Target UML diagram: *SD*: sequence diagram, *SD2.0*: sequence diagram with UML2.0 standard, *UCD*: use case diagram.
Supported system: *OO*: object-oriented systems, *L*: legacy systems.
Abstraction level: --: low abstraction, +-: medium abstraction, ++: high abstraction.
Automation: *F*: fully automatic, *S*: semi-automatic.

Once the system is executed, the results of the scenarios are reflected in the execution traces. Each scenario gives a single execution trace, giving a single sequence diagram. These execution traces are files containing a sequence of statements referred to as events. Each event serves as a representation of a method call and can be tailored to display pertinent information. For instance, it can include details such as the class, method, and package associated with the call's origin.

The previous steps are part of dynamic analysis, as they involve collecting information about the system at runtime. On the other hand, static analysis collects information offline, acquiring information by examining the software's source code. Static analysis serves the purpose of enhancing the execution traces with additional data. The common technique to analyze source code is parsing it into an AST and analyzing it sequentially to extract the needed information.

The initial execution trace typically contains an important number of events. Some useless events, such as implementation details and repeated events, affect the abstraction level. Besides abstraction issues, analyzing these traces becomes difficult because of the massive and unstructured data. In addition, the resulting abstraction level remains quite low when generating sequence diagrams from these initial traces. We must reduce traces by removing unnecessary events and data to enhance this abstraction. Furthermore, combining traces originating from scenarios belonging to the same use case or business logic becomes advantageous. Combining traces can be accomplished through trace merging, which can follow a pairwise approach, merging two execution traces simultaneously using bioinformatics sequence alignment algorithms [16] or employing advanced techniques such as machine learning for multiple trace alignments and reduction, enhancing the overall trace abstraction[29, 30].

In previous work [4], we proposed the separation of platform-independent concepts from platform-specific concepts within the architecture-driven modernization (ADM) process [31]. This separation results in the extraction of the plat-form-independent model (PIM) expressed in the UML class diagram, which presents the static aspect of the system. They operated under the hypothesis that platform-related concepts tend to be repetitive or semi-repetitive in the source code. We can apply a similar approach to extract important information from execution traces and create abstract behavioral UML diagrams that represent the business logic of the systems.

The final step in this approach is generating the UML diagrams based on the refined execution traces.

5. CONCLUSION AND FUTUR WORK

This paper reviewed approaches to recover sequence and use case diagrams from source code through reverse engineering techniques. We categorized the approaches based on their analysis type, which could be static, dynamic, or a combination of both. By examining and comparing these existing techniques, we've identified an appropriate approach for reverse engineering diagrams from source code while maintaining a high level of abstraction. Our proposed approach effectively combines the capabilities of both static and dynamic analyses.

Despite the considerable research dedicated to recovering high-level models, the focus on the behavioral aspect remains moderate compared to the attention given to the static aspect. The outcomes often suffer from a lack of robust standards guiding the process, leading to results that may lack clarity and abstractness. As presented, model-driven approaches hold promise in overcoming these challenges by abstracting the process through models and leveraging established standards such as the ADM proposed by the Object Management Group (OMG).

One challenge in our approach is addressing scalability, given its involvement in multiple intermediate steps that require proper standardization efforts, including aspects such as instrumentation and specifying the format of the execution trace. Therefore, the first step is to suggest a suitable instrumentation technique that can be applied to any given system, regardless of the implementation platform, and to specify the format of the execution trace based on well-established work such as [32]. Alternatively, it should be easy to customize to support multiple programming languages, such as using the instrumentation of assembled object code [33]. A platform-independent instrumentation tool is necessary because, in our approach, we plan to reverse engineer behavioral UML diagrams for multiple implementation platforms. Using different tools for each programming language would be a complex and impractical task.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

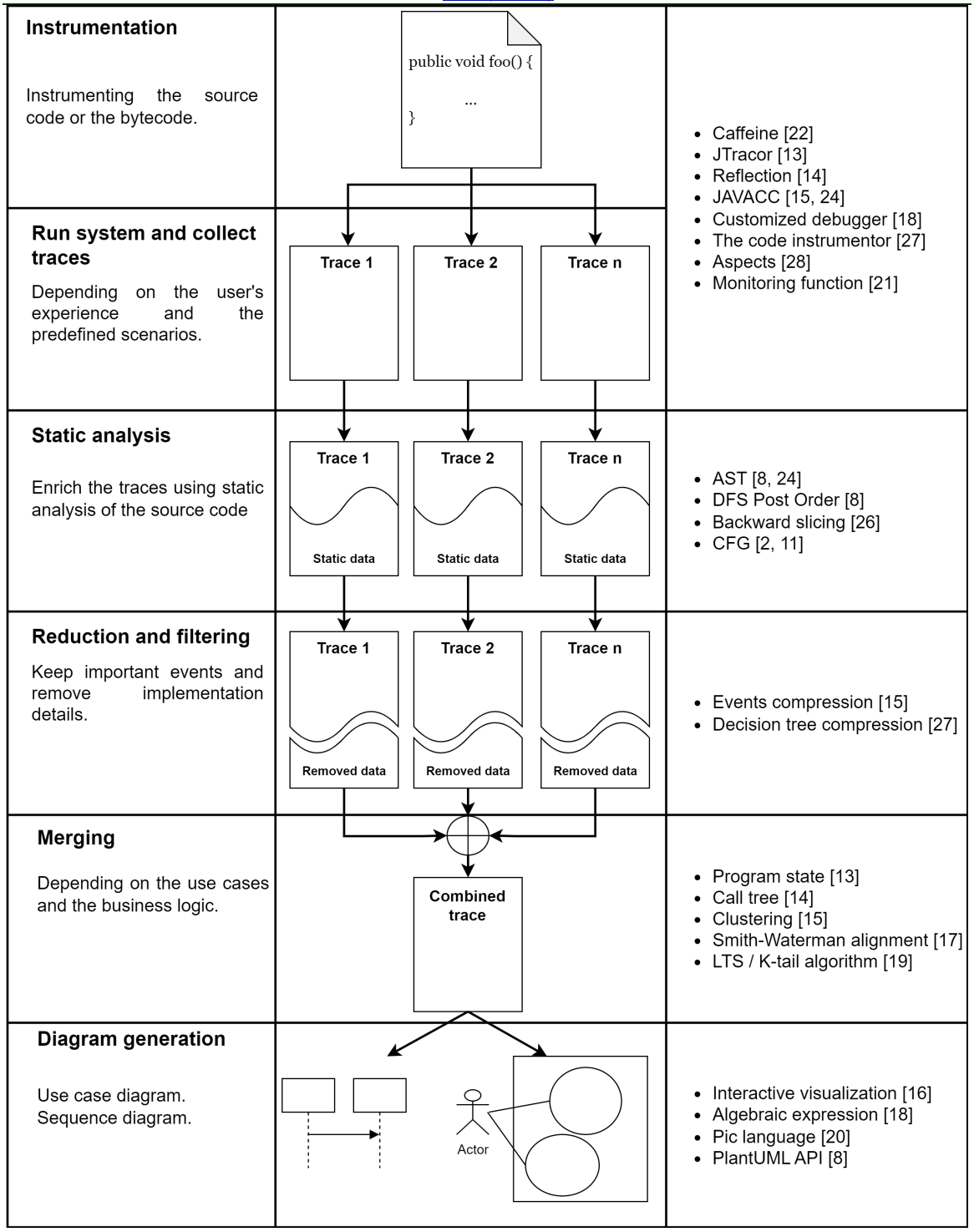

Figure 1: The Overview of Our Reverse Engineering Approach.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

REFERENCES:

[1] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE software,* vol. 7, no. 1, pp. 13-17, 1990.

[2] Y. Labiche, B. Kolbah, and H. Mehrfard, "Combining static and dynamic analyses to reverse-engineer scenario diagrams," in *2013 IEEE International Conference on Software Maintenance*, 2013: IEEE, pp. 130-139.

[3] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, 2003, pp. 24-27.

[4] I. Khriss and G. Chénard, "Automatic Discovery of Platform Independent Models of Legacy Object-Oriented Systems," in *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, 2016: The Steering Committee of The World Congress in Computer Science, Computer …, p. 3.

[5] G. Chénard, I. Khriss, and A. Salah, "Towards the automatic discovery of platform transformation templates of legacy object-oriented systems," in *Proceedings of the 6th International Workshop on Models and Evolution*, 2012, pp. 51-56.

[6] H. Abdelmalek, G. Chénard, I. Khriss, and A. Jakimi, "A Bimodal Approach for the Discovery of a View of the Implementation Platform of Legacy Object-Oriented Systems under Modernization Process," in *CATA*, 2020, pp. 98-111.

[7] I. Khriss, A. Jakimi, and H. Abdelmalek, "Towards an Effective Implementation of a Model-Driven Engineering Approach for Software Development," in *2020 1st International Conference on Innovative Research in Applied Science, Engineering and Technology (IRASET)*, 2020: IEEE, pp. 1-6.

[8] E. Fauzi, B. Hendradjaya, and W. D. Sunindyo, "Reverse engineering of source code to sequence diagram using abstract syntax tree," in *2016 International Conference on Data and Software Engineering (ICoDSE)*, 2016: IEEE, pp. 1-6.

[9] A. Nanthaamornphong, K. Morris, and S. Filippone, "Extracting uml class diagrams from object-oriented fortran: Foruml," in *Proceedings of the 1st International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, 2013, pp. 9-16.

[10] A. Nanthaamornphong and A. Leatongkam, "Extended ForUML for automatic generation of UML sequence diagrams from object-oriented Fortran," *Scientific Programming,* vol. 2019, 2019.

[11] C. Alvin, B. Peterson, and S. Mukhopadhyay, "Static generation of UML sequence diagrams," *International Journal on Software Tools for Technology Transfer,* vol. 23, pp. 31-53, 2021.

[12] Y.-G. Guéhéneuc and T. Ziadi, "Automated reverse-engineering of UML v2. 0 dynamic models," *WS Proc. ECOOP,* vol. 5, 2005.

[13] R. Delamare, B. Baudry, and Y. Le Traon, "Reverse-engineering of UML 2.0 sequence diagrams from execution traces," in *Workshop on Object-Oriented Reengineering at {ECOOP 06}*, 2006.

[14] Q. Li, S. Hu, P. Chen, L. Wu, and W. Chen, "Discovering and mining use case model in reverse engineering," in *Fourth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2007)*, 2007, vol. 4: IEEE, pp. 431-436.

[15] P. Dugerdil and J. Repond, "Automatic generation of abstract views for legacy software comprehension," in *Proceedings of the 3rd India software engineering conference*, 2010, pp. 23-32.

[16] H. Grati, H. Sahraoui, and P. Poulin, "Extracting sequence diagrams from execution traces using interactive visualization," in *2010 17th Working Conference on Reverse Engineering*, 2010: IEEE, pp. 87-96.

[17] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology,* vol. 147, no. 1, pp. 195-197, 1981.

[18] 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 *2011 16th IEEE International Conference on Engineering of Complex Computer Systems*, 2011: IEEE, pp. 107-116.

[19] 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.

[20] M. K. Sarkar and T. Chaterjee, "Reverse engineering: An analysis of dynamic behavior

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

of object oriented programs by extracting UML interaction diagram," *International Journal of Computer Technology and Applications,* vol. 4, no. 3, p. 378, 2013.

[21] M. Hammad and M. Al-Hawawreh, "Generating sequence diagram and call graph using source code instrumentation," in *Information Technology-New Generations: 14th International Conference on Information Technology*, 2018: Springer, pp. 641-645.

[22] Y.-G. Guéhéneuc, R. Douence, and N. Jussien, "No Java without caffeine: A tool for dynamic analysis of Java programs," in *Proceedings 17th IEEE International Conference on Automated Software Engineering*, 2002: IEEE, pp. 117-126.

[23] T. Ziadi, L. Helouet, and J.-M. Jézéquel, "Revisiting statechart synthesis with an algebraic approach," in *Proceedings. 26th International Conference on Software Engineering*, 2004: IEEE, pp. 242-251.

[24] P. Dugerdil and P. Jossi, "A legacy systems use case recovery method," in *In: ICSOFT 2010: proceedings of the 5th International Conference on Software and Data Technologies, Athens, Greece, July 22-24. Setúbal: SciTePress, 2010, vol. 2, p. 232-237*, 2010.

[25] E. Gamma, R. Johnson, R. Helm, R. E. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[26] D. W. Binkley and K. B. Gallagher, "Program slicing," *Advances in computers,* vol. 43, pp. 1-50, 1996.

[27] P. Dugerdil and D. Sennhauser, "Dynamic decision tree for legacy use-case recovery," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013, pp. 1284-1291.

[28] J. D. Gradecki and N. Lesiecki, *Mastering AspectJ: aspect-oriented programming in Java*. John Wiley & Sons, 2003.

[29] T.-D. B. Le and D. Lo, "Deep specification mining," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 106-117.

[30] B. Afshinpour, R. Groz, M.-R. Amini, Y. Ledru, and C. Oriat, "Reducing Regression Test Suites using the Word2Vec Natural Language Processing Tool," in *SEED/NLPaSE@ APSEC*, 2020, pp. 43-53.

[31] OMG. (2007) Architecture-Driven Modernization Task Force. Available: https://www.omg.org/adm

[32] C. T. Pereira, L. I. Martínez, and L. M. Favre, "TRACEM-Towards a standard metamodel for execution traces in model-driven reverse engineering," in *XXVIII Congreso Argentino de Ciencias de la Computación (CACIC)(La Rioja, 3 al 6 de octubre de 2022)*, 2023.

[33] T. Kempf, K. Karuri, and L. Gao, "Software instrumentation," *Wiley encyclopedia of computer science and engineering,* pp. 1-11, 2007.