Program comprehension through reverse-engineered sequence diagrams: A systematic review

Article *in* Journal of Software: Evolution and Process · July 2018

DOI: 10.1002/smr.1965

3 authors:

Taher Ahmed Ghaleb
Queen's University
15 PUBLICATIONS **28** CITATIONS
SEE PROFILE

Musab Alturki
King Fahd University of Petroleum and Minerals
17 PUBLICATIONS **250** CITATIONS
SEE PROFILE

Khalid Aljasser
King Fahd University of Petroleum and Minerals
5 PUBLICATIONS **10** CITATIONS
SEE PROFILE

Received 00 Month 0000; | Revised 00 Month 0000; | Accepted 00 Month 0000

DOI: xxx/xxxx

RESEARCH ARTICLE

Program Comprehension through Reverse-engineered Sequence Diagrams: A Systematic Review

Taher Ahmed Ghaleb[1] | **Musab A. Alturki**[2] | **Khalid Aljasser**[2]

[1]School of Computing, Queen's University, Kingston, Canada
[2]Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia

Correspondence
Taher Ahmed Ghaleb, School of Computing, Queen's University, Kingston, Canada
Email: taher.ghaleb@queensu.ca

Author pre-print copy. The final publication is available at Wiley via:
https://doi.org/10.1002/smr.1965

Abstract

Reverse engineering of sequence diagrams refers to the process of extracting meaningful information about the behavior of software systems in the form of appropriately generated sequence diagrams. This process has become a practical method for retrieving the behavior of software systems, primarily those with inadequate documentation. Various approaches have been proposed in the literature to produce from a given system a series of interactions that can be used for different purposes. The reason for such diversity of approaches is the need to offer sequence diagrams that can cater for the users' specific goals and needs, which can vary widely depending on the users' perception and understandability of visual representations and the target application domains. In this paper, we systematically review existing techniques in this context while focusing on their distinct purposes and potentials for providing more understandable sequence diagrams. In addition, a qualitative evaluation of the studied techniques is conducted to expose their adequacy and applicability for effective program comprehension. After analyzing the state-of-the-art potentials for using reverse-engineered sequence diagrams, we finally propose several extensions to the UML sequence diagrams. Our proposed extensions are aimed to improve program understandability by enriching sequence diagrams with useful information about program control flow. Moreover, our extensions promote utilizing existing UML primitives to incorporate additional information about software behavior rather than introducing nonstandard tools.

KEYWORDS:
program analysis, program comprehension, software visualization, reverse engineering, reverse-engineered sequence diagram

1 | INTRODUCTION

Maintaining and improving (legacy) software systems that are scarcely or poorly documented has been a well-recognized challenge for software engineers. Software maintenance typically involves manually and carefully studying the source code to deduce the structural or behavioral design of such systems before performing any software maintenance task [1]. This daunting investigation is a tedious, highly ad-hoc and error-prone process, which, as a result, can significantly reduce the productivity of software engineers. Therefore, reverse engineering techniques have been introduced to facilitate the analysis of software components and automate the process of extracting information in a readable format.

In general, reverse engineering of software behavior is more challenging than that of software structure. In particular, existing techniques of analyzing software interactions utilize different methods for identifying software components and their behaviors. The overall process comprises a set of operations, namely parsing programs, tracing their interactions and visualizing resultant behaviors in manageable and understandable views. Object construction at runtime, method polymorphism, and other dynamic operations complicate this process further. The identification of such behaviors requires sophisticated techniques to analyze the source code and/or trace the program execution.

This paper conjoins a set of correlated concepts concerning program comprehension, including reverse engineering, program analysis, and software visualization. In the literature, surveys in this context were conducted to highlight and compare different kinds and characteristics of those techniques from different perspectives. However, there has been little focus on investigating how effective reverse-engineered sequence diagrams are for program comprehension. Indeed, the amount and kind of information presented in sequence diagrams determine how meaningful and understandable they are, and, hence, indicate how effective the technique is. Therefore, we endeavor in this work to comprehensively review all techniques performing reverse engineering of sequence diagrams as a means of visualizing software behavior. One main objective is to present the trade-offs among the different characteristics of the techniques developed for this purpose, especially when considering usability and scalability. A second main objective is to identify fundamental limitations of the UML sequence diagrams standard that can directly impact program comprehension. Ultimately, we believe that such a review sets the stage for future directions in this particular area of research in software engineering.

With that motivation in mind, an overview of the techniques that contribute to improving understandability of software interactions by means of sequence diagrams is introduced. First, we perform a critical review of related surveys in the literature and show how they differ from this review. Second, the state-of-the-art techniques are systematically collected and then classified based on the type of analysis they employ (i.e., static, dynamic, or hybrid program analysis). After that, techniques in this context are compared to each other according to a set of attributes that expose their effectiveness towards program comprehension. Finally, we draw a few recommendations from this discussion as sequence diagram extensions that can potentially enhance comprehensibility of software systems.

The rest of the paper is organized as follows. Section 2 qualitatively analyzes existing surveys in the context of reverse engineering and comprehension of software systems and compares them with our work in this paper. Then, Section 3 provides preliminaries on reverse engineering of sequence diagrams, program comprehension, program visualization, and the various program analysis techniques. Section 4 presents our systematic methodology for selecting and reviewing the relevant studies to the use of sequence diagrams for program comprehension. Section 5 defines a set of attributes we use for comparing the collected reverse engineering techniques, followed by a comparative analysis and discussion of major observations. Based on the results of this analysis, we present in Section 6 a list of potential extensions to the UML sequence diagram standard to capture additional particular aspects of program control flow. Section 7 highlights potential validity threats to our research methodology and data extraction process and how they were addressed. The paper finally concludes in Section 8 with a discussion of the findings and possible future work.

2 | RELATED WORK

There are several surveys and reviews in the literature that are relevant to what is conducted in this paper. Their relevance is basically expressed by what kind of techniques or methods they covered in their studies. In the context of this paper, we selected the surveys that focused on studying either program visualization, program analysis, reverse engineering, or program comprehension, from different perspectives. The purpose here is to show the motivation behind conducting our survey and how it is different from the existing ones in the literature.

A recent survey [2] in software visualization studied how this field of research is concerned with graphically presenting various aspects of software (e.g., structure, behavior, artifacts, and evolution) in an easy to understand fashion. [3] Mattila*et al.* [2] also presented various reviews in this area and how each review is distinguished from each other in terms of their focus and objective. We list in Table 1 all the relevant surveys in software visualizations, including the surveys presented in Mattila's study and other additional surveys. Each of such surveys studies the state-of-the-art techniques that improves a certain aspect of software system through software visualization.

We observe from Table 1 that most related surveys in software visualization focus on techniques or tools for visualizing non-behavioral aspects of software (such as architecture, reuse, performance and development process), except the one provided by Hamou *et al.*. [15] We, therefore, considered this survey as a starting point for looking for other surveys focusing on program comprehension of software behavior. After investigating all references of and citations to the study by HamouâĂŘLhadj and Lethbridge, [15] we collected and analyzed all its relevant surveys (listed in Table 2).

Table 2 shows that there is a limited set of surveys targeting visualization of software behavior, with the most recent one published in 2011. In addition, all such surveys focus their studies on techniques used for visualizing software interactions and behavior using various kinds of analysis techniques and visualization forms. However, they do not particularly pay attention to the effectiveness and limitations of sequence diagrams and how program comprehension could be achieved through them. Moreover, they lack an in-depth discussion of what information of software behavior is presented in the non-standard forms of visualization and how they could be supplied to sequence diagrams.

In particular, Pacione *et al.* [25] introduced a performance comparison of only 5 different techniques used for dynamic software visualization in addition to a reflexion model. A similar study was also introduced by Hamou-Lhadj and Lethbridge, [15] who discussed the benefits and drawbacks of trace exploration techniques showing how they work and how they could be improved in order to satisfy different maintenance activities. Although various techniques that use sequence diagrams as their main output were discussed in these two studies, no clues on how they could be generalized to cover different kinds of information about program control flow were given. Briand *et al.* [24] also reviewed program analysis techniques that were used for distributed and non-distributed systems showing how each technique works, what level of information it presents, and whether control flow

TABLE 1 Existing surveys/reviews in software visualization and reverse engineering

Survey/Review	Focus/Objective	Publication Year
[2]	Software visualization today	2016
[4]	Visualization of code clones	2015
[5]	Validation of visualization techniques	2014
[6]	Visualization of software performance	2014
[7]	Visualization for Software reuse	2014
[8]	Visualization for agile software development	2014
[9]	Visualization of software evolution	2013
[10]	Visualization of static software aspects	2011
[11]	Software reverse engineering goals and challenges	2011
[12]	Visualization of software architecture	2008
[13]	Requirements of software visualization tools	2007
[14]	Awareness of human activities in software development	2005
[15]	Visualization of software execution traces	2004
[16]	Visualization for software maintenance	2003
[17]	3D software visualizations.	2003
[18]	Users' perceptions about software visualization	2001
[1]	Reverse engineering tools for program understanding	2000
[19]	Visualization to support reverse engineering	1998

TABLE 2 Existing surveys/reviews in software behavior comprehension

Survey/Review	Focus/Objective	Publication Year
[20]	Validating dynamic program analysis techniques	2011
[21]	Applications of dynamic program analysis techniques	2009
[22]	Tool features for understanding sequence diagrams	2008
[23]	Design choices of SD reverse engineering tools	2006
[24]	Sequence diagrams for distributed systems	2006
[15]	Trace exploration techniques	2004
[25]	Dynamic visualization tools	2003

is maintained. The study by Briand *et al.*was the initial motivation for performing our study, since the number of criteria used for comparing different program analysis techniques in that study is relatively small, there are more recent studies in the area, and there is lack of focus on how sequence diagrams could generically be extended to capture as much useful information as possible about program behavior. Cornelissen [21] surveyed existing dynamic program analysis techniques, in which programs are monitored at runtime and traces of execution are collected to construct appropriate visualization diagrams. The main focus of this survey was to show existing methodologies and tools for that purpose along with their target applications. Another survey by Cornelissen *et al.*was also conducted to validate those dynamic analysis techniques over several case studies. [20] However, these two studies neglected the usefulness of static program analysis and how it can also play an important role in enriching reverse-engineered sequence diagrams. In addition, such studies lacked a discussion about how sequence diagrams compare with other visualization tools.

The evaluation of reverse engineering techniques that aim to improve program comprehension is usually carried out using controlled experiments. [26] However, there are a few studies that evaluated the sufficiency and suitability of reverse-engineered UML sequence diagrams (or extensions to them) for program comprehension. Surveys by Bennett *et al.* [22] and Merdes and Dorsch [23] are the only two surveys that focused their analyses on evaluating sequence diagrams. Such two surveys presented the experiences, design choices, and evaluation of visualization using reverse-engineered sequence diagrams. Moreover, they demonstrated the challenges and limitations of sequence diagrams and how can they be eliminated by making use of the features provided by tools that employ reverse-engineered sequence diagrams for displaying program interactions. They investigated the different features and facilities (e.g., searching, zooming, semantic zooming, etc.) that tools should provide in order to improve understandability of program behavior with the use of the existing notation of the UML sequence diagram. However, they did not address or study how to resolve current

weaknesses of the UML sequence diagram notation and how its current notation may be enriched with valuable information for improved program comprehension. They were also too specific to selected tools and techniques.

3 | REVERSE ENGINEERING OF SEQUENCE DIAGRAMS

Software reverse engineering is the practice of analyzing software artifacts to extract information and knowledge about software design and implementation and leverage it into software engineering processes. It is considered one of the most significant endeavors in software engineering that facilitate the recovery of system structure and behavior.[27] Reverse engineering is usually accomplished through a set of processes that typically include analyzing a program statically (e.g., based on the program source code) or dynamically (based on bytecode or program traces). Analyzing programs involves collecting relevant information about software behavior and then transforming the collected information into a higher-level, more abstract representation or model.[28] Such a representation can then be used for certain tasks, such as checking conformance of the system to its design specification documents.[29] Although the abstraction process usually requires human intervention so that certain decisions can be made (e.g., not all details about systems need to be derived), reverse engineering can significantly reduce the number of needed interactions.[23]

In most cases, reverse engineering techniques rely on heuristics, which may, subsequently, result in generating imprecise representations of program control flow of behavior. As there is no consensus on what precision means exactly in this context, we refer to the convention in which precision is used as a measure of the conformance of the resulting representation to the flow of control of programs.[30,31] Although it is important to be precise in visualizing programs, users may sometimes need to focus on understanding a specific aspect of the program. Therefore, hiding unimportant interactions from the produced diagrams can reduce their complexity, which can help users achieve their goals in a better way. Indeed, user participation in the refining process is needed[32] as user needs may differ from one user to another. Moreover, there may be trade-offs among the different characteristics of reverse engineering techniques, especially when considering performance, usability, and scalability. For example, producing complete, fine-grained models gives a comprehensive overview of system objects and interactions, but requires higher memory usage and may even impede comprehension by presenting too many details that can be distracting to some users. Furthermore, reverse engineering of sequence diagrams may face some scalability issues when dealing with large and complex systems. This is primarily caused by the level of details of information normally presented in sequence diagrams. Moreover, the presence of infeasible paths that might be present throughout the interprocedural control flows of programs can further complicate sequence diagram unnecessarily. The identification of such infeasible paths is known to be hard.[33]

3.1 | Program Comprehension

Program comprehension, or program understanding, is defined as the activity of identifying the different aspects of software systems to acquire knowledge about its structure and behavior.[34,35] This activity is important for various purposes, such as maintenance, inspection, extension, or reuse of existing software systems. Studies in the literature address program comprehension from different perspectives, such as theories, methods, tools, and the cognitive process.[36]

Challenges in understanding programs using sequence diagrams are concerned with the complexity of existing (legacy) systems and the amount of information they hide behind their implementation. These challenges are mostly faced when the source code of such systems is not available. Nevertheless, it is also the case even with the availability of source code, especially if no (or only inadequate) documentation exists. Therefore, research trends in this context have been diversified into different areas of software understanding, and the majority of them here focused on software visualization.[2]

Different approaches have been introduced in the literature to help in performing program comprehension in general. Top-down (e.g., Brooks's model[37]) and bottom-up (e.g., Soloway's model[38]) comprehension approaches are performed over source code where the former requires users to have a prior knowledge about the program, while in the later, users are expected to have no domain knowledge of it. In the top-down approach, expectations of the program are firstly built by users and then mapped to the source code. Users may also perform bottom-up comprehension by going through program source code and then building a high-level abstraction of its functionality. A combination of these two models may be required if users have a partial knowledge of program functionality.[39]

Apart from approaches that require the availability of source code and human participation, program analysis techniques, either static or dynamic, represent one of the key approaches that support program comprehension.[21,40] They facilitate the extraction of program components into readable and manageable formats. In the case of understanding software structure, static analysis is indeed sufficient, whereas it needs to be incorporated into the dynamic analysis in order to get an overall picture of program behavior. Ideally, software visualization represents the essential aid to increasing the level of program understandability.[2] Many program analysis techniques (most of them are tool-supported) have been introduced with the goal of enhancing program comprehension through the use of various styles of visualization representing software programs through the use of

either structure or behavior traces.[22] In this paper, we focus our attention on those techniques targeting the visualization of program behavior and interactions using sequence diagrams.

In general, techniques that aim to improve program comprehension are ideally evaluated using controlled experiments, which involve the preparation of a set of comprehension tasks that relate to comprehension activities,[26] such as maintenance activities, to be carried out by human subjects. The main objective of this kind of experiments is to measure the time spent by users to respond to the predefined comprehension tasks and evaluate the correctness of their responses. Such a method of evaluation has widely been used in different studies to evaluate their tools in comparison with other approaches.[20,41,42] The main challenge in conducting this kind of experiments is concerned with assembling a reasonable number of suitable subjects, which is not an easy task.[43] This is because the individual knowledge and experience of subjects in the area can impact the accuracy of the obtained results. Our main observation in this context is the absence of similar studies for evaluating the sufficiency and suitability of UML sequence diagrams for program comprehension. Several techniques commonly used sequence diagrams as the main tool of visual representation of program interactions, but not so many papers have evaluated its sufficiency for understanding program behavior. As an example, Bennett *et al.*[22] conducted an evaluation of the actual use of features provided by the tools that employ reverse-engineered sequence diagrams for representing program interactions, but the evaluation was based on personal judgments via different criteria. On the other hand, other studies evaluated how their resulting reverse-engineered diagrams improved program comprehension, but the evaluations were based a limited number of questions and participating users and were not conducted under a controlled environment.[44,45]

3.2 | Visualization of Program Behavior and Interactions

As the ultimate objective of reverse engineering is to express meaningful information to humans, considerable attention in the literature has been given to the generation and use of attractive visualizations.[21] Visualizing the program behavior is one of the challenges that can directly affect how meaningful the retrieved program information is. Visualizing all possible interactions of non-trivial programs (i.e., complex systems) can produce more complicated sequence diagrams, which are likely to exceed the cognitive abilities of human beings. On the other hand, hiding portions of the interactions of small programs (e.g. introductory student programs) from the produced sequence diagrams may result in losing important information that might have helped in understanding the programs thoroughly.

Reverse-engineered sequence diagrams actually differ from forward-engineered (i.e., human-designed) sequence diagrams. Sequence diagrams created in the design phase of system development convey abstract information about how the system should behave, with logical messages between objects and minimal technical details. On the other hand, sequence diagrams that are reverse-engineered using reverse engineering tools are more detailed and more representative about the actual behavior of the system. However, in terms of maintenance efficiency, they are almost similar as expressed by various people in a controlled experiment,[46] with little tendency towards the use of diagrams created during design as they are, as expressed by some users, easier to understand, in comparison with the reverse-engineered ones.

UML sequence diagrams have commonly been used in the literature as means of visualization of program behavior represented by the interactions between the different objects of software systems. Based on the Unified Modeling Language (UML) standard,[47] sequence diagrams can model the flow and interactions of a software system visually with the use of incoming and outgoing messages, which facilitate understanding and validating the program logic and control flow. They demonstrate essential insights of software behavior, which allows users to understand the functionality of existing systems for maintenance or testing purposes. In addition, since the UML sequence diagram is a *de facto* standard, it is normally preferred over other non-standard forms of visualization and enjoy more usability (due to increased interoperability and consistency of generated specifications) and is supported by a wider development community.

Nevertheless, UML sequence diagrams suffer from some general drawbacks that make them unsuitable in some situations. These drawbacks are mainly concerned with scalability and expressiveness issues when representing the different aspects of program behavior. First, program behavior is represented using a series of interactions that expand vertically, which can make it difficult for users to track interactions and to have an overall picture of the system behavior, especially when there is a considerable number of interactions. In addition, it is sometimes difficult to recognize the set of interactions that are executed inside a certain method, while this can easily be performed using activity diagrams. Furthermore, fragments are restricted and do not adapt well with some unstructured control flow constructs. For example, the `break` fragment defined by UML 2.0 exits from its immediate enclosing loop fragment, which makes it impossible to jump over a number of levels of nesting. This particular limitation was addressed by Rountev *et al.*[30] who proposed a generalized break fragment that enables exiting from a certain surrounding fragment. Finally, using lines-with-arrows for representing messages makes it almost impossible to represent compound calls that may involve other calls inside their parameters.

On the other hand, researchers tended to develop new (non-standard) methods of visualization of software behavior to overcome the limitations of sequence diagrams and fulfill specific needs of program comprehension that are not directly supported in the UML standard. For instance, Cornelissen *et al.*[48] proposed a way of representing program interactions using a circular bundle where an overall view of software behavior is presented, while Fittkau *et al.*[49] introduced the utilization of city metaphor to show the interactions between the different software entities. Other research efforts addressed the limitations of sequence diagrams by extending their current notation to capture the general control flow,[30] model security

patterns,[50] model thread creation, waiting and notification,[51] or show concurrent and distributed interactions.[24,33] Other techniques used State Chart Diagrams to display information about the various objects' states during their lifetime.[52,53] Rohr *et al.*,[33] on the other hand, used Markov Chains, Timing Diagrams, and Component Dependency Graphs in addition to sequence diagrams to visualize different behavior-related views of software. A combination of activity diagrams, class diagrams, and sequence diagrams was also used by Korshunova *et al.*[29] for representing program structure and behavior.

3.3 | Program Analysis

Program analysis methods that are used for the systematic retrieval of design and behavior of software systems can be either static, relying solely on the source code, or dynamic, where execution traces of the program are analyzed.[54] It is also common to combine both kinds of analysis within the same technique to be able to accomplish certain program comprehension tasks that would otherwise be hard or impossible to accomplish. For example, in object-oriented systems, polymorphism, and dynamic binding make it difficult for static analysis alone to anticipate runtime behavior. Furthermore, certain assumptions and filtering policies are usually made when performing program analysis for comprehension to cope with the complexity of systems and the needs of users. For example, in dynamic analysis of execution traces, it might be useful to assume that there are no unstructured control flow constructs (e.g., `goto`) in the source code as their presence may complicate both the analysis process and the resulting diagrams.

Static analysis is a software exploration process that relies on the source code of the program in order to derive both: the structure and behavior of software systems including all interactions (i.e., method calls) between objects.[55] Static analysis is usually accomplished without executing the intended programs.[56] This kind of program analysis essentially works by parsing program source files and logging all interactions between internal system components.

Dynamic analysis of software systems is concerned with investigating their behavior at runtime.[57] This kind of analysis, by its nature, does not require the availability of source code, although it may make use of it when present. Indeed, having the source code facilitates injecting tracing code snippets into programs that can help in capturing all interactions taking place at runtime. This can be done by logging all required information about method calls along with their callers and callees. Alternatively, several other dynamic analysis-based techniques do not require the source code to be available. Instead, they depend on (customized) debuggers that can supply them with the required information about program behavior.[52]

Recently, several techniques have been developed to combine both aforementioned types of analysis together in order to obtain what is called *hybrid analysis*. Such techniques are considered to be more effective and precise as the results produced from one analysis is complemented by the results of the other analysis .[58,59] Hybrid techniques exploit the power of both kinds of analysis while alleviating their weaknesses.

4 | OVERVIEW OF STATE-OF-THE-ART TECHNIQUES

This section presents our systematic process of collecting and selecting relevant studies from the literature, analyzing their contents, and discussing their different characteristics in terms of techniques proposed, challenges faced, and strengths/limitations presented.

4.1 | Research Method

In order to systematize our review of the state-of-the-art techniques, we define a data collection protocol and an analysis policy of the collected studies. Our research methodology is inspired by two recent research frameworks in the literature.[2,60] Such research frameworks mainly follow the Kitchenham's guidelines for systematic literature reviews in software engineering.[61] Based on Kitchenham's guidelines, it is important to identify the sources used to search for primary studies, identify their restrictions, and then mitigate any limitations experienced. In addition, all the potentially important sources of papers should be used to make sure no relevant studies are missed.[62] Moreover, Wohlin[63] also encourages to perform a snowballing search[64] as a complementary search to reduce the likelihood of missing important literature, which makes the literature review more reliable.

Therefore, in this paper, we obtain and select our papers of interest using three different techniques: query-based search, venue-based search, and snowballing search. The query-based search depends on building a search query composing a set of keywords. The search query is then used for querying one or more public digital libraries. The venue-based search requires defining a set of most relevant venues to the research topic and investigating all their published papers. The snowballing search requires obtaining a set of most relevant papers and investigating all the papers citing or cited by them.

Each of such search techniques has its own limitation and may miss some relevant and high-impact studies. Keywords in the query-based search may not be comprehensive enough to capture all the relevant papers. It is also highly possible that relevant studies are published in venues other than the predefined ones. Moreover, it is possible for a research paper to have missed citing some related work. In addition, each of the three techniques

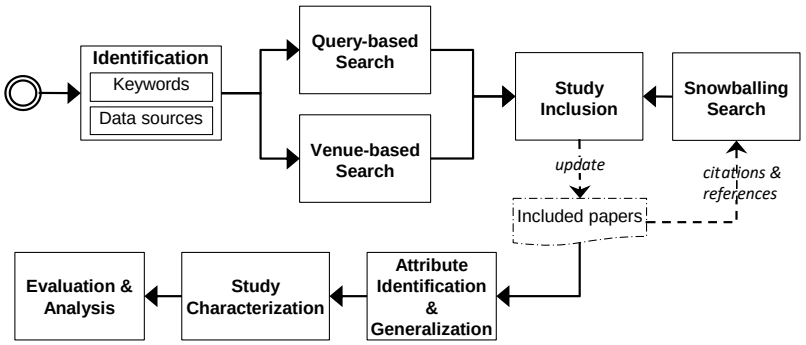

FIGURE 1 Our research methodology

may introduce papers that are irrelevant to the intended research topic. Therefore, it is better to combine more than one paper selection techniques to ensure obtaining and not missing relevant papers.

Mattila *et al.*[2] employed the query-based and venue-based search techniques. However, snowballing was not conducted and the authors clearly consider this limitation to be a threat to the validity of their study. On the other hand, Molleri *et al.*[60] employed the venue-based and snowballing search techniques, while no query-based search was conducted. The authors reported that new sources were identified through the snowballing search, which required them to repeat their snowballing search a second time. Such an overhead in snowballing could be reduced if a query-based search was performed at the beginning. Therefore, in this paper, we propose a hybrid paper selection methodology that performs all the three search techniques and integrates their resulting studies. Moreover, we perform some filtering techniques to reduce the redundancy in our analysis and eliminate the non-relevant or low-impact studies.

Our research methodology starts with defining a set of keywords, data sources, search strategy, inclusion criteria, and data extraction methodology. This is followed by a deep analysis of the particular studies that are highly relevant to our survey. Fig. 1 depicts the research methodology we followed in order to guarantee (to the best possible extent) that all the relevant papers to the use of sequence diagrams for program comprehension are included in our study. The overall process comprises four main stages: 1) identification of data sources, keywords and information to be extracted, 2) performing both a query-based search and a venue-based search, 3) conducting an iterative study inclusion process that involves cross-reference checking using the snowballing search, and finally 4) reviewing all the papers collected to extract the information required for our study.

4.1.1 | Identification of Keywords, Data Sources, and Information to be extracted:

The initial step of our paper collection process is to define the key terms to be used for looking into the literature. To this end, we define 18 key phrases through which we intend to reduce search results by showing only the studies that are highly relevant to our survey. Such keywords have particularly been selected because they convey a thorough description of our topic, which is using reverse-engineered sequence diagram for program comprehension. We kept iteratively building up our set of keywords while we were reviewing the papers collected by using the essential keywords, which are "*Reverse-engineered*", "*Sequence diagram*", "*Program Analysis*", and "*Program comprehension*". From the studies initially reviewed, we collected additional keywords (probably synonyms) that could improve our search query. We kept repeating such a process until we were finding no new keywords.

The set of synonyms of our major keywords are: "Scenario diagram", "*Interaction diagram*", "*Sequence view*", "*Reverse engineering*", "*Design recovery*", "*Behavior recovery*", "*Source code analysis*", "*Program understanding*", "*Software comprehension*", "*Software visualization*", "*Static analysis*", "*Dynamic analysis*", "*Source code instrumentation*", and "*Execution trace analysis*". This combination of keywords composes various synonyms of the main topic of our study concerning reverse-engineered sequence diagrams. However, since keywords alone are generally insufficient for collecting all relevant studies (as they may result in search misses and false hits), we augment this step with further steps of identification.

The second step is the identification of the data sources we will use for applying our search queries. Here, we choose the *Scopus* database to be the main source for retrieving the initial set of papers to be included and reviewed. Then, we specifically defined the key venues in this area of research and manually searched their proceedings for the sake of getting additional relevant studies. The key venues we identified are: *IEEE International Conference on Program Comprehension (ICPC)*, *IEEE Working Conference on Reverse Engineering (WCRE)*, *ACM SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE))*, and *IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*

The third step is specifying the list of information we intend to keep records on from the collected papers in order to help us analyze and evaluate the different techniques. The list of information to be recorded includes paper bibliography details, type of analysis, methodology, experiments, strengths and weaknesses. We could through some information identify different attributes that may help us compare different studies, and those attributes are described in detail in subsection 5.1.

4.1.2 | Query-based Database Search:

A search query string was prepared using an eloquent combination of the keywords identified, as shown below, and then applied to ***Scopus***, one of the largest citation databases of peer-reviewed literature. We used the advanced search mode of ***Scopus*** where we could supply our designed search query. The search process was conducted over the title, abstract, and keywords (i.e., `TITLE-ABS-KEY`) of all the papers in the database.

```
TITLE-ABS-KEY(("Sequence diagram" OR "Scenario diagram" OR
               "Interaction diagram" OR "Sequence view"
              )
              AND("Reverse engineering" OR "Reverse-engineered" OR
                  "Design recovery" OR "Behavior recovery" OR
                  "Static analysis" OR "Dynamic analysis" OR
                  "Source code analysis" OR "Program Analysis" OR
                  "Program comprehension" OR "Program understanding" OR
                  "Software comprehension" OR "Software visualization" OR
                  "Source code instrumentation" OR "Execution trace analysis"
                 )
             )
```

It is clear from the query string that we aim to select all papers that utilize sequence diagrams for representing reverse-engineered program behavior. Therefore, one of the keywords "*Sequence diagram*", "*Scenario diagram*", "*Interaction diagram*", "*Sequence view*" mush exist in the title, abstract and/or the keywords parts of the papers. Because the studies in the literature are inconsistent in selecting key terms representing their work, we assume that every single paper contains at least one of the following phrases: "*Reverse-engineered*", "*Program analysis*", "*Program comprehension*", or at least one of their alternatives. We expect papers that employ program analysis to also mention whether it is static or dynamic, but that should not necessarily be located in the title, abstract, or keywords portions of the paper. However, when we verified this by trying, for example, to eliminate the phrase "Program analysis" from the search query one time, and the "Static Analysis" and "Dynamic analysis" the other time, the results obtained got reduced, which indicates that we might miss some of the important papers. Therefore, we used all the combinations of correlated alternatives of phrases together, even we know that some of them may may sometimes not appear in some papers.

The search query is performed in the end of December 2017 and retrieved 147 results. Out of which, 16 search results were outright excluded since they were conference proceedings summaries. Hence, only 131 papers remained for further analysis. The resulting papers were published between 1979 and 2017 in various journals and conferences.

4.1.3 | Venue-based Search:

Generally, this stage is orthogonal to query-based search and can, therefore, be carried out simultaneously with it or even ahead of it. In our case, venue-based search is primarily conducted to confirm the results obtained through query-based search and to ensure that we did not miss studies that were possibly left out by the query. The main task of this stage is to collect the relevant papers from well-known venues that are mainly focused on reverse engineering, program comprehension, program analysis, and software visualization. Therefore, we looked into the papers published in the proceedings of *ICPC*, *WCRE*, *VISSOFT*, and *PASTE*. In each conference proceedings, we search for the same combination of keywords in a semi-automatic way (i.e., sometimes by manual search). We selected these particular conferences because their proceedings are quite relevant to the overall theme of this review and there is a perfect match between the keywords that formulate this study and the key terms in their titles and descriptions.

ICPC is the premier venue for program comprehension research and has had 25 editions from 1993 to 2017 (was formerly called IWPC prior to 2006). After going through the proceedings of each year (we were unable to find the 1995 proceedings though), we were able to find 8 related papers, three of which were duplicates (have already been found through the query-based search), and thus, ended up with five new papers.

WCRE is the main venue for research on reverse engineering and has had 24 editions from 1993 to 2017 (no conference held in 1994). In 2014, it was actually combined with another conference called *Conference on Software Maintenance and Reengineering (CSMR)* under the *Software Evolution Week*. Then, from 2015 on, these two conference appeared under one umbrella called SANER (that is *International Conference on Software Analysis, Evolution, and Reengineering*). After going through all the proceedings, we found five related papers, four of which were duplicates, and we ended up with only one new paper.

VISSOFT is the major venue for research on software visualization and has had 16 editions from 2002 to 2017 (was not held in 2004 and 2012): 4 editions as *VISSOFT* working conference, 6 editions as *VISSOFT* workshop, and 5 editions as *SOFTVIS* symposium. After looking into the proceedings of each edition, we were able to find two related papers, one of which was a duplicate, and, thus, we ended up with only one new paper.

PASTE is the main venue for research on program analysis and has had 11 editions from 1998 to 2013. There has been no specific pattern of occurrence for this conference though. After going through each edition's proceedings, we were able to collect only one related paper, which was already found by the Query-based search. Therefore, there were no new papers in this particular conference related to the scope of our study.

There also exist other general purpose conferences, such as the *International Conference on Software Engineering (ICSE)*, *International Conference on Software Maintenance and Evolution (ICSME)*, *Automated Software Engineering (ASE)*, and *International Conference on Fundamental Approaches to Software*

Engineering (FASE); and journals, such as *IEEE Transactions on Software Engineering (TSE)*, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *Journal of Software: Evolution and Process (JSEP)*, and *Software: Practice and Experience (JSPE)*. All such venues are in fact indexed in Scopus, which means that applying the same search query in each venue independently would lead to the same resulting studies collected by our generic search query. Based on our manual and semi automated analysis of the four most relevant venues, we observed that there is no significant number of studies added to the list of the generic query. Therefore, for the general purpose venues, we only rely on our enhanced search query and the snowballing search.

4.1.4 | Snowballing (Cross-referencing):

In order to complement our selected set of studies, we further performed a cross-referencing process through a backward snowball of the already included papers. This was carried out through two major steps. First, we sorted all the collected papers based on the number of citations in a descending order, so that highly cited papers are visited first in order to reduce the number of duplicates when we proceed further with this process. After that, we went through the lists of citations of each paper and collected new studies based on the aforementioned keywords and inclusion criteria. Second, we sorted the papers one more time, but now based on the publication year so that papers published recently are listed on the top. Then, we carefully read each paper, looked into its related work, and scanned its references and identified new relevant studies to our survey. As a result of this process, we included 12 out of the 17 initially collected papers (5 papers were eliminated after applying the quality and citation policy discussed in the next subsection), which have then been aggregated with the comprehensive list of candidate papers.

4.1.5 | Study Inclusion:

Based on the resulting list of papers, we start to explore each paper individually to decide whether or not it would be selected for review. We depend for our final selection of each paper primarily on the relevance and impact of the paper.

1. First, in order to include a paper, we check to what extent it relates to reverse engineering sequence diagrams to support program comprehension. We first perform an initial screening of each paper based on the title, abstract, and keywords. We limit our selection to papers written in English only. Although this was the role of the query, we noticed that some papers did mention the key phrases used in the query string, but in a different context. For example, one of the papers published in the period between 1979 and 2000 had the following phrases in its abstract: “time-sequence diagrams" and “time-sequence dynamic analysis". Such statements clearly indicate that the paper matches the query, while however it is not related to our study at all. We figured out that the paper talks about a dynamic analysis method of slipway cribs based on time-sequence. Therefore, our careful reading of the three parts of the paper has produced a subset of papers to be considered for the following filtering process. We actually noticed several papers are duplicate, conference proceedings’ summaries listed as papers, and applying the reverse engineering of sequence diagrams for purposes other than visualizing program behavior. This particular filtering has resulted in a total of 78 papers for the next filtering step.

2. The next filtering step is based on the influence of the obtained papers. The influence here is expressed by the citation count (Scopus citation count was used) for each paper as a function of the publication year. In other words, if the paper has a small number of citations and is not recent, it is therefore excluded from the list of included papers. Usually, papers published recently may gain fewer citations than older ones. Following [2], we initially include all most recent papers published in 2016 and 2017, regardless of their number of citations. For publications of earlier years, a thresholding policy has been applied to limit the number of citations to papers of certain years. Papers whose number of citations is less than the threshold specified for its publication year are excluded from further review. Thresholds are distributed as shown in Table 3. To make sure that we are not excluding high quality papers, we also investigated their citations, and we noticed that most of them are not from from the high quality studies published in the following years. We apply the same criteria for the studies collected using the venue-based search and the snowballing search. As a result of this filtration step, 20 papers (i.e., 14 of Scopus and venue-based papers and 6 of snowballing papers) have been excluded.

3. Since we include all the studies published in 2016 and 2017 regardless of their number of citations, and those published in 2015 with at least one citation, we end up with some studies that are of poor quality. A paper may be considered of a poor quality if it has less number of citations and published in a non-reputable conference/journal. Before excluding such studies, we carefully review each paper to validate whether its findings contributes to the body of the research in this area. To mitigate the bias in deciding whether a paper is of a good or bad quality, we performed the reviews of each paper by two authors of this paper independently, where any disagreements were mediated by the third author. An coincided decision was deduced to exclude five papers, while there was a disagreement on only one paper, where one author decided to include while the other author decided to exclude. The reason of such a disagreement was because that paper was of an acceptable quality but published in an unknown venue. After a discussion with the third author, we eventually decided to exclude that paper. As a result of this

TABLE 3 Inclusion criteria based on the number of citations

Publication year	# of citations to get included
2017 & 2016	0
2015	1
2014-2013	4
2012-2010	8
2009-2000	12
$<$ 2000	16

filtration step, 6 papers (i.e., 4 of Scopus and venue-based papers and 2 of snowballing papers) have been excluded due to their low quality findings.

Overall, the total number of relevant papers to be included has been reduced, after applying our inclusion and exclusion criteria. Out of the 78 relevant papers that were initially selected, 52 papers (i.e., $\approx$ 67%) are considered for further analysis and data extraction.

4.1.6 | Summary Statistics of The Collected Studies:

Table 4 lists a summary statistics of the study collection process, while in Table 5 the distribution of the studies over the well-known venues is presented. For each data source, the number of relevant papers found, and number of excluded/included papers are presented. As we can see, the final set of studies considered for further analysis and review composes 50 papers. A large subset of 37 papers out of 52 are studies presenting new or extended techniques for the use of reverse-engineered sequence diagrams for program comprehension. The other 15 papers, listed in Table 6, also study the use of sequence diagrams for program comprehension but from different perspectives. In other words, they introduced complementary evaluations, analyses, or extensions of sequence diagrams rather than techniques for reverse engineering. Some of such papers presented reverse engineering techniques for non-desktop applications (e.g., mobile or web applications). In this survey, we limit the scope of our survey to the techniques that target programming languages for desktop application.

TABLE 4 Statistics of the collected studies

Source	Total papers	Relevant studies	Excluded studies	Included studies
Query-based search	147	46	13	33
Venue-based search	2, 364	15	8	7
Snowballing search	1, 273	17	5	12
Total	3, 784	78	26	52
				37 Techniques
				15 Other studies

4.1.7 | Data Extraction and Study Analysis:

Reviewing the 52 collected papers shows that the majority of the studies introduced new techniques for the reverse engineering of sequence diagrams. Some other studies simply proposed further extensions, applications, or more in-depth analyses related to previously proposed techniques. Such papers address other important issues regarding the reverse-engineered sequence diagrams, such as the differences between them and forward-engineered sequence diagrams, extensions to UML sequence diagram, comparing existing techniques, etc. Therefore, such studies have not been included in the list of the techniques-driven papers discussed in Section 5, but instead, they have been used to enrich the discussions all over this paper about particular aspects of sequence diagrams. Papers proposing program analysis techniques have been handled separately and then classified based on the types of program analysis (i.e., static, dynamic, or hybrid) they apply. Although some of the studies did not explicitly mention their employed type of program analysis, we were able to infer it from the other techniques they are dependent on.

For each paper proposing a new or extending an existing technique, we extract details that describe the methodology, characteristics, experiments, and case studies of the technique proposed. Each paper was reviewed by an author of this paper, and then double-checked by another author for

TABLE 5 Distribution of studies over venues

Venue	Total number of papers	Number of studies
ICPC/IWPC	807	8
WCRE/SANER	1, 055	5
VISSOFT	365	2
PASTE	137	1
ICSE	5, 233	6
ICSME	1, 247	3
ASE	1, 347	1
FASE	1, 502	3
TSE	3, 515	1
TOSEM	441	1
JSPE	953	1
SPE	3, 385	1
Others	–	19
Total		52

TABLE 6 Studies on sequence diagrams not included in our comprehensive analysis

Study Ref.	Focus/Objective
65	The COMET method using the UML notation.
66	Evaluating UML extended sequence diagram notation called *saUML*
67	Assessing of *saUML* using a controlled experiment
68	Expressing security requirements in UML diagrams
69	Animating sequence diagrams
70	Control flow analysis of UML sequence diagrams
71	Comparing the layout of sequence diagrams of two tools
72	Evaluating user-constructed sequence diagrams
73	Modeling the behavior of mobile applications
74	Modeling the stealthy behavior of Android application
75	Combining different artifacts to get a thorough model
76	Statically generating sequence diagrams from Android source code
77	Reverse engineering of UML sequence diagrams for web applications
78	Modeling event-based interactions of JavaScript
79	Modeling Ajax web applications

confirmation. The extracted information are based on a set of 17 defined attributes. Some of the attributes are inspired from the related surveys presented in Table 2. We also propose other to allow studying and comparing important aspects of the techniques (More details about our set of attributes are discussed in subsection 5.1). Each paper review was based on reading the paper and capturing the important characteristics of the techniques, including the information about the 17 attributes. However, for some attributes, we could not identify information connected to them in some papers and, as a result, we considered them as *unknown*. For example, some papers mention that their techniques are tool-supported (e.g., as Eclipse plugins) but we were unable to find any evidence about their tools. For all the tools that we were able to find their URLs, we provided them as footnotes. Tools that were introduced as Eclipse plugins were just mentioned as so. Moreover, for some studies, we had to approach other sources (e.g., research groups, theses, dissertations, and the profiles of the researchers) related to the reviewed studies for the sake of gathering as much relevant information as possible about their proposed techniques.

4.1.8 | Study Replication:

A replication of our study (e.g., to extend the list of papers) is supported by a depiction of the research methodology presented in Fig. 1 and described in Section 4. In replicating our study, the same search query can be used to query the Scopus database or any other paper indexing databases (e.g., Google Scholar or DBLP) and digital libraries (e.g., IEEE Xplore or ACM Digital Library).

In terms of paper selection criteria, a paper excluded in our study may be included in a study replication in the future due to several reasons. The first reason is related to the inclusion criteria defined in our study. On one hand, in our study, we excluded the papers that do not have the minimum number of citations defined by our citation thresholds. Such papers may receive more citations in the future and, hence, will be included in futures studies as they appear to be more influential. On the other hand, we considered including all the papers published in 2016 and 2017 regardless of their citation count (even though we excluded some due to low quality). However, in the future, papers with no citations in 2016 or 2017 may be excluded in the future if they keep having such zero citation count. The second reason is related to the bias in selecting papers and our judgment of some studies to be irrelevant or of low quality. Such a decision may be subjective if performed by other researchers, which may lead to inconsistent results in comparison to ours.

In terms of data extraction, we believe that using our set of attributes would support obtaining all the important information about the techniques and analyses performed by each collected paper. However, future replicators of the study may identify more attributes that can help in evaluating the different techniques. In addition, deciding weather a paper introduces a novel program analysis technique, extends an existing technique, or just performs further analyses should be straight forward and is expected to produce consistent results to ours.

Finally, although our research methodology may be considered representative, it is, in this paper, more focused to techniques that use reverse-engineered sequence diagrams for program comprehension. This led to eliminating some relevant studies that support program comprehension through reverse-engineered visualization forms other than sequence diagrams. Hence, generalizing our research methodology to studies in other fields of research may not be possible due to the variety of publication venues, and also the unsuitability of our attributes to the contents of papers of other domain.

4.2 | Classification of the Studies

We categorize the collected studies that introduced new techniques based on the type of program analysis they employed for three reasons. First, this classification was commonly used in the literature to distinguish between techniques that depend on program source code and those that depend on program execution. Second, this classification gives a lucid impression about the amount of information that can be obtained about program behavior and interactions using each type of analysis. Third, we noticed that some techniques could infer some information about program behavior (e.g., control flow) using one type of program analysis (say dynamic), while it was known before in the literature that this particular information can only be obtained using the other type of analysis (say static). For example, program control flow was known to be difficult to obtain through dynamic analysis, but afterward, some later techniques were developed that could infer control flow using special algorithms based on the call graph. Hybrid techniques, on the other hand, show how combining both types of analyses can result in more powerful methods, but the at the expense of performance due to the increased overhead.

Fig. 2 shows the distribution of the collected papers over years in 3-year periods. We observe that this field of research has matured to mature during the past and current decades, where its top maturity was during the period between 2006 and 2008. We also observe that most of the static techniques were proposed during the periods of 2000-2002 and 2012-2014, with few studies in the period of 2006-2008. Dynamic techniques appear to be mostly proposed during the period between 2006 and 2011, with only a few papers in the past three years. This does not mean that researchers are no longer proposing dynamic analysis-based techniques in general, but this could indicate that they are utilizing visualization forms other than sequence diagrams. Hybrid techniques appear to be fairly stable across all the periods, and so is the other studies that perform additional analyses on program comprehension using reverse-engineered sequence diagrams.

In the following subsections, we list for each type of program analysis all its dependent techniques, with brief, but illustrative, descriptions of their approaches, strengths, and potential weaknesses. Further details and analyses of the techniques and their relative differences are presented and discussed in Section 5.2

4.2.1 | Static Analysis:

Techniques of this kind of program analysis essentially work by parsing program source files and logging all interactions between internal system components. Starting from the main entry point of the program (e.g., `main` method), the analyzer keeps track of all object construction operations and method invocations between them while taking into account all constructs that control the flow. For every particular interaction in the program, information obtained through the analysis are usually logged into memory as interaction traces to be used by the visualization process at a later stage.

Although static analysis is an efficient analysis method that can easily recover the structure and design of a program, it can only provide a conservative approximation of the runtime behavior of the program, which may lead to less precise behavioral representations. This limitation

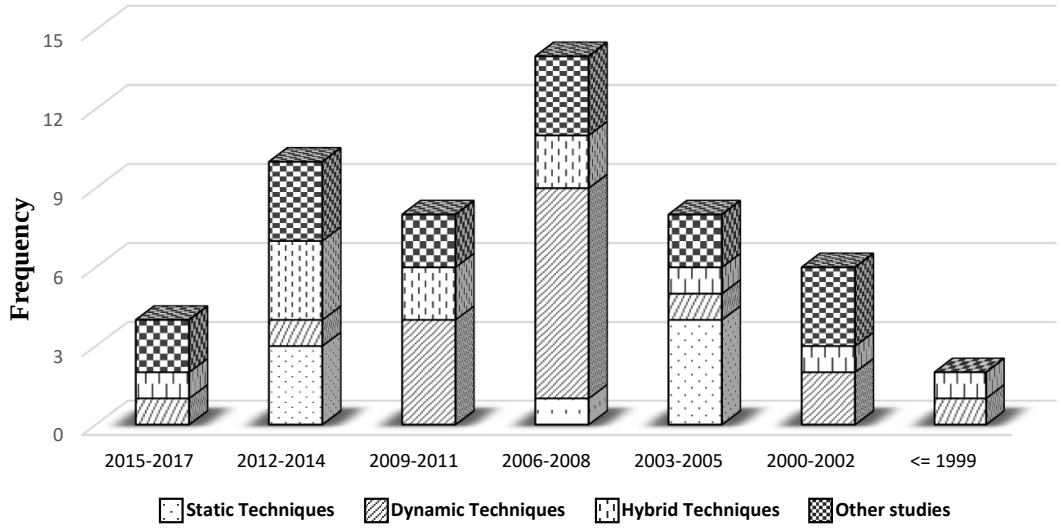

FIGURE 2 Distribution of the collected papers over years

poses many challenges when analyzing programs that make use of dynamic features. For instance, in object-oriented languages with dynamic class loading, such as Java, the interactions between those classes cannot actually be captured or even depicted via static analysis of source code. Another example is the challenge of analyzing distributed software systems that employ multi-users or multi-threaded processes that might have parallel interactions.[24] Furthermore, users (or threads) in such systems can be added or removed at runtime, further complicating the analysis task. In addition, capturing the communication between threads or remote objects is not likely to occur in the static analysis because such information can only be gathered during the execution of the program. However, using static analysis, it is sometimes possible to show an expectation of how such interactions would be executed in the program at runtime. Nevertheless, according to Myers's study[80] and Gueheneuc's recommendation[81], incorporating behavioral information through the static analysis of source code is essential for any technique that aims to visualize dynamic execution traces.

Korshunova *et al.*[29] presented a tool that can 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). The derived structural and behavioral models were represented and stored in XMI files,[82] which represent UML elements in an eXtensible Markup Language (XML) format. Objects and method calls were used to construct sequence diagrams, while activity diagrams were generated from the conditional and loop constructs.

Rountev *et al.*[30] proposed two simple extensions to the UML sequence diagram whose purpose was to precisely capture the intraprocedural control flow of programs. In particular, they proposed a generalized break fragment that enables exiting from a certain surrounding fragment. In addition, they introduced an approach that can precisely map the Control Flow Graph (CFG) to UML and can handle any reducible exception-free CFG. The same first author (with other co-authors) proposed a number of control flow coverage criteria for checking the interactions among a set of interacting objects in a sequence diagram.[83] Another work was also introduced by the same research group, based on the proposed techniques, built an enhanced visualization of sequence diagrams.[84] They actually resolved the scalability limitations of the UML sequence diagrams by proposing a set of tool features, such as zooming, filtering, and details on demand. Accordingly, they could reduce the complexity of the generated diagrams while preserving their readability and navigability.

Lu's study[85] formally demonstrated the relationship conformance of differently generated sequence diagrams. Sequence diagrams of a certain artifact might be reused in an application with various changes in that specific application. In addition, the names of lifelines, messages, and system variables may be changed as well. Therefore, it is important to verify that the sequence diagrams of an application conform to the ones in the artifact, by avoiding name conflicts, and that was the main objective of the work presented by Lu and Kim.[85]

Martinez *et al.*[86] proposed a model-driven framework that applies the Architecture-Driven Modernization (ADM) principles (mainly provided as Eclipse plugins) for the reverse engineering of sequence diagrams. This framework was validated by OMG standards and Eclipse EMF (an open source application platform that is the for model-driven development). However, the resulting sequence diagrams by this framework are simple and even does not make use of all the primitives of the UML notation.

Roubtsov *et al.*[87] proposed I2SD, a tool for the reverse engineering of Java interceptors. An interceptor is a way of defining a behavior, explicitly in the source code, into the execution of a Java application. Their tool was deployed as a stand-alone tool and was also integrated with NetBeans to support developers (or quality assessors) with means of visualizing interceptors while developing programs using the Enterprise JavaBeans. A notable feature of their tools is that they can deal with incomplete programs (i.e., programs that are still under development).

Tonella and Potrich [88] proposed RevEng, a tool for recording interaction diagrams (collaboration and sequence diagrams) from C++ code. The tool employs a static analysis of the code to produce an approximated system behavior in any execution and for any possible input. The tool is applicable for partial analysis of systems (e.g., incomplete systems), and supports focusing on a certain functionality of a system.

4.2.2 | Dynamic Analysis:

Although dynamic analysis can recover precise information about the actual behavior of a system, it is challenged by the problem of coverage [89]. A run of the program explores a sample execution trace typically covering only a few behaviors out of many more possible ones. Increasing coverage, and hence obtaining a complete view of the program's behavior, requires multiple executions of the program with varying parameters representing different execution scenarios. Having multiple runs, however, entails the generation of multiple sequence diagrams, posing the other challenge of combining these diagrams into a single comprehensive one. [24]

Another limitation of dynamic analysis is that the information it gathers always represents the series of interactions in a program without considering the control flow constructs through which these interactions passed. [90] It, therefore, lacks the ability to capture information about whether these interactions passed through conditional alternatives (e.g., `if`, `switch`, exception handling, etc.), repetitions (e.g., `for`, `while`, `do..while`, etc.) or recursive calls. Notice that such information can be easily collected using static analysis of the source code. To allow dynamic analysis-based techniques to identify such kind of information, it is helpful to utilize the availability of source code to inject certain scripts for recognizing the changes in the program control flow. Such scripts will do their job, while the source code of the program is instrumented using multiple traces. [24]

Another challenge is that dynamic analysis can only work with complete software systems, rather than software fragments or components. This means that dynamic analysis of subsystems may require them to be instrumented by automatically employing program stubs (e.g. a specially crafted main class) to enable the collection of sample executions of certain program components. [91]

Briand *et al.* [24] introduced a dynamic analysis technique of program execution that works by instrumenting programs through the creating several aspect-oriented snippets (in AspectJ) that can capture object creation and method calls, and then log all interactions into appropriate trace files. They addressed the issue of distributed systems, and how message senders and receivers can be identified when objects are executed at different network nodes. Their resulting sequence diagrams were partial in a way that each individual trace has its own sequence diagram, and thus, a single combined diagram is not produced.

Grati *et al.* [92] introduced an instrumentation-based approach for collecting, visualizing, and compacting execution traces of different interaction scenarios. Its compaction works by merging different fragments, deleting unnecessary objects and messages, and also renaming them with meaningful names. In comparison with Briand's approach [24], diagrams here were generated with a less time and expressive output.

Oechsle *et al.* [93] targeted students by proposing *JAVAVIS* to help them understand the interactions executed at runtime of small-sized programs. They represented each active method by an object diagram, whereas the whole program was represented using a single sequence diagram. Their approach was well presented as a visual debugger providing a graphical user interface through which a user can run the program step-by-step and visually observe what is happening, control the speed of the animations, and select the classes required to be displayed. Each loop iteration is represented as a separate block of messages. Calls to system classes' methods are filtered out and not shown in the sequence diagram.

Rohr *et al.* [33] proposed a monitoring tool called *Kieker*, which continuously (or on-demand) observe the behavior of Java programs. Their instrumentation strategy was based on AspectJ as well. Their main focus was on Web applications in which the behavior differs from one user to another. The purpose of their approach was to minimize the monitoring overhead as much as possible. Performance analysis and evaluation of the approach were also demonstrated. [94] Their experimental evaluation showed that *Kieker* could achieve a smaller linear overhead compared with others, which makes it potentially appropriate in industrial settings.

Another tool, called *Scene*, was proposed by Koskimies and Mossenbock [95] for the reverse engineering of sequence diagrams for programs developed using *Oberon*, an object-oriented programming language. Their tool provides different facilities for users, such as running instrumented program using different inputs and navigating the classes, methods, or calls representing certain interactions in the diagrams.

A reusable and extensible framework called *Form* was proposed by Souder *et al.* [96] to allow the construction of tools for the analysis of runtime behavior of distributed and standalone software systems. The authors validated that framework by building a tool that can profile Java distributed systems and then generate UML sequence diagrams representing their execution.

The goal of two techniques [97,98] was to reduce the size of sequence diagrams by removing the less important details and methods from them. One of these techniques applied the dominance algorithm on the dynamic call graphs to compute the dominance relation between objects, while the other were based on four compaction rules for eliminating similar, repeated, and repeated interactions. Based on their previous tool (called *Amida*, [99]) they proposed two different but complementary tools: the first one performs the dynamic analysis of the program execution, while the other statically analyzes the produced trace file. Their experiments showed that their approach could remove 40% of the objects from execution traces. However, the elimination of such objects can reduce the precision of the resulting diagram, which lead to missing important details about the program behavior.

Compacting sequence diagram was also addressed by Dugerdil *et al.*, [100] where their approach reduced the complexity of the generated diagrams by merging lifelines of similar functionality into clusters (using the Single Linkage Algorithm (SLA) clustering algorithm, [101]) removing accessors, and compressing repetitive sequences. As a result, the produced sequence diagrams only present interactions between the different clusters.

Jiang *et al.* [102] proposed an approach for mining API usage scenarios while running application programs, filtering them, and then synthesizing UML sequence diagrams that illustrating them in a well-formed manner. Here, instead of compacting the resulting diagram, another approach was employed to disconnect nested combined fragments to simplify their presentation to users, which led to losing some of the communication calls with the ability to retrieve them whenever a deep revision of usage scenarios is needed.

Ziadi *et al.* [103] introduced a technique that performs dynamic analysis of Java programs without the need for source code; i.e., only the Java bytecode was needed. Each trace of program execution was represented as a Labeled Transition System (LTS), a variant of classical finite automata. All labeled transition systems related to one program are then merged using the *k-tail* algorithm. [90] Sequence diagrams are then identified as regular expressions to eventually generate sequence diagrams that conform to all traces. A step-by-step description of the approach was presented. While conditional alternatives and iterations are hard to recognize via dynamic analysis, this approach was able to detect them while merging the different traces. They could achieve that by using the *k-tail* algorithm that had the ability to go through the different paths in the finite states and detect all possible blocks of interactions. However, the authors stated that the *k-tail* algorithm could sometimes be inaccurate, and more accurate algorithms needed to be investigated.

Maoz and Harel [104] proposed a model-based dynamic technique that is different from the others in the sense that it relies on the designer's-provided input. In other words, the interactions between system's objects are collected from the execution traces and then abstracted to be visualized based on the behavioral models used for its design. This is helpful to explore the relationships between a system's actual runtime and its design specification> For example, one can use such a technique for testing whether a system functionality conforms its expected design, or even identifying how different system runs behave. However, some information about program execution are hidden in the produced sequence diagrams, as the techniques is based on certain filtering rules.

Based on reverse-engineered sequence diagrams, Sag and Tarhan [105] proposed a *COSMIC* (Common Software Measurement International Consortium) solver by developing an AspectJ-based tracer that retrieves sequence diagrams during program execution with the help of AspectJ-based instrumentation. Such diagrams are then used by a *COSMIC* calculator that applies *COSMIC* measurement rules to calculate function points.

Noda *et al.* [106] proposed a dynamic analysis-based technique that reduces the number of lifelines (and interactions) in reverse-engineered sequence diagrams by selecting only the important objects in a software system. The identification of the core objects is achieved by detecting the long-lived and frequently accessed objects in the execution traces. Although this approach makes sequence diagrams more readable, it could lead to losing some information about certain interactions in the system.

Lo *et al.*proposed an approach for mining modal and symbolic specifications [107,108]. Their approaches mine system's execution traces to identify objects and group them into class-level abstractions and visualize them using Live Sequence Charts (LSC), a formal extension of UML sequence diagrams.

Malnati *et al.* [109] presented JThreadSpy, a dynamic analysis-based tool that captures execution traces of each program thread and then displays them in a UML sequence diagram. The tool was aimed to help students understand the impact of synchronization and critical section, deadlocks, and other concurrent program anomalies in a multicore architecture.

4.2.3 | Hybrid Analysis:

Hybrid program analysis techniques are generally considered to be time-consuming as the instrumentation overhead increases with the implementation of the two types of analysis. [58] In addition, the multiple sequence diagrams generated by dynamic analysis and the one generated by the static analysis all need to be merged into a single comprehensive diagram to represent the entire program behavior. [80]

Noda *et al.* [44] proposed a tool that could minimize the number of interactions between objects by means of meta patterns usage, which led to reducing the total number of messages in the history of object interaction resulting in a vertically compressed sequence diagram. Their method relies on the control dependencies with ASTs captured through a static analysis along with the dynamic program dependence graph collected during program execution. An evaluation of the effectiveness of their approach on program comprehension was conducted in a non-systematic way based on three case studies and developers' points of view about the resulting diagrams.

Labiche *et al.* [58] presented a reverse engineering technique that combines both static and dynamic analyses. Their main objective was 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. Other information can be obtained via static analysis, which in turn collects the control flow information to eventually generate separate scenario diagrams (sequence diagrams representing specific program scenarios) per each trace. This technique, however, does not produce a complete sequence diagram that combines the different generated sequence diagrams.

Malloy and Power [110] introduced a tool called *SPIDOR* for the reverse engineering of sequence diagrams (and other diagrams) from C++ programs. Their tool was developed in Python, with the help of Python/C API, along with *gcc* and AspectC++. The main objective of this tool was to visualize the dynamic interactions among objects in C++ applications where users can, through a profiler, supply filters (i.e., objects and message of interest) to reduce the size of the produced diagrams. However, users may need to navigate the other produced diagrams in order to get a full picture of a software behavior, as the generated sequence diagrams do not hold all the important information about the flow of control of that software.

Myers *et al.*[80] introduced a technique that uses both static and dynamic analyses to collect information about programs. Their objective was to improve the visual appearance of the generated sequence diagrams. They did so by introducing an algorithm that compacts a large amount of information of call/message interactions between system objects. Although this kind of abstraction is useful, it can adversely affect precision as important interactions may be hidden from the user.

Dalton and Hallstrom[111] proposed a platform-neutral visualization toolkit for *TinyOS* applications for the same of improving program comprehension. Their approach employed both static and dynamic program analyses using the Java-based analysis and instrumentation library for TinyOS 2.0. Static analysis was used to generate call graphs and to instrument source code. A dynamic analysis was used to trace program behavior and visualize it by means of sequence diagrams. Although sequence diagram generated by the toolkit proposed are visualized in an easy to understand and track fashion, users can only visualize a single runtime trace at a time.

Hamou-Lhadj and Lethbridge[45] proposed an approach for retrieving summarized sequence diagrams based on static and dynamic call graphs of a certain trace and requires a user input to accomplish this task. The summarization works by removing the unnecessary routines (e.g., constructors) or any other methods the user wishes to exclude from the resulting diagram. This may actually be useful for software engineers who can imagine which methods are important and which are not. Normal users, however, are not aware of the importance of methods as they may filter some methods out while they perform an essential functionality of the system.

Toda *et al.*[112] also proposed a technique for abstracting sequence diagrams, but here by obtaining information from design patterns employed in the source code to group objects in the resulting lifelines of sequence diagrams. This helps in reducing the volume of interactions, which hence helps in improving understandability of the visualized software behavior.

Ng *et al.*[113] proposed MoDeC, a semi-automated dynamic technique (that depends on another static technique called DeMIMA[114]) for the identification of behavioral and creational design motifs in source code. It requires users to describe and design motifs and to trigger different scenarios to generate traces of a certain system. After that, it builds a dynamic model of the motifs depending on the static occurrences identified by DeMIMA, which is then translated into a constraint satisfaction problem (CSP). Solving the CSP helps in assigning roles to the objects and interactions from the program diagrams in the motif scenario diagram.

Recently, Srinivasan *et al.*[115] also introduced a technique that combines both static and dynamic analyses. Their technique produces a sequence diagram from the source code analysis and another sequence diagram from a dynamic execution trace. Then, after applying appropriate filtering and redundancy removal routines, these two diagrams are combined according to object signatures and line numbers. The resulting sequence diagram was further optimized to reduced its complexity while preserving its comprehensiveness, and this led to enhancing program program understandability.

4.3 | Other Related Studies

In this subsection, we discuss the studied presented in Table 6. The first set includes studies that provided further analyses and experiments on how to improve program comprehension using sequence diagrams, but without proposing any new reverse engineering approaches. The other set of studies includes the ones that do not address computer programs (i.e., either mobile or web applications). For example, Gomaa[65] introduced a tutorial on COMET, a Concurrent Object Modeling and Architectural Design Method of real-time and distributed applications. This method actually integrates object-oriented and concurrency concepts and applies the UML notation, including sequence diagram notation for expressing the dynamic characteristics of such systems.

Xie *et al.*conducted an empirical evaluation of UML sequence diagrams extended with a special notation for representing thread interactions called *saUML*, synchronization-adorned UML.[66] Another assessment was also conducted by a subgroup of the same authors using a controlled experiment with a set of comprehension tasks.[67] Such studies, however, assessed the positive impact of their extended sequence diagrams towards program comprehension through manually designed (rather than reverse-engineered) diagrams. Jürjens[68] also proposed a set of extensions to UML diagrams (including sequence diagrams) to express security requirements, and their semantics are formally modelled.

Burd *et al.*[69] proposed a technique for animating sequence diagrams by showing them to users in a step-by-step fashion instead of showing the whole diagram in a static way. They applied their technique to a simple scenario of sending a fax, and then evaluated how their animated diagram improved program understandability, in comparison with the static diagram, using a questionnaire-based survey composed of two questions and two participants. However, such a study evaluated animated and static sequence diagrams that were programmatically drawn (i.e., they were not revere-engineered as well) with a very limited number of participants and comprehension tasks.

Garousi *et al.*[70] studied how to maintain the program control flow in UML sequence diagrams by proposing an extended metamodel for activity diagram to improve understandability of the program control flow in sequence diagrams. Another study compared the layout of sequence diagrams produced by two popular tools: *Rational Rose* and *Borland Together*.[71] That comparison was performed using a set of criteria for the purpose of distinguishing the level of comprehension of two software systems using the sequence diagram designed by the two different tools. However, that study was not carried out under a controlled environment where users themselves should be involved in the assessment process of understandability and usefulness of the differently generated diagrams. Another study conduced a controlled experiment on a set of students by asking them to

manually construct sequence diagrams from execution traces of a system.[72] The aim of that study was to see whether users would be able to diagnose system failures using execution-based tracing information only. The results indicate that, in order to diagnose a failure, one should first localize the fault from the execution trace and then use trace modeling to properly finish the diagnosis.

From a different perspective, there were other relevant studies that deal with reverse-engineered sequence diagrams but with systems other than computer programs, such as mobile or web applications. In our study, we limit the scope to techniques that deal with computer applications. However, we give here a brief description about some of such techniques, but we omit them from the comprehensive evaluation conducted later in this paper. For example, modeling the behavior of mobile applications was also investigated in the literature, either using the UML standard or relevant extensions.[73] Huang *et al.*[74] proposed a tool for modeling the stealthy behavior of Android application by detecting the semantic mismatch between the program behavior and user interface. Kowalczyk[75] proposed a method to enhance understandability of the behavior of mobile applications by combining different artifacts, from different sources, to obtain a thorough and accurate model. Another recent work proposed *StaticGen*, a tool for generating sequence diagrams of Android applications using static program analysis.[76]

Another trend of reverse engineering techniques focuses on web-based applications.[77,78,79] 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. Alalfi *et al.*[77] described how *PHP2XMI*, a reverse engineering tool, can be used to retrieve role permissions of web pages at the access level and represent them in sequence diagrams. They evaluated *PHP2XMI* at the entity level and investigated how it can be used to test other security vulnerabilities of web applications, such as SQL injection. Their approach was able to filter execution traces related to database insertions, and automatically exclude any information that might complicate the comprehension process, and eventually produced the corresponding sequence diagrams that represent the interactions between users and web pages.

5 | EVALUATION OF THE STUDIED TECHNIQUES

In this section, we construct an attribute framework and assign attribute values to each studied technique based on the data collected thorough our review of the papers. We then use the results to distinct and compare the different techniques proposed by the studied papers. In our analysis, we highlight the sufficiency, usability, and usefulness of the sequence diagrams that are reverse-engineered by these techniques, and study their suitability for program comprehension purposes. This is accomplished by (a) defining a set of attributes and (b) projecting them onto the techniques surveyed. The in order to reveal their strengths and limitations, which can help us to identify the gaps that should be addressed in future work.

5.1 | Attribute Framework

After we have identified the papers of interest, we construct an attribute framework that can be used to characterize and compare the selected papers. In this section, we describe the attributes used to extract the data from the selected studies and how we came up with such attributes.

5.1.1 | Attribute Identification:

Constructing an attribute framework was motivated by our intention to show how the sequence diagrams produced by each of the techniques are understandable, scalable and usable. The understandability power of each technique is expressed by the amount of information about program interactions can be retrieved through the type(s) of program analysis employed. For example, being able support multiple programming languages and satisfy multiple objectives can demonstrate how a techniques is more usable and applicable to a wider range of software systems. We also introduce an attribute for showing whether the case studies used to asses the proposed techniques of the surveyed papers, in addition to another attribute for assessing whether program comprehension was evaluated by the surveyed studies.

To build our attribute framework, we inspired some attributes from relevant surveys (shown in Table 2) in program analysis and program comprehension. Examples of the inspired attributes are: the programming language, the visualizing diagram(s), lifeline levels, support of conditions, support of loops, tool support, and multi-threading support. Then, we performed one pass of paper reviews for the set of recent papers (i.e., papers published in 2017) and recorded all the key information about the methodology, produced diagrams, and findings of each paper. For each paper being reviewed in this pass, we kept assigning values to the attributes that were inspired by the related studies. At the end of this paper review pass, we grouped the recorded information under a set of initial attributes.

5.1.2 | Attribute Generalization:

Some attributes have been chosen to represent a set or group of attributes. For instance, the 'loop identification' attribute refers to the ability of the technique to gather information about the different kinds of iterative constructs, such as `for`, `while`, and `do..while`. What urged us to group them this way was the observation that some techniques were restricted to identifying only a limited number of constructs of each category. This is due to

TABLE 7 Attribute Framework

	Attribute	Description
1	Dependent Technique(s)/Tool(s)	Lists all supporting tools or techniques employed by a given technique
2	Major Objective(s)	Describes the main objective of the technique on which the authors concentrated
3	Case Study	Describes whether the technique has been empirically tested and validated on real projects
4	Condition Identification	Indicates the ability of the technique to capture alternatives in a program (`if`, `switch`, etc.)
5	Loop Identification	Indicates the ability of the technique to capture iterative loops (`for`, `while`, etc.)
6	Recursive calls	Indicates the ability of the technique to capture recursive calls
7	Threads Identification	Indicates the ability of the technique to identify communicating threads
8	Multi-users Identification	Indicates the ability of the technique to identify communicating users
9	Compaction/Summarization	Indicates whether the technique filters the messages to summarize the resulting diagram
10	Multiple Scenarios	Indicates whether the technique performs the instrumentation with different scenarios
11	Merging Diagrams	Indicates whether the technique combines scenario diagrams into a single sequence diagram
12	Tool Support	Indicates whether the technique has an available tool or is just a prototype
13	Comprehension Evaluation	Indicates whether the technique was evaluated towards improving program comprehension
14	Program Analysis	Identifies the type of analysis (static, dynamic or hybrid) employed by the technique
15	Programming Language(s)	Identifies the type of programming language on which the technique works
16	Lifeline Level	Refers to the level at which entities are interacting with each other
17	Utilized Diagram(s)	Lists all possible output visualization diagrams used by the technique

the fact that having the ability to identify one form of loop constructs, for example, indicates the capability of identifying the other forms as well. Therefore, techniques that can capture at least one construct of each of the condition identification and loop identification attributes are considered to be satisfying that attribute.

While reviewing the rest of the papers, some attributes in the initial list were dropped (e.g., potential stakeholders) due to inadequate information for them in the majority of the papers. However, we keep some other attributes (e.g., comprehension evaluation) that were assigned values for only a few papers due to their key importance in distinguishing the techniques from each other. The reason why we keep such attributes is that they . The resulting framework was then used for performing another pass of paper review over the rest of the papers.

5.1.3 | Resulting Attribute Framework:

Our final framework composes a total of 17 attributes listed in Table 7. The majority of the attributes are identified during our review of the papers. These attributes have not been previously used in relevant surveys to distinguish program analysis techniques for program comprehension. As we stated before, the remaining attributes were inspired by relevant surveys (Table 2) in program analysis and program comprehension. We utilize these attributes to characterize all the techniques that perform program analyses to generate reverse-engineered sequence diagrams. We aim through our attributes to expose knowledge about how usable, scalable, and effective the state-of-the-art techniques to improve program comprehension.

A subset of these attributes (e.g., the programming language, the other diagram(s) used, and the dependent tool(s)) may seem unhelpful in demonstrating how the surveyed techniques are useful for program comprehension. They can, however, help researchers and practitioners to select the appropriate technique that satisfies their needs from different perspectives, such as usability and scalability. All the remaining attributes, on the other hand, are more explanatory towards demonstrating the effectiveness of the surveyed techniques for program understanding and how their produced diagram(s) are expressive and meaningful.

5.2 | Characterization of the Studied Techniques

In this section, we present the assignment of attributes to the studies that employ reverse-engineered sequence diagrams for program comprehension. We then use of the assignment results to summarize the research body.

5.2.1 | Attribute Assignment:

Using the aforementioned attribute framework, we carefully reviewed all the candidate studies and assign appropriate attribute values for each. The attribute values effectively capture the essence of the studies and their proposed techniques in terms of usability, scalability, and comprehensibility for a clear distinction between (and comparison of) the the studied techniques. The assignment process is performed by the first two authors of this

TABLE 8 Article Characterization Results

Study Ref.	Dependent Technique(s)/Tool(s)	Major Objective(s)	Case Study
[85]	*Unknown*	Conformance of different SDs	JHotDraw and Monetary Access Control
[30]	RED	Capturing exact intraprocedural control flow	21 Java library components
[83]	RED	Coverage in sequence diagrams' interactions	18 components of Java libraries
[84]	RED	Interactive visualization of sequence diagrams	Standard Java libraries
[29]	Columbus/CAN, DOT	Coordinating objects and messages	30 KLoCs, 60 KLoCs projects
[86]	Eclipse-MDT MoDisco	Model-driven reverse engineering	*eLib* Java program
[87]	SQuAVisiT, [116] EJB [117]	Sequence diagrams of interceptors	GlassFish Product Manager
[88]	*Unknown*	Producing interaction diagrams from C++ Code	A C++ software of over 400 KLoCs
[24]	*Unknown*	Distributed systems	A library system
[92]	Smith-Waterman algorithm [118]	Interactive visualization & compact diagrams	ATM simulation system
[93]	JDI	Learning programming	Small-sized programs
[33]	UMLGraph, Plotutils, Graphviz	Reducing time overhead & multi-user web apps	iBATIS JPetStore
[99]	JVM Tool Interface (JVMTI)	Recording and visualizing execution traces	A tool management system
[98]	Amida [99]	Compacting SDs	jEdit, Gemini, Scheduler, LogCompactor
[97]	Amida [99]	Compacting SDs	jEdit, Eclipse
[95]	Oberon [119]	Compacting SDs + User-Interaction	Kepler, Compiler Construction Framework
[96]	JVM Profiler Interface (JVMPI)	A framework for distributed systems analysis	A Technical Report System (TRS)
[100]	JavaCC, SLA clustering [101]	Composing lifelines into clusters	Their trace analyser itself
[102]	MAS, [120] ATL [121]	Understanding API usage	Not mentioned
[103]	K-tail algorithm [90]	Merging SDs	A project of 500+ classes with 25K LOCs
[105]	LogSequencer [122]	Building a COSMIC Solver	A payment database application
[106]	SELogger [123]	Summarizing SD lifelines and interactions	jpacman, JHotDraw
[104]	S2A [124]	Model-based reverse-engineered SDs	News Ticker, C++ mobile app, Biological system
[108]	AspectJ	Mining symbolic scenario-based specification	Space Invaders game
[107]	AspectJ	Mining modal scenario-based specifications	Jeti
[109]	ASM framework [125]	Visualizing multi-threaded interactions in sequence diagrams	Simple lock and deadlock examples
[44]	JavaCC, JDI	Abstracting the history of object interactions	jpacman, JHotDraw, Checkstyle
[53]	Rigi [126]	Overlapping info of static and dynamic views	FUJABA project
[52]	JExtractor, Rigi, [126] SCED [127]	Deriving SCD from SD	FUJABA project
[58]	Their work [24]	Reducing time overhead	5 large systems & 2 small programs.
[110]	*gcc*, AspectC++	Visualizing dynamic interactions of objects	An Arcade game
[80]	*Unknown*	Compacting SDs	Eclipse IDE, HSQLDB, Jetty Platform
[111]	*nesC* analysis and instr. library	Visualizing runtime behavior of TinyOS apps	Blink & RadioCountToLeds apps
[45]	BIT, [128] their technique [129]	Summarizing large traces	Weka
[115]	*Unknown*	Generate concise sequence diagrams	Polyshape & Animal system
[113]	DeMIMA [114]	Demonstration of design motifs	JHotDraw & 5 motif-applying Java applications
[112]	Reticella [44]	Sequence diagram abstraction and objects grouping	JHotDraw

survey (i.e., one author assigns values to the attributes for all the papers and then the other author confirms the findings through another review of the paper).

For each attribute, we assign a value (or a set of values) to the proposed techniques based on the data collected through our review of the selected studies. While reviewing each study, we tried our best to assign values to the attributes based on our judgments rather than what the authors claim to contribute. For example, we investigated different sources to make sure that a techniques is really tool-supported. We also infer some facts from the backend technique(s) or tools used to construct the proposed program analysis techniques. For example, it is not likely that a static analysis-based technique would recover multi-threaded interactions. However, for some papers, where no enough technical details available or the methodology description is vague, we proceed with what the authors claim with reservation. If a paper is an extension of another prior work, we make an assumption that most of the characteristics of the prior work exist in the recent work. Nevertheless, we inspect the recent work to check whether such an assumption is valid.

5.2.2 | Characterization Results:

The characterization of the 37 selected articles is shown in Tables 8 and 9. We split the results into two tables due to the space limitation of the pages. Table 8 shows three characteristics of the studied techniques, namely its dependent technique(s), the major objective(s), and the case studies used to

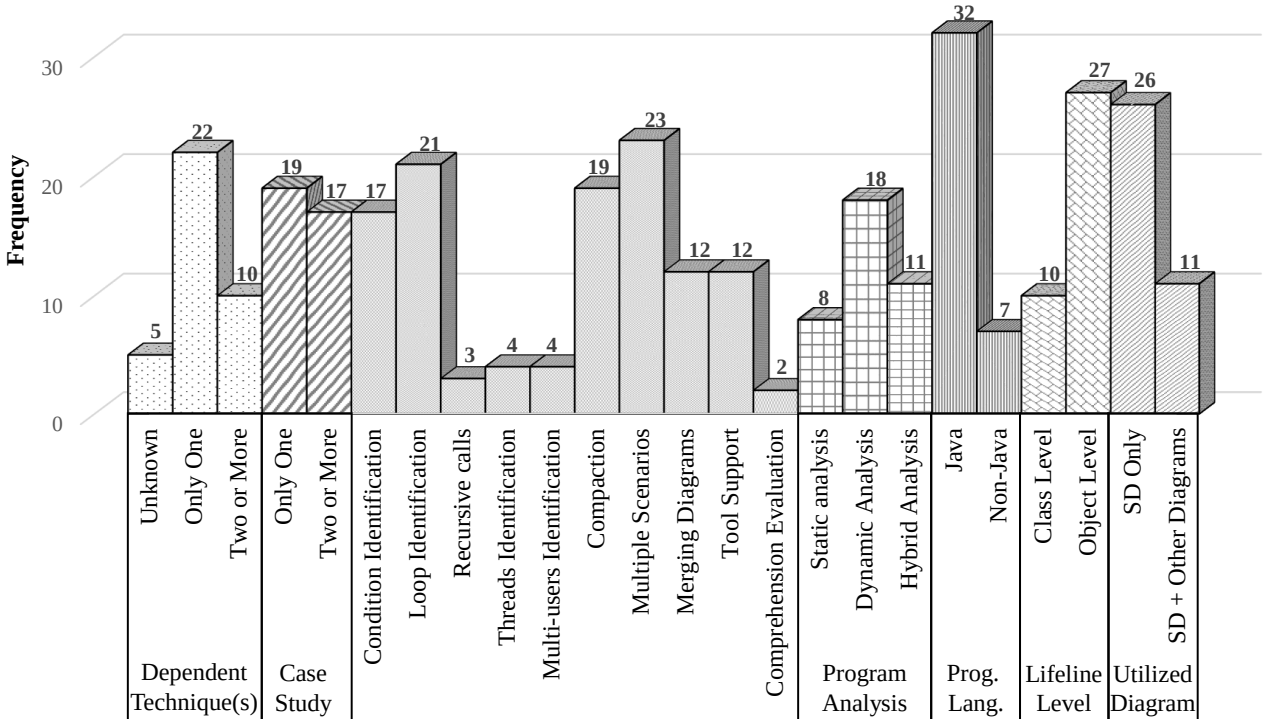

FIGURE 3 Distribution of the attributes across the studied papers

demonstrate its usefulness. Table 9 shows how the reviewed techniques is distinguished from or compared to the other techniques in relation to the remaining 14 attributes. Overall, techniques with the lowest number of checkmarks are supposed to employ the UML standard of sequence diagrams, or even sometimes parts of its features. Such techniques were basically proposed for purposes other than improving program comprehension. In particular, pure static analysis-based techniques offer fewer features in comparison with those that perform dynamic analyses, even though the program control flow can better be captured using the static analysis of program source code.

Fig. 3 shows the distribution of the attributes across the studied papers. Since some of the attributes have been assign non-logical values (e.g., Dependent Techniques, Case Study, Program Analysis, Programming Languages, Lifeline Level, and Utilized Diagrams), we classify the values into two or three categories to facilitate showing the distribution for each of such attributes. For example, for the Program Analysis attribute, we show the frequency of Static, Dynamic, and Hybrid techniques, independently. In the following subsection, we analyze and discuss all the observations obtained from our comprehensive reviews of the selected studies.

5.2.3 | Analysis & Discussion:

In this section, we interpret our findings in terms of the patterns that we recognize, the feature that are not adequately support by existing program analysis techniques but may be useful for program comprehension, the commonly used features, and what is still needed to improve program comprehension through reverse-engineered sequence diagrams? To this end, we analyze the summarized results presented in Tables 8 and 9 looking for the most and least common attributes and interesting attribute combinations.

Most common attributes: Understanding the most common attributes (as demonstrated in Fig. 3) gives an impression of the most widely supported features of program analysis techniques that can improve program comprehension. As we can see, Java is the most common value (under the Programming Language attribute). Java is considered the most popular object-oriented programming language nowadays. However, there are other popular programming languages that are being used by developers these days (e.g., .NET languages, Python, etc.), and many other languages may be introduced in the future. This raises two questions: 1) Can these techniques be adapted to be applicable to other programming languages? 2) Is the UML sequence diagram standard appropriate for representing interactions and control flow of programs developed by other programming languages? The work proposed by Lu *et al.* [85] answers the first question as it applies its techniques to two totally different programming languages, Java and Prolog. The second question is also addressed by that work in addition to others. [29,95,110,111]

The second most common attributes are the lifeline (object level) and the use of sequence diagram only. The level of interactions (i.e., *class level* versus *object level* lifeline) is also an important issue to be considered when visualizing software behavior. Visualizing interactions in the class level

TABLE 9 Article Characterization Results (Continued)

Study Ref.	Condition Identification	Loop Identification	Recursive calls	Threads Identification	Multi-users Identification	Compaction/Summarization	Multiple Scenarios	Merging Diagrams	Tool Support	Comprehension Evaluation	Program Analysis	Programming Language(s)	Lifeline Level	Utilized Diagram(s)
85	✓	✓	.	.	.	.	.	.	.	.	Static	Java & Prolog	object	SD
30	✓	✓	.	.	.	.	.	.	✓[1]	.	Static	Java	object	SD
83	✓	✓	.	.	.	.	.	.	.	.	Static	Java	object	SD
84	✓	✓	.	.	.	.	.	.	.	.	Static	Java	object	SD
29	✓	✓	.	.	.	.	.	.	✓[2]	.	Static	C++	object	SD, CD & AD
86	.	.	.	.	.	.	✓	.	.	.	Static	Java	object	SD
87	✓	✓	.	.	.	.	✓	.	✓[3]	.	Static	Java	object	SD
88	.	.	.	.	.	.	✓	.	.	.	Static	C++	object	SD & ColD
24	✓	✓	.	.	✓	.	✓	.	.	.	Dynamic	Java	object	SD
92	✓	✓	.	.	.	✓	✓	✓	.	.	Dynamic	Java	object	SD
93	.	.	.	.	.	.	✓	✓	✓[4]	.	Dynamic	Java	object	SD & OD
33	.	.	.	.	✓	.	✓	✓	✓[5]	.	Dynamic	Java	class	SD, MC, CDG & TD
99	.	✓	✓	.	.	.	✓	.	✓[6]	.	Dynamic	Java	object	SD
98	.	✓	✓	.	.	✓	✓	✓	.	.	Dynamic	Java	object	SD
97	.	✓	.	.	.	✓	✓	✓	.	.	Dynamic	Java	object	SD
95	.	.	.	.	.	✓	✓	.	✓[7]	.	Dynamic	Oberon	object	SD
96	.	.	.	✓	✓	✓	✓	.	✓[8]	.	Dynamic	Java	object	SD
100	✓	✓	.	.	.	✓	.	.	✓[9]	.	Dynamic	Java	class	SD
102	✓	✓	.	.	.	.	✓	✓	.	.	Dynamic	Java	class	SD & SMD
103	✓	✓	.	.	.	.	✓	✓	.	.	Dynamic	Java	object	SD
105	.	.	.	.	.	.	✓	.	.	.	Dynamic	Java	object	SD
106	.	.	.	.	.	.	✓	.	.	.	Dynamic	Java	object	SD
104	.	.	.	.	.	✓	✓	✓	✓[10]	.	Dynamic	Java & C++	object	SD
108	.	.	.	.	.	✓	.	.	.	.	Dynamic	Java	class	SD & LSC
107	.	.	.	.	.	✓	.	.	.	.	Dynamic	Java	class	SD & LSC
109	.	.	.	✓	.	.	.	.	.	.	Dynamic	Java	object	SD
44	✓	✓	✓	.	.	✓	.	.	✓[11]	✓	Hybrid	Java	object	SD
53	.	.	.	.	.	✓	✓	✓	.	.	Hybrid	Java	class	SD & SCD
52	.	✓	.	.	✓	✓	✓	✓	.	.	Hybrid	Java	class	SD & SCD
58	✓	✓	.	.	.	✓	✓	.	.	.	Hybrid	Java	object	SD
110	.	.	.	.	.	✓	.	.	✓[12]	.	Hybrid	C++	object	SD, CD, CG & ComD
80	✓	✓	.	✓	.	✓	.	.	.	.	Hybrid	Java	object	SD
111	.	.	.	.	.	.	✓	.	.	.	Hybrid	nesC	class	SD, CG
45	✓	✓	.	.	.	✓	.	.	.	✓	Hybrid	Java	class	SD
115	✓	✓	.	.	.	✓	✓	✓	.	.	Hybrid	Java	class	SD
113	✓	✓	.	✓	.	✓	✓	✓	.	.	Hybrid	Java	object	SD
112	.	.	.	.	.	✓	.	.	.	.	Hybrid	Java	object	SD

SD: Sequence Diagram, **AD**: Activity Diagram, **MC**: Markov Chains, **TD**: Timing Diagram, **SCD**: State Chart Diagram, **CD**: Class Diagram, **OD**: Object Diagram, **CDG**: Component Dependency Graphs, **SMD**: State Machine Diagram, **CG**: Call Graph, **LSC**: Live Sequence Chart, **ColD**: Collaboration Diagram, **ComD**: Communication Diagram

[1] RED: *No link available!*
[2] CPP2XMI: http://www.laquso.com
[3] I2SD: http://www.laquso.com
[4] JAVAVIS: *No link available!*
[5] Kieker: http://kieker-monitoring.net/features
[6] AMIDA: http://sel.ist.osaka-u.ac.jp/people/ishio/amida/
[7] SCENE: *The link provided in the paper is no longer working!*
[8] FORM: *The link provided in the paper is no longer working!*
[9] Integrated in Rational Software Architect (RSA), an Eclipse plugin
[10] Tracer: http://www.wisdom.weizmann.ac.il/~maozs/tracer/
[11] Reticella: Implemented as an Eclipse plugin
[12] SPIDOR: *No link available!*

can provide a general overview of the communications between high-level system components. However, our results show that the majority of the techniques employed the object level (i.e., detailed interactions). The reason is that the level of abstraction in the class level may hinder many of the key interactions that might be useful for users to see, especially when such interactions are performed between objects of the same class. On the other hand, program interactions at the object level may lead to complex diagrams that are not easily comprehensible. Typically, using object-level visualization involves showing interactions between classes as well. All the static techniques we studied use objects as lifelines, while dynamic and hybrid techniques differ from each other in this perspective, with the majority of them using objects as lifelines as well.

It also appears that the majority of the techniques that generate reverse-engineered sequence diagrams do not utilize other forms of visualization. The techniques that rely on additional diagrams or views may introduce another form of complication to the users, since the use would need to open different diagrams in order to understand a certain behaviour of the program. Some of the techniques generate the other diagrams as intermediate representations in order to deduce informative elements for the generation of sequence diagrams accordingly.[102] Some others supplement their resulting sequence diagrams with additional views for summarizing interaction messages, showing supplementary information, or just visualizing them in other forms of visualization rather than sequence diagrams.[29,33,52,53,93]

We can also observe that most of the techniques produce multiple diagrams. It is common for dynamic analysis-based techniques to produce multiple diagrams, each of which represents a different scenario of program execution. Hybrid techniques also generate more than one sequence diagram: for through a static analysis and one or more through a dynamic analysis. Consequently, generating multiple diagrams for a single program would distract users as they may need to move from one diagram to another in order to grasp certain behaviors of the program. Hence, it is more useful to employ some mechanisms to combine the generated diagrams into a single comprehensive one. However, only half of the techniques that generate multiple scenario diagrams attempted to merge the resulting diagrams to produce a single general diagram. Other techniques consider such a functionality is superfluous and leave it as a future work. This may give an impression of the challenge of combining different diagrams as it is considered to be a tree- or graph-related problem.

Most common objectives: All our studied papers proposed program analysis techniques to generate reverse-engineered sequence diagrams. In general, the common objective amongst most of the papers is to provide as much expressive and understandable sequence diagrams as possible. However, every paper was motivated by certain problem and attempt to target that problem as a goal. We were able to identify various objectives as presented in Table 8. The most common objective among the techniques was to sequence diagram compaction. Compacting sequence diagrams is concerned reducing the size of the generated sequence diagrams to better improve program comprehension. Other techniques focused on capturing particular aspects of program interactions other than the control flow, or analyzing the performance of the techniques proposed. To the best of our knowledge, except for two studies, none of the papers studied performed an experimental evaluation of how the proposed technique improved program comprehension, even though some papers explicitly state that their main objective is to support program comprehension.

Deciding on which program analysis techniques (i.e., Static, Dynamic, or Hybrid) a paper should propose is tightly related to what objective the paper is aiming to target. Gathering information about program interactions through static and dynamic analyses would definitely lead to more informative reverse-engineered sequence diagrams. Static program analysis is the basis of reverse engineering. However, based on our findings, it is possible for dynamic analysis techniques to capture most of the information that can be captured through static analysis techniques. Therefore, our results show that the majority of the state-of-the-art techniques are rely on dynamic program analysis. Overall, there are 8 static, 18 dynamic and 11 hybrid program analysis techniques. The main issue of static analysis-based techniques is that they sometimes produce the *expected software behavior* rather than the real behavior of the program. In addition, not all interactions shown in statically-generated sequence diagrams are assumed to be executed during program runtime. Hence, dynamic techniques are more preferred in this respect as they reflect the actual software behavior, even though they tend to complicate the resulting diagrams and produce more than one scenario diagram for a single program. This is why hybrid techniques became more popular recently, as they can deduce information from statically-generated diagrams to abstract dynamically-generated ones, while paying the price of processing overhead as a consequence.

Least common attributes: As we can see, There are several features that are not supported by most of the techniques studied, such as the recursive calls, thread identification, and multi-users identification. Recursion generally be direct or indirect. Direct recursion occurs when a method calls itself from within its body. Indirect (mutual) recursion occurs when a method `methA` calls another methods `methB`), which, in turn, calls `methA`. Although such scenarios of recursive calls can be easily detected using static program analyses, we have found no static technique claiming doing so. However, there were two dynamic techniques[98,99] and one hybrid technique[44] that were capable of detecting recursive calls, but they did not show how recursive interactions are presented in the resulting sequence diagrams. All the other techniques present interactions executed inside recursive calls as series of messages, or perhaps recognize them as loops (especially for dynamic analysis-based techniques). Features concerning the dynamic nature of software, such as the identification of the interactions between threads or users within a running software, are unobserved in all static techniques and only applied in a few dynamic techniques. This is expected as recognizing such kind of interactions requires sophisticated methodologies to be utilized. Only a few techniques dealt with multi-threaded[80,96] and distributed systems.[24,33,52,96]

Another least common attribute is the one related to evaluating program comprehension. As we mentioned earlier, every technique has its own objective in terms of abstraction, scalability, usability, and coverage. However, only two studies ([44] and [45]) evaluated how their reverse-engineered

diagrams could improve program comprehension. The authors of the other techniques usually used their own (subjective) judgments on how their generated sequence diagrams could increase the level of program understandability. Most of such techniques are concerned with sequence diagram compaction (e.g., [44,95,100,115]). For example, Oechsle *et al.* [93] proposed a technique to aid students in understanding their own programs but they did not empirically evaluated how much increase in program understandability their techniques introduced. This encourages researchers in this particular area to investigate how effective and adequate are reverse-engineered sequence diagrams for program comprehension, in comparison with other means of visualization.

Correlated attributes: The identification of conditional branches and iterative interactions are the most correlated attributes. All the techniques that are capable of identifying conditions are also capable for identifying loops, but not the vise versa. There are four techniques that were capable of identifying loops but did not support (or at least we did not find any evidence) condition identification. Techniques that employ static analysis are naturally capable of capturing conditional and repetitive branches of programs, which could also be achieved by some dynamic techniques after employing supplementary algorithms for detecting them during program execution. However, there is one static technique that did not supplement its produced sequence diagrams with such information as the main objective behind the study was to show how model-driven principles can be utilized for the reverse engineering of sequence diagrams.[86] Hybrid techniques are also supposed to support the visualization of such information since they employ static analysis in addition to dynamic analysis. Yet, some hybrid techniques appeared to misidentify conditions and loops,[52,53,110,111] despite the fact that such information can easily be obtained from source code. For purely dynamic techniques, employing sophisticated algorithms in order to group all interactions executed within loops or alternatives is required.

Another implicit correlation between two attributes appears with the multiple scenarios and merging diagrams. Basically, a technique would merge diagrams if it produces many. In addition, direct rendering of program interactions into a sequence diagram, that is mostly experienced by dynamic techniques, may result in an awkward and hard-to-comprehend visualization. To alleviate such a problem, it is better for program analysis techniques to identify certain patterns that can help in deducing control flow constructs, such as alternatives, repetitions, or even recursive calls. Almost all the hybrid techniques, except one technique, employed the summarization functionality to their dynamically-produced diagrams based on statically-generated information from the source code. On the other hand, around half of the dynamic techniques employed such a functionality to compact the resulting sequence diagrams. They do so by reducing the amount of messages that are detected to belong to a unified fragment, or sometimes the messages that are following the same pattern in their behavior. None of the static analysis-based techniques employed any summarization functionality. The reason could be because the flow of interactions can easily be extracted from the Abstract Syntax Tree (AST) of the source code.

Attributes with heterogeneous values: The dependent technique and the case study attributes experience high variations in their assigned values for the techniques studied. As Table 8 shows, almost all techniques depend on at least one other technique in order to accomplish their goals. This dependency is habitual to allow focusing on essential objectives rather than building a tool from scratch. In previous studies,[58,97,98] authors built their techniques based on tools that were previously designed by them but for other purposes. However, some studies,[24,80,85] we were unable to find sufficient information about their dependent technique(s) or tool(s). It is also worth noting here that we do not list the programming environment(s) used to develop the techniques as one of the dependent techniques.

Similarly, the variety of evaluation experiments and case studies conducted to validate the techniques collected indicates the absence of any unified benchmark to be used for such a purpose. This encourages researchers in this particular area to investigate how to come up with a framework through which the impact and sufficiency of reverse-engineered sequence diagrams for program comprehension can be evaluated. Such a framework may also involve other means of visualization in the evaluation process. Another observation is that all the techniques, including the dynamic ones, were validated using open source projects as case studies,[130] even though dynamic analysis does not require source code to be available. Authors have not actually stated the reason behind this, but we believe that the main purpose of doing this is to provide other researchers with enough freedom to replicate the experiments. In addition, there are some dynamic analysis-based techniques that need to instrument the source code in order to get the required information during runtime.

General Remarks: To summarize, we observe that the state-of-the-art techniques do not usually focus on the amount of information that should be incorporated in the resulting diagrams to effectively understand programs. Most of the attention was given to the use of the standard UML sequence diagrams and refining the obtained visualization by means of sequence diagram compaction or merging, or even by using additional diagrams of different forms. However, to properly address the issue of program comprehension, much attention should be paid to the collection of more expressive information about program control flow using static analysis first, and then integrating it with dynamic information. This would actually help users understand written programs and conduct maintenance tasks effectively, increasing their productivity. Moreover, doing so can reduce the need for other features of program analysis techniques as the resulting sequence diagrams would be more expressive by themselves, which can also minimize the complexity and improve the performance of such techniques.

We notice that only one-third of the techniques (i.e., 12 out of 37) have built tools to make them available to users. However, for a few studies that claimed tool support, we were unable to find any information about where their tools are actually available (i.e., no URLs or Eclipse plugins were provided). For some others, links to tools were provided but are no longer accessible online. Therefore, we tried to look for them using manually searching online and looking into the web pages of the authors and their research groups. For the ones we were able to locate their tools, we provided

the information about where to get them as footnotes appearing beside the checkmarks shown in the 'Tool Support' column of Table 9. For techniques whose supporting tools were unreachable, the footnotes show the names of the tools along with a '*No link available!*' phrase. Techniques that are not supported by tools are marked with a dot "·".

From a different perspective, there are commercial modeling tools (e.g., *Rational Rose*, *Visual Paradigm*, *Enterprise Architect* and *Altova UModel*) and also open-source tools (e.g., *ArgoUML*, *StarUML* and *Modelio*) that allow users to perform a reverse engineering of sequence diagrams. Such tools support different programming languages, including Java, C++, and C#. In addition, such tools allow users to manually design sequence diagrams (i.e., forward engineering). However, all such tools utilize the standard UML notation for reverse-engineered sequence diagrams and provide users with the option to enhance them manually after generation. Moreover, all these tools are considered to be static-analysis-based tools as they require source code to be available for the reverse engineering process. Among all techniques surveyed, we found only one technique in the literature [115] that relied on a commercial modeling tool for constructing sequence diagrams, which was *Visual Paradigm*, while none of the techniques attempted to utilize or extend a publicly available open source modeling tool.

6 | POTENTIAL EXTENSIONS TO UML SEQUENCE DIAGRAMS FOR ENHANCING PROGRAM COMPREHENSION

Since the core objective of reverse-engineered sequence diagrams is to demonstrate the interactions executed in a program using certain message elements, it is necessary to incorporate such elements with more meaningful information that can increase the understandability level of software behavior. To this end, we propose to extend the UML sequence diagram standard with proper notations that can more precisely reflect the information that needs to be visualized.

From a technical perspective, static-analysis-based techniques are preferable over dynamic ones for extracting and abstracting such kind of information. This is due to the fact that static analysis has full access to program source code, which enables capturing and gathering such information from the AST of the analyzed programs. Therefore, programming languagesâĂŹ parser generators (such as CUP LALR [131] for java) are considered the starting point for employing a reverse engineering of such kind of informative sequence diagrams, but they may require more sophisticated extensions (such as [132]) to allow tracing the AST nodes, node-by-node, and then collect the required control flow nodes. One can equip such extensions to the open-source, publicly available modeling tools that already support the reverse engineering of sequence diagrams (such as the ones introduced in the previous section).

6.1 | Lifeline and Method Categories

This extension is useful for identifying general categories of classes/objects represented by lifelines in the UML sequence diagram. Each category can relate to either the source that originated the interaction message or the one that received it. This distinction of lifelines could be useful in recognizing how the system is interacting with the different internal/external entities within the system itself or the environment under which it is executed. Examples of lifeline categories may include program objects, anonymous objects, program classes, program interfaces, system console, file system, GUI controls, network sockets, remote objects, etc. With respect to method differentiation, it can happen that, in addition to the existing message kinds available in UML, methods called in the program can be of different categories as well. For example, `native` methods in Java are not defined in Java programs themselves. Instead, these methods are actually written in other languages and can be called through Java JNI. Static, overridden, synchronized, and other types of methods should also be identified in the resulting sequence diagram to help users figure out the actual flow of control in the program. One can look at the work proposed by Dugerdi and Repond *et al*., [100] who studied how to cluster lifelines based on functionality, as a starting and motivating point.

6.2 | Variable declarations and assignments

Variable names are commonly used in the labels of the messages or fragment operands. Showing only the usage or application of a variable poses some kind of ambiguity for users since there is no information given about that variable in the sequence diagram. For example, in an `opt` fragment, we may come across the following condition: `i>0`, but no further information is given about what `i` is, how its value is changed and when. Therefore, we suggest planting variable declarations (maybe along with all assignment statements that can affect their values as well) in an appropriate way so that users can track them easily.

6.3 | Events and event handlers

As mentioned earlier in this paper, GUI-based interactions should be visualized in sequence diagrams to allow users to identify the interactions among the execution of GUI-based applications. The current standard UML notation does not support the identification of the different events a program can execute. For example, button-pressing, mouse-clicking, window-resizing and many other events can trigger a call for a set of program interactions. Characterizing such events along with their sources in the sequence diagram would increase the the level of comprehensibility of any GUI-based software. Keep in mind that all interactions that can be triggered by GUI events are considered as inactive when static analysis is used. Therefore, it is necessary to employ dynamic analysis techniques in order to effectively support such an extension.

6.4 | Nested calls

Nested calls are the calls that trigger other calls inside their arguments. This means that in order to invoke this kind of method calls, all its in-argument calls must be invoked first. Representing such a case in the standard sequence diagram can be achieved, but in a way that is different from what appears in the source code. For example, a call for a method, say `foo`, that has two in-argument calls inside, say `a` and `b`, can be represented by this statement: `foo(a(), b());`. This means that the original caller only originated the call for the method `foo`, but to accomplish this call, it is required to call the method `a` then the method `b` (assuming a left-to-right evaluation order). Standard UML sequence diagrams would represent this situation using three independent messages from the caller to the callee in the sequence: `a()`, `b()`, and `foo()`. Although this representation is semantically fine, it does not reflect the actual flow of control in the program. Therefore, nested calls should have a meaningful notation that makes them comprehensible by system users.

6.5 | Chained calls

Chained calls refer to the sequence of calls that are dependent on each other. Here, the original call does not have calls inside its arguments, but inside the object that is returned by that call. For example, the statement `foo().a().b();` represents a chain of calls where the call for the method `b` requires a call to the method `foo` first and then to the method `a`. Again, the UML standard of sequence diagrams does not have a precise representation for such kind of execution-dependent calls, and it would be very useful for program comprehension to have such a form of calls faithfully represented, and hence the need for an extended notation.

7 | THREATS TO VALIDITY

Our research methodology composes different steps of identifying, searching for, filtering, selecting, and reviewing the relevant studies. Part of the process is automated (e.g., performing the search query and study inclusion based on citation count), while the majority of the process is manually implemented. Therefore, our methodology may possibly miss miss some relevant studies. Nevertheless, we considered several measures to ensure that significant studies are collected and analyzed, as follows:

- **Keywords:** It is indeed impossible to come up with a comprehensive set of keywords that perfectly satisfies the goal of obtaining studies with no additional processing. Although keywords identified in our search query comprises various synonyms of the main key terms of our goal, we might have missed some relevant studies. To address this threat to construct validity, we constructed a search query that is eloquent enough to grab the most relevant studies. We do so by developing a representative combination of the defined keywords to make sure that every relevant paper should have mentioned at least one of the keywords combinations in the paper's title, keywords, or abstract. However, not all papers containing such keywords are considered relevant. Therefore, we performed a further analysis of the initial list of studies to ensure that the related papers are included while the unrelated papers are excluded.

- **Data sources:** We identified one global database and four major conferences as the main sources of data collection. However, it is not unusual to miss relevant papers from Scopus and the other venues, or possibly find them in other sources. Therefore, to address this threat to construct validity, we introduced a hybrid approach that integrates results obtained by using the search query with those obtained from the individual venue-based search. We then used the obtained studies to perform a snowballing search. Snowballing allows to obtain relevant studies that were missed by our both query-based and venue-based search techniques. We performed the snowballing process on the papers citing or were cited by the papers collected through the query-based and venue-based searches.

- **Data extraction:** There is a possible threat to internal validity threat that may impact our approaches for paper selection and data extraction. We mitigated the paper selection bias (i.e., paper relevance and quality) by screening each paper independently by two researchers, where any

disagreements were mediated by a third researcher. Paper reviews and data extraction on the final set of our collected papers were performed by two authors of this paper. To address any potential inaccuracies in the collected data, each paper was reviewed by an author of this paper, and then double-checked by another author for confirmation. For papers where there was lack of technical details, we had to approach other sources (e.g., research groups, theses, dissertations, and researchers' profiles) related to the reviewed studies for the sake of gathering as much relevant information as possible about the proposed techniques in the collected papers.

- **Classification of the techniques:** Another possible threat to internal validity is related to the classification accuracy of studies according to the type of program analysis used. In some studies, as the descriptions for some of the techniques lacked sufficient information in this regards. However, we did carefully review those techniques (including the techniques they are dependent on) to deduce any pertinent information about the type of analysis performed by each technique.

- **Study replication:** Replicating the study presented in this paper is possible as long as all the steps of the research methodology presented in Section 4 and shown in Fig. 1 are followed. However, there could be some inconsistencies in the results obtain due to potential changes in citation counts and possible differences in opinions regarding the exclusion of low quality papers. To reduce any potential inconsistencies in prospective replicating studies in the future, we described every particular step of our methodology, thresholds used, number of included/excluded papers in each step, and the distribution of papers across venues. However, there could be an external threat to validity threat when it comes to generalizing our research methodology to studies in other fields of research. Due to the variety of publication venues, and also the unsuitability of our attributes to the contents of papers of other domains, our research methodology may not be suitable.

8 | CONCLUSIONS

This paper presents a systematic review in which reverse engineering techniques that focus on recovering program interactions and presenting them using sequence diagrams are studied. These techniques usually employ static, dynamic, or hybrid program analysis in order to extract the needed information from programs. Several approaches in this regard have been investigated and presented in a way that their features, limitations, and operation are demonstrated. Then, we have identified several attributes against which the reviewed techniques are qualitatively evaluated and compared with each other, followed by an extensive summary of how the collected techniques fulfill these attributes with an analytical discussion about the major observations about the commonalities and differences between the different techniques. Finally, the paper presents a set of potential extensions to sequence diagrams to improve the presentation and understandability of program interactions and different aspects of control flow.

The paper draws three main conclusions with respect to the application of reverse-engineered sequence diagrams as a means of enhancing program comprehension of program interactions and control flow. We articulate these conclusions as follows:

- The UML sequence diagram still needs to be accompanied by an expressive and extended notation to allow encompassing more details about program interactions and control flow that are important for program comprehension. It is known that the UML standard has been adopted by many tools that produce reverse-engineered sequence diagrams and introducing an extension to its notation would require extending all existing tools in order to make them able to incorporate that extension. This implies that compatibility issues would be faced and the new extensions can simply be tool-specific. This issue was discussed by Rountev *et al.*[30] who showed the trade-off between precision and interoperability and suggested that currently available control flow primitives in UML should be utilized instead of introducing additional ones, if interoperability is a driving concern.

- There are only a few studies that evaluated the effectiveness of reverse-engineered UML sequence diagrams for program comprehension, and these studies were limited in terms of the number of participants and questions.[44,45] However, these two studies presented hybrid reverse-engineering techniques, which means that purely static and purely dynamic techniques for reverse engineering of sequence diagrams have not been evaluated for that purpose yet. For purely dynamic analysis techniques, there are some other studies that evaluated how effective their visualization of software behavior, but that studies were actually based on forms of visualization other than sequence diagrams where the flow of control of programs is not maintained (e.g., *EXTRAVIS*[20] and *EXPLORVIS*.[41]) On the other hand, we could actually find three other studies that aimed to evaluate sequence diagrams for program comprehension,[66,67,71] but these studies only considered sequence diagrams (and extensions to it) that are forward-engineered rather than reverse-engineered. Therefore, it is strongly recommended for future research to address the evaluation of reverse-engineered UML sequence diagrams, taking into consideration comparison between: a) statically-generated and dynamically-generated diagrams, b) standard and extended diagrams, and c) sequence diagrams in general and other forms of visualization.

- Tools supporting the reverse engineering techniques of sequence diagrams have not been compared with the commercially available tools, or even with the open source ones. In general, commercially tools are primarily designed for modeling software systems using different UML diagrams (including sequence diagrams). What the researchers might not possibly realize is that most of these techniques support the

reverse engineering of sequence diagrams from programs developed by different programming languages. What actually distinguishes the techniques reviewed in this paper from that tools is the effort towards improving the presentation of software behavior through sequence diagrams. The commercial tools compete with each other with providing a user-friendly environment to facilitate the manual improvement of the appearance of the resulting (designed or reverse-engineered) diagrams, such as filtering, formatting, searching, etc. Studies in the literature, on the other hand, compete with each other in providing more efficient techniques that can produce compact, expressive, and easy-to-comprehend diagrams with a minimal or no participation from users.

For future work, we recommend focusing on program visualization using (extended) reverse-engineered sequence diagrams and further investigating their added value for program comprehension. To this end, we aim to strictly address additional visualization goals,[133,134] such as expressiveness, scalability, usability, and user-interactivity. We believe that taking the satisfaction of such visualization goals into account, while developing a visualization tool, would lead to having users being able to precisely and effectively comprehend program control flow and software behavior in general without the need to navigate to source code or to use another conventional method of understanding. In addition, UML sequence diagrams are well-known by almost all software engineers and have already been used by most of software engineering tools. Therefore, having an extended version of that notation in the future to be used for program comprehension is more preferable as it would require a minimal learning curve for its prospective users, and expected to be superior to the other forms of visualization.

Moreover, we aim to consider the current limitations of the UML standard of sequence diagrams by addressing the extensions proposed in this paper, and even more. We believe that program understandability and maintainability would highly be improved if such extensions have been embedded to UML sequence diagrams using proper meaningful notations. Doing so would require developing a program analysis technique that is capable of extracting all program information needed to construct each of the extensions. In other words, existing static program analysis techniques, as noticed in this paper, only gather information that is required to build the standard UML sequence diagram, which means that their resulting diagrams would not help us demonstrate the other aspects of program control flow. On the other hand, existing dynamic program analysis techniques, by their nature, cannot capture the flow of control of programs. Therefore, there is a necessity to develop new techniques that integrate both static and dynamic analyses. In such a way, the static program analysis is used to capture program control flow, while the dynamic program analysis is used to enrich the resulting static diagram with the behavioral aspects of the program captured at runtime.

References

1. Müller Hausi A, Jahnke Jens H, Smith Dennis B, Storey Margaret-Anne, Tilley Scott R, Wong Kenny. Reverse engineering: A roadmap. In: :47–60ACM; 2000.
2. Mattila Anna-Liisa, Ihantola Petri, Kilamo Terhi, Luoto Antti, Nurminen Mikko, Väätäjä Heli. Software visualization today: systematic literature review. In: :262–271ACM; 2016.
3. Diehl Stephan. *Software visualization: visualizing the structure, behaviour, and evolution of software.* Springer Science & Business Media; 2007.
4. Basit Hamid Abdul, Hammad Muhammad, Koschke Rainer. A survey on goal-oriented visualization of clone data. In: :46–55IEEE; 2015.
5. Seriai Abderrahmane, Benomar Omar, Cerat Benjamin, Sahraoui Houari. Validation of software visualization tools: A systematic mapping study. In: :60–69IEEE; 2014.
6. Isaacs Katherine E, Giménez Alfredo, Jusufi Ilir, et al. State of the art of performance visualization. *EuroVis 2014.* 2014;.
7. Schots Marcelo, Vasconcelos Renan, Werner Cláudia. A quasi-systematic review on software visualization approaches for software reuse. *Technical report.* 2014;.
8. Paredes Julia, Anslow Craig, Maurer Frank. Information visualization for agile software development. In: :157–166IEEE; 2014.
9. Novais Renato Lima, Torres André, Mendes Thiago Souto, Mendonça Manoel, Zazworka Nico. Software evolution visualization: A systematic mapping study. *Information and Software Technology.* 2013;55(11):1860–1883.
10. Caserta Pierre, Zendra Olivier. Visualization of the static aspects of software: A survey. *IEEE transactions on visualization and computer graphics.* 2011;17(7):913–933.
11. Canfora Gerardo, Di Penta Massimiliano, Cerulo Luigi. Achievements and challenges in software reverse engineering. *Communications of the ACM.* 2011;54(4):142–151.

12. Carpendale Sheelagh, Ghanam Yaser. *A survey paper on software architecture visualization.* : University of Calgary; 2008.
13. Kienle Holger M, Muller Hausi A. Requirements of software visualization tools: A literature survey. In: :2–9IEEE; 2007.
14. Storey Margaret-Anne D, Čubranić Davor, German Daniel M. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In: :193–202ACM; 2005.
15. Hamou-Lhadj Abdelwahab, Lethbridge Timothy C. A survey of trace exploration tools and techniques. In: :42–55IBM Press; 2004.
16. Koschke Rainer. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice.* 2003;15(2):87–109.
17. Teyseyre Alfredo R, Campo Marcelo R. An overview of 3D software visualization. *IEEE transactions on visualization and computer graphics.* 2009;15(1):87–105.
18. Bassil Sarita, Keller Rudolf K. Software visualization tools: Survey and analysis. In: :7–17IEEE; 2001.
19. Cross James H, Hendrix T Dean, Barowski Larry A, Mathias Karl S. Scalable visualizations to support reverse engineering: A framework for evaluation. In: :201–209IEEE; 1998.
20. Cornelissen Bas, Zaidman Andy, Deursen Arie. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering.* 2011;37(3):341–355.
21. Cornelissen Bas, Zaidman Andy, Van Deursen Arie, Moonen Leon, Koschke Rainer. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering.* 2009;35(5):684–702.
22. Bennett Chris, Myers Del, Storey M-A, et al. A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. *Journal of Software Maintenance and Evolution: Research and Practice.* 2008;20(4):291–315.
23. Merdes Matthias, Dorsch Dirk. Experiences with the development of a reverse engineering tool for UML sequence diagrams: a case study in modern Java development. In: :125–134ACM; 2006.
24. Briand Lionel C, Labiche Yvan, Leduc Johanne. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transactions on Software Engineering.* 2006;32(9):642–663.
25. Pacione Michael J, Roper Marc, Wood Murray. A comparative evaluation of dynamic visualisation tools. In: :80–89IEEE Computer Society; 2003.
26. Pacione Michael J, Roper Marc, Wood Murray. A novel software visualisation model to support software comprehension. In: :70–79IEEE; 2004.
27. Chikofsky Elliot J, Cross James H, others . Reverse engineering and design recovery: A taxonomy. *IEEE Software.* 1990;7(1):13–17.
28. Briand Lionel C. The experimental paradigm in reverse engineering: Role, challenges, and limitations. In: :3–8IEEE; 2006.
29. Korshunova Elena, Petkovic Marija, Brand MGJ, Mousavi Mohammad Reza. CPP2XMI: reverse engineering of UML class, sequence, and activity diagrams from C++ source code. In: :297–298IEEE; 2006.
30. Rountev Atanas, Volgin Olga, Reddoch Miriam. Static control-flow analysis for reverse engineering of UML sequence diagrams. *ACM SIGSOFT Software Engineering Notes.* 2005;31(1):96–102.
31. Rountev Atanas, Connell Beth Harkness. Object naming analysis for reverse-engineered sequence diagrams. In: :254–263ACM; 2005.
32. Lanza Michele, Ducasse Stéphane. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering.* 2003;29(9):782–795.
33. Rohr Matthias, Hoorn André, Matevska Jasminka, et al. Kieker: Continuous monitoring and on demand visualization of Java software behavior. In: ACTA Press; 2008.
34. Von Mayrhauser Anneliese, Vans A Marie. Program comprehension during software maintenance and evolution. *Computer.* 1995;28(8):44–55.
35. Rugaber Spencer. Program comprehension. *Encyclopedia of Computer Science and Technology.* 1995;35(20):341–368.

36. Storey Margaret-Anne. Theories, methods and tools in program comprehension: Past, present and future. In: :181–191IEEE; 2005.

37. Brooks Ruven. Using a behavioral theory of program comprehension in software engineering. In: :196–201IEEE Press; 1978.

38. Soloway Elliot, Ehrlich Kate. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering.* 1984;SE-10(5):595–609.

39. Kulkarni Aniket. Comprehending source code of large software system for reuse. In: :1–4IEEE; 2016.

40. Eisenbarth Thomas, Koschke Rainer, Simon Daniel. Aiding program comprehension by static and dynamic feature analysis. In: :602–611IEEE; 2001.

41. Fittkau Florian, Finke Santje, Hasselbring Wilhelm, Waller Jan. Comparing trace visualizations for program comprehension through controlled experiments. In: :266–276IEEE Press; 2015.

42. Wettel Richard, Lanza Michele, Robbes Romain. Software systems as cities: a controlled experiment. In: :551–560ACM; 2011.

43. Penta Massimiliano Di, Stirewalt RE Kurt, Kraemer Eileen. Designing your next empirical study on program comprehension. In: :281–285IEEE; 2007.

44. Noda Kunihiro, Kobayashi Takashi, Agusa Kiyoshi. Execution Trace Abstraction Based on Meta Patterns Usage. In: :167–176IEEE; 2012.

45. Hamou-Lhadj Abdelwahab, Lethbridge Timothy. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In: :181–190IEEE; 2006.

46. Fernández-Sáez Ana M, Chaudron Michel RV, Genero Marcela, Ramos Isabel. Are forward designed or reverse-engineered UML diagrams more helpful for code maintenance?: a controlled experiment. In: :60–71ACM; 2013.

47. Booch Grady, Rumbaugh James, Jacobson Ivar. *The unified modeling language user guide.* Pearson Education India; 1999.

48. Cornelissen Bas, Holten Danny, Zaidman Andy, Moonen Leon, Van Wijk Jarke J, Van Deursen Arie. Understanding execution traces using massive sequence and circular bundle views. In: :49–58IEEE; 2007.

49. Fittkau Florian, Krause Alexander, Hasselbring Wilhelm. Software landscape and application visualization for system comprehension with ExplorViz. *Information and Software Technology.* 2016;.

50. Berghe Alexander, Scandariato Riccardo, Yskout Koen, Joosen Wouter. Design notations for secure software: a systematic literature review. *Software & Systems Modeling.* 2015;:1–23.

51. Artho Cyrille, Havelund Klaus, Honiden Shinichi. Visualization of concurrent program executions. In: :541–546IEEE; 2007.

52. Systä Tarja, Koskimies Kai, Müller Hausi. Shimba—an environment for reverse engineering Java software systems. *Software: Practice and Experience.* 2001;31(4):371–394.

53. Systa Tarja. On the relationships between static and dynamic models in reverse engineering java software. In: :304–313IEEE; 1999.

54. Nielson Flemming, Nielson Hanne R, Hankin Chris. *Principles of program analysis.* Springer; 2015.

55. Gosain Anjana, Sharma Ganga. Static Analysis: A Survey of Techniques and Tools. In: Springer 2015 (pp. 581–591).

56. Grass Judith E.. Object-oriented design archaeology with CIA++. *Computing Systems.* 1992;5(1):5–67.

57. Ball Thomas. The concept of dynamic analysis. In: :216–234Springer; 1999.

58. Labiche Yvan, Kolbah Bojana, Mehrfard Hossein. Combining Static and Dynamic Analyses to Reverse-Engineer Scenario Diagrams. In: :130–139IEEE; 2013.

59. Lamprier Sylvain, Baskiotis Nicolas, Ziadi Tewfik, Hillah Lom-Messan. CARE: a platform for reliable Comparison and Analysis of Reverse-Engineering techniques. In: :252–255IEEE; 2013.

60. Molléri Jefferson Seide, Petersen Kai, Mendes Emilia. Survey Guidelines in Software Engineering: An Annotated Review. In: :58ACM; 2016.

61. Kitchenham Barbara. Procedures for performing systematic reviews. *Keele, UK, Keele University.* 2004;33(2004):1–26.

62. Greenalgh T. How to read a paper: Papers that summarise other papers: systematic reviews and meta-analysis. *British Medical Journal*. 1997;315:672–5.

63. Wohlin Claes. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: :38ACM; 2014.

64. Webster Jane, Watson Richard T. Analyzing the past to prepare for the future: Writing a literature review. *MIS quarterly*. 2002;:xiii–xxiii.

65. Gomaa Hassan. Designing concurrent, distributed, and real-time applications with UML. In: :737–738IEEE Computer Society; 2001.

66. Xie Shaohua, Kraemer Eileen, Stirewalt RE Kurt. Empirical evaluation of a UML sequence diagram with adornments to support understanding of thread interactions. In: :123–134IEEE; 2007.

67. Xie Shaohua, Kraemer Eileen, Stirewalt RE Kurt, Dillon Laura K, Fleming Scott D. Assessing the benefits of synchronization-adorned sequence diagrams: two controlled experiments. In: :9–18ACM; 2008.

68. Jürjens Jan. Towards development of secure systems using UMLsec. In: Springer 2001 (pp. 187–200).

69. Burd Elizabeth, Overy Dawn, Wheetman Ady. Evaluating using animation to improve understanding of sequence diagrams. In: :107–113IEEE; 2002.

70. Garousi Vahid, Briand Lionel C, Labiche Yvan. Control flow analysis of UML 2.0 sequence diagrams. In: :160–174Springer; 2005.

71. Wong Kenny, Sun Dabo. On evaluating the layout of UML diagrams for program comprehension. *Software Quality Journal*. 2006;14(3):233–259.

72. Fleming Scott, Kraemer Eileen, Stirewalt RE Kurt, Xie Shaohua, Dillon Laura. A study of student strategies for the corrective maintenance of concurrent software. In: :759–768IEEE; 2008.

73. Kraemer Frank Alexander. Engineering android applications based on UML activities. In: :183–197Springer; 2011.

74. Huang Jianjun, Zhang Xiangyu, Tan Lin, Wang Peng, Liang Bin. AsDroid: detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In: :1036–1046ACM; 2014.

75. Kowalczyk Emily. Modeling App Behavior from Multiple Artifacts. In: :385–386IEEE; 2016.

76. Alvin Chris, Peterson Brian, Mukhopadhyay Supratik. StaticGen: Static Generation of UML Sequence Diagrams. In: :173–190Springer; 2017.

77. Alalfi Manar H, Cordy James R, Dean Thomas R. Automated reverse engineering of UML sequence diagrams for dynamic web applications. In: :287–294IEEE; 2009.

78. Alimadadi Saba, Sequeira Sheldon, Mesbah Ali, Pattabiraman Karthik. Understanding JavaScript event-based interactions. In: :367–377ACM; 2014.

79. Amalfitano Domenico, Fasolino Anna Rita, Polcaro Armando, Tramontana Porfirio. The DynaRIA tool for the comprehension of Ajax web applications by dynamic analysis. *Innovations in Systems and Software Engineering*. 2014;10(1):41–57.

80. Myers Del, Storey Margaret-Anne, Salois Martin. Utilizing debug information to compact loops in large program traces. In: :41–50IEEE; 2010.

81. Guéhéneuc Yann-Gaël, Ziadi Tewfik. Automated reverse-engineering of UML v2.0 dynamic models. In: Citeseer; 2005.

82. Grose Timothy J, Doney Gary C, Brodsky Stephen A. *Mastering XMI: Java Programming with XMI, XML and UML*. John Wiley & Sons; 2002.

83. Rountev Atanas, Kagan Scott, Sawin Jason. Coverage criteria for testing of object interactions in sequence diagrams. In: :289–304Springer; 2005.

84. Sharp Richard, Rountev Atanas. Interactive exploration of UML sequence diagrams. In: :1–6IEEE; 2005.

85. Lu Lunjin, Kim Dae-Kyoo. Required behavior of sequence diagrams: Semantics and conformance. *ACM Transactions on Software Engineering and Methodology (TOSEM)*. 2014;23(2):15.

86. Martinez Liliana, Pereira Claudia, Favre Liliana. Recovering sequence diagrams from object-oriented code: An ADM approach. In: :1–8IEEE; 2014.

87. Roubtsov Serguei, Serebrenik Alexander, Mazoyer Aurélien, Brand Mark, Roubtsova Ella. I2SD: reverse engineering Sequence Diagrams Enterprise Java Beans from with interceptors. *IET software.* 2013;7(3):150–166.

88. Tonella Paolo, Potrich Alessandra. Reverse engineering of the interaction diagrams from C++ code. In: :159–168IEEE; 2003.

89. Zhu Hong, Hall Patrick AV, May John HR. Software unit test coverage and adequacy. *Acm computing surveys (csur).* 1997;29(4):366–427.

90. Biermann Alan W, Feldman Jerome A. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers.* 1972;100(6):592–597.

91. Carr Jacob Shannan, Kachmarck Brian Thomas. *Generating module stubs.* US Patent 9,117,177; 2015.

92. Grati Hassen, Sahraoui Houari, Poulin Pierre. Extracting sequence diagrams from execution traces using interactive visualization. In: :87–96IEEE; 2010.

93. Oechsle Rainer, Schmitt Thomas. JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java debug interface (JDI). In: Springer 2002 (pp. 176–190).

94. Hoorn André, Rohr Matthias, Hasselbring Wilhelm, et al. Continuous monitoring of software services: Design and application of the Kieker framework. In: Kiel University, Germany; 2009.

95. Koskimies Kai, Mossenbock Hanspeter. Scene: Using scenario diagrams and active text for illustrating object-oriented programs. In: :366–375IEEE; 1996.

96. Souder Tim, Mancoridis Spiros, Salah Maher. Form: A framework for creating views of program executions. In: :612IEEE Computer Society; 2001.

97. Watanabe Yui, Ishio Takashi, Ito Yoshiro, Inoue Katsuro. Visualizing an execution trace as a compact sequence diagram using dominance algorithms. *Program Comprehension through Dynamic Analysis.* 2008;:1.

98. Taniguchi Koji, Ishio Takashi, Kamiya Toshihiro, Kusumoto Shinji, Inoue Katsuro. Extracting sequence diagram from execution trace of Java program. In: :148–151IEEE; 2005.

99. Ishio Takashi, Watanabe Yui, Inoue Katsuro. AMIDA: A sequence diagram extraction toolkit supporting automatic phase detection. In: :969–970ACM; 2008.

100. Dugerdil Philippe, Repond Julien. Automatic generation of abstract views for legacy software comprehension. In: :23–32ACM; 2010.

101. Maqbool Onaiza, Babri Haroon. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering.* 2007;33(11).

102. Jiang Juanjuan, Koskinen Johannes, Ruokonen Anna, Systa Tarja. Constructing usage scenarios for API redocumentation. In: :259–264IEEE; 2007.

103. Ziadi Tewfik, Da Silva Marcos Aurélio Almeida, Hillah Lom-Messan, Ziane Mikal. A fully dynamic approach to the reverse engineering of UML sequence diagrams. In: :107–116IEEE; 2011.

104. Maoz Shahar, Harel David. On tracing reactive systems. *Software & Systems Modeling.* 2011;10(4):447–468.

105. Sag Muhammet Ali, Tarhan Ayça. Measuring COSMIC software size from functional execution traces of Java business applications. In: :272–281IEEE; 2014.

106. Noda Kunihiro, Kobayashi Takashi, Toda Tatsuya, Atsumi Noritoshi. Identifying Core Objects for Trace Summarization Using Reference Relations and Access Analysis. In: :13–22IEEE; 2017.

107. Lo David, Maoz Shahar, Khoo Siau-Cheng. Mining modal scenario-based specifications from execution traces of reactive systems. In: :465–468ACM; 2007.

108. Lo David, Maoz Shahar. Specification mining of symbolic scenario-based models. In: :29–35ACM; 2008.

109. Malnati Giovanni, Cuva Caterina Maria, Barberis Claudia. JThreadSpy: teaching multithreading programming by analyzing execution traces. In: :3–13ACM; 2007.

110. Malloy Brian A, Power James F. Exploiting UML dynamic object modeling for the visualization of C++ programs. In: :105–114ACM; 2005.

111. Dalton Andrew R, Hallstrom Jason O. A toolkit for visualizing the runtime behavior of TinyOS applications. In: :43–52IEEE; 2008.

112. Toda Tatsuya, Kobayashi Takashi, Atsumi Noritoshi, Agusa Kiyoshi. Grouping objects for execution trace analysis based on design patterns. In: :25–30IEEE; 2013.

113. Ng Janice Ka-Yee, Guéhéneuc Yann-Gaël, Antoniol Giuliano. Identification of behavioural and creational design motifs through dynamic analysis. *Journal of Software: Evolution and Process.* 2010;22(8):597–627.

114. Guéhéneuc Yann-Gaël, Antoniol Giuliano. DeMIMA: A multilayered approach for design pattern identification. *IEEE Transactions on Software Engineering.* 2008;34(5):667–684.

115. Srinivasan Madhusudan, Yang Jeong, Lee Young. Case studies of optimized sequence diagram for program comprehension. In: :1–4IEEE; 2016.

116. Brand Mark, Roubtsov Serguei, Serebrenik Alexander. SQuAVisiT: A flexible tool for visual software analytics. In: :331–332IEEE; 2009.

117. EJB 3.1 Expert Group: EJB 3.1 Expert Group. Interceptors 1.1 2009; .

118. Smith Temple F, Waterman Michael S. Identification of common molecular subsequences. *Journal of molecular biology.* 1981;147(1):195–197.

119. Mössenböck Hanspeter, Wirth Niklaus. The programming language Oberon-2. *Structured Programming.* 1991;12(4):179–196.

120. Mäkinen Erkki, Systä Tarja. MAS—an interactive synthesizer to support behavioral modelling in UML. In: :15–24IEEE Computer Society; 2001.

121. The Eclipse Foundation, Atlas Transformation Language https://eclipse.org/atl/Accessed on 2015-04-22; .

122. LogSequencer 1.0 http://logsequencer.soft32.comAccessed on 2016-12-22; .

123. Matsumura Toshinori, Ishio Takashi, Kashima Yu, Inoue Katsuro. Repeatedly-executed-method viewer for efficient visualization of execution paths and states in java. In: :253–257ACM; 2014.

124. Harel David, Kleinbort Asaf, Maoz Shahar. S2A: A compiler for multi-modal UML sequence diagrams. *Fundamental Approaches to Software Engineering.* 2007;:121–124.

125. ASM Home page http://asm.objectweb.org/Accessed on 2017-09-05; .

126. Müller Hausi A, Tilley Scott R, Wong Kenny. Understanding software systems using reverse engineering technology perspectives from the Rigi project. In: :217–226IBM Press; 1993.

127. Koskimies Kai, Systa Tarja, Tuomi Jyrki, Mannisto Tatu. Automated support for modeling OO software. *IEEE software.* 1998;15(1):87–94.

128. Lee Han Bok, Zorn Benjamin G. BIT: A Tool for Instrumenting Java Bytecodes. In: :73–82; 1997.

129. Hamou-Lhadj Abdelwahab, Braun Edna, Amyot Daniel, Lethbridge Timothy. Recovering behavioral design models from execution traces. In: :112–121IEEE; 2005.

130. Ghaleb Taher Ahmed. The role of open source software in program analysis for reverse engineering. In: :1–6IEEE; 2016.

131. Hudson Scott E, Flannery Frank, Ananian C Scott, Wang D, Appel AW. *CUP LALR parser generator for Java.* Accessed on 28-12-2016; 1999.

132. Nystrom Nathaniel, Clarkson Michael R, Myers Andrew C. Polyglot: An extensible compiler framework for Java. In: :138–152Springer; 2003.

133. Shahin Mojtaba, Liang Peng, Babar Muhammad Ali. A systematic review of software architecture visualization techniques. *Journal of Systems and Software.* 2014;94:161–185.

134. Tsantalis Nikolaos, Chatzigeorgiou Alexander, Stephanides George, Halkidis Spyros T. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering.* 2006;32(11):896–909.