Enhanced Visualization of Method Invocations by Extending Reverse-engineered Sequence Diagrams

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

Khalid Aljasser Musab A. Alturki
Information & Computer Science Department
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia
{aljasser,musab.alturki}@kfupm.edu.sa

***Abstract*—Software maintainers employ reverse-engineered sequence diagrams to visually understand software behavior, especially when software documentation is absent or outdated. Much research has studied the adoption of reverse-engineered sequence diagrams to visualize program interactions. However, due to the forward-engineering nature of sequence diagrams, visualizing more complex programming scenarios can be challenging. In particular, sequence diagrams represent method invocations as unidirectional arrows. However, in practice, source code may contain compound method invocations that share values/objects implicitly. For example, method invocations can be *nested*, e.g., `fun(foo())`, or *chained*, e.g., `fun().foo()`. The standard notation of sequence diagrams does not have enough expressive power to precisely represent compound scenarios of method invocations. Understanding the flow of information between method invocations simplifies debugging, inspection, and exception handling operations for software maintainers. Despite the research invested to address the limitations of UML sequence diagrams, previous approaches fail to visualize compound scenarios of method invocations. In this paper, we propose sequence diagram extensions to enhance the visualization of (i) three widely used types of compound method invocations in practice (i.e., *nested*, *chained*, and *recursive*) and (ii) *lifelines of objects returned* from method invocations. We aim through our extensions to increase the level of abstraction and expressiveness of method invocation code. We develop a tool to reverse engineer compound method invocations and generate the corresponding extended sequence diagrams. We evaluate how our proposed extensions can improve the understandability of program interactions using a controlled experiment. We find that program interactions are significantly more comprehensible when visualized using our extensions.**

***Index Terms*—Sequence diagram, extended notation, program comprehension, method invocation, controlled experiment**

I. Introduction

Sequence diagrams allow software maintainers to get a visualized outlook of program interactions. Software maintainers use reverse-engineered sequence diagrams for legacy systems or when software documentation is absent, poor, or outdated. Much research has studied the adoption of reverse-engineered sequence diagrams to visualize program interactions [1]–[37]. However, due to the forward-engineering nature of sequence diagrams, visualizing more complex programming scenarios can be challenging. For example, sequence diagrams represent method invocations as unidirectional arrows. However, in practice, source code may contain compound method invocations, in which values/objects are communicated between callers and callees implicitly. For example, method invocations can be nested, e.g., `fun(foo())`, or chained, e.g., `fun().foo()`. The standard primitives of UML sequence diagrams do not have enough expressive power to precisely represent such scenarios of method invocations.

Prior research has assessed the use of reverse-engineered sequence diagrams for program understanding [38]. Previous studies proposed (i) extensions to sequence diagrams to support more control flow scenarios [2], [9], [12], [39]–[44] and (ii) non-standard forms of visualization of program interactions, such as Circular Bundles [45], City Metaphor [46], and Markov Chains & Timing Diagrams [12]. Still, previously proposed solutions fail to visualize compound scenarios of method invocations. Understanding the information flow between compound method invocations enables software maintainers to perform debugging, inspection, and exception handling operations.

In this paper, we propose extensions to sequence diagram to enhance the visualization of compound method invocations in Java. In particular, our proposed extensions represent (a) three types of compound method invocations (i.e., nested, chained, and recursive calls) and (b) lifelines that correspond to objects returned from method invocations. We choose such types of compound method invocations as they are widely used in practice. We aim through our extensions to increase the level of abstraction and expressiveness of method invocation code. We choose to extend the standard notation rather than using another alternative notation to make our approach more interoperable. We use typical code examples of method invocations to demonstrate the notation of our extensions. We use our tool for reverse-engineering software systems [47] to generate the extended representation of sequence diagrams used in this study. Moreover, we conduct a controlled experiment to evaluate how our proposed extensions can improve the understandability of program interactions at the method level. Our results show that the communications between program methods/objects are more comprehensible when visualized using our extensions.

In summary, this paper makes the following contributions:

- Highlights on the limitations of reverse-engineered UML sequence diagrams in representing program interactions.

Preprint..

- Novel sequence diagram extensions for enhanced visualization of compound scenarios of method invocation.
- A detailed demonstration of the proposed extensions using common programming scenarios.
- A controlled experiment to evaluate the effectiveness of the proposed sequence diagram extensions for program understandability, in comparison with the (baseline) standard UML notation of sequence diagrams.

Paper Organization: The rest of this paper is organized as follows. Section II presents background on sequence diagrams and program comprehension. Section III provides a detailed description of our proposed extensions to sequence diagrams. Section IV describes our controlled experiment to evaluate the proposed sequence diagram extensions for program comprehension. Section V discusses our experimental results. Section VI discusses validity threats to our results. Section VII presents some relevant studies in supporting program comprehension through reverse engineering. Finally, Section VIII concludes the paper and discusses the possible future work.

II. Background

This section presents some background about reverse-engineered sequence diagrams and program comprehension.

A. Program interactions as sequence diagrams

The Unified Modeling Language (UML 2.0 [48]) is the *de facto* standard for modeling software behavior using sequence diagrams. Reverse-engineered sequence diagrams can be derived from existing source code using static or dynamic analysis techniques [38]. They give insights of the software behavior, which help software maintainers understand how system objects interact with each other.

Program interactions in imperative programming languages are represented using method invocations. The UML notation of sequence diagrams supports representing intraprocedural control flow of programs. Despite the added features to UML sequence diagrams (e.g., combined fragments),[1] the existing notation still has limitations that may lead to imprecise or wrong representations of program interactions. Such limitations have urged prior studies to introduce more creative solutions to visualize complex program interactions.

B. Program comprehension

Program comprehension is the activity of understanding the static and dynamic aspects of software systems, namely the structure and behavior [49]. Program visualization tools play a vital role in this regards. Program visualizations display various aspects of program structure or behavior to reduce source code manual navigation. Prior research has evaluated the effectiveness of visualization techniques towards program comprehension using controlled experiments. Controlled experiments maintain a set of tasks that are related to program comprehension activities (e.g.,maintenance) [50]. Controlled experiments are widely used to measure (i) the time invested by users to respond to the predefined comprehension tasks and (ii) the correctness of user responses [49], [51]–[53]. Bennett *et al.* [54] conducted an evaluation of the actual use of features provided by the tools that employ reverse-engineered sequence diagrams for representing program interactions. Xie *et al.* [39], [55] conducted an empirical evaluation of UML sequence diagrams with an extended notation for thread interactions. However, such experiments have not assessed the positive impact of their extended sequence diagrams towards program comprehension using forward-engineered, rather than reverse-engineered, sequence diagrams.

[1] https://www.uml-diagrams.org/sequence-diagrams-combined-fragment.html)

III. The Proposed Extensions to Sequence Diagrams

In this section, we present our proposed sequence diagram extensions. The design principles of our proposed extensions include: (a) the identification of the most important limitations of sequence diagrams that hinder program comprehension; (b) targeting most commonly used program features; (c) maintaining simplicity while maximizing effectiveness for program comprehension; (d) minimizing the gap with the standard notation (i.e., reusing some notational components while alleviating ambiguity); (e) introducing flexibility to handle complex interactions; and (f) maintaining a design that compatible to the standard sequence diagram rather than as a completely separate tool.

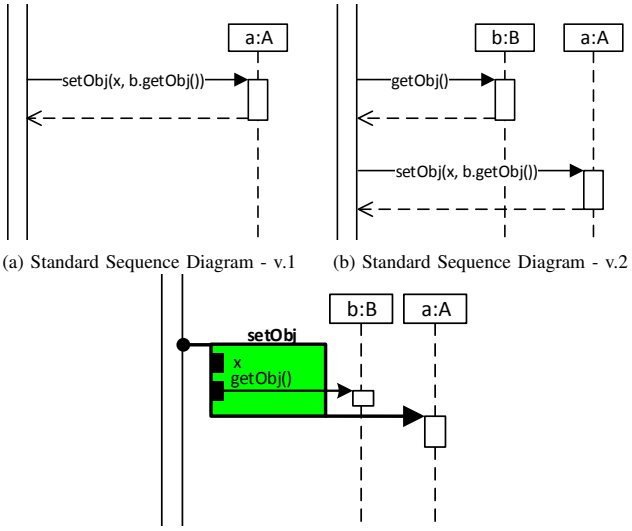
We propose four sequence diagram extensions that capture the compound scenarios of method invocations in Java, namely nested calls (III-A), chained calls (III-B), recursive calls (III-C), and returned objects (III-D). We use typical examples to describe each extension. Our extension allow more complex scenarios in which different types of method invocations may overlap (e.g., nested of nested calls or chains of nested calls). More examples of complex scenarios of our extensions can be found in our online appendix [56].

A. Nested method calls

Method calls may compose of nested calls passed as parameters. Program execution of such cases starts with invoking the methods passed as parameters. After that, the program invokes the original method call. Invoking methods passed as parameters can either be in a left-to-right or right-to-left order. In our case, since we use Java, left-to-right associativity is considered. The following statement has two nested calls: a call to the `getObj` method of object `b` is nested with the call to the `setObj` method of object `a`;

```
a.setObj(x, b.getObj());
```

The standard sequence diagram represents the above programming statement using separate unidirectional, horizontal arrows as shown in Fig. 1a and Fig. 1b (tool-dependent). Although the first standard representation (Fig. 1a) simplifies the whole statement using a single message arrow, it does not reflect the actual number of interactions. The other representation (Fig. 1b) shows the exact number of interactions, but does not precisely reflect the actual interactions between methods/objects.



(a) Standard Sequence Diagram - v.1 (b) Standard Sequence Diagram - v.2

(c) Extended Sequence Diagram

Fig. 1: Nested calls notation

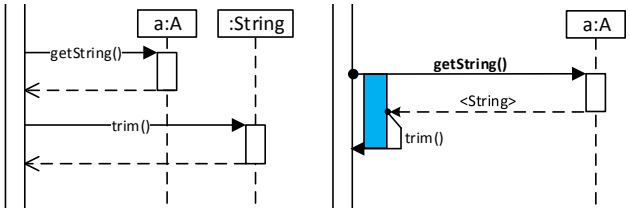
To resolve such a confusion, we propose a novel sequence diagram extension that distinguishes this kind of method calls from the other kinds of calls. This extension provides an extended component to the standard notation of sequence diagrams, as shown in Fig. 1b. This notation reflects the actual communication of information with the exact number of interactions. As shown in the figure, the method `setObject` is firstly called but lastly executed. Also, the parameter `x` does represent an interaction while the second parameter, the call to the `getObj` method, represents a nested interaction that is executed inside and before its enclosing method. This methodology is also applicable to the interactions of the `return` and `new` statements in case they involve method calls within their parameters.

B. Chained method calls

It is common in object-oriented programming to see what is called chained calls. Chained calls are the set of calls that depend on each other in their execution. This means that the object needed to call one method is returned by its preceding method call. As an example, we provide the following chain of calls:

```
a.getString().trim();
```

We observe that the method `getString` of the object `a` will be called and will return an anonymous object of type `String`, which will be used to call the method `trim`. The standard sequence diagram deals with such calls separately, which means that the diagram will show that these two calls are independent and do not depend on each other (as shown in Fig. 2a). However, it shows the exact number of interactions, but take in mind that some tools represent the whole statement using one message while some others only represent the first method call of the chain.



(a) Standard Sequence Diagram (b) Extended Sequence Diagram

Fig. 2: Chained calls notation

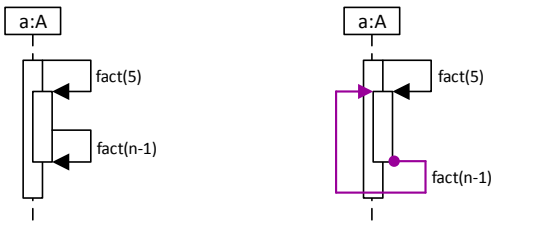
We address this scenario by providing an extended notation that can precisely represent the chain of the calls and the dependence of one call on another. As demonstrated in Fig. 2b, the extended notation is intuitively expressive and reflects the actual flow of communication between methods and the exact number of interactions.

C. Recursive method calls

Recursive calls are repetitive calls that make a sequence of commands executing multiple times. Unlike loops, there is no specific control structure in programming languages that can indicate the presence of recursive calls. Still, recursive calls can be captured through a careful static analysis of the program source code. Recursion may not only occur by a call from the method itself, but it can also occur between different methods calling each other repetitively. For example, if $method_1$ calls itself, then a direct recursion happens. Also, if $method_1$ calls $method_2$, and then $method_2$ calls $method_1$; then this is also considered as an indirect recursion. Let's take the following example:

```
void fact(int n){
    fact(n-1);
}
main(){
    fact(5);
}
```

The $fact$ method is recursive as it calls itself (assuming that we are inside an object a of class A). In the standard representation shown in Fig. 3a, the recursive call is represented as a self-message, whereas self-messages may represent calling methods of the same lifeline. Our extended notation shown in Fig. 3b expressively depicts such a recursive call so that users can be recognized directly from the first sight.



(a) Standard Sequence Diagram (b) Extended Sequence Diagram

Fig. 3: Recursive calls notation

D. Objects returned from method calls

Objects in Java are created by (i) using `new` statements or (ii) calling methods. The UML sequence diagram uses lifelines to represent objects but does not show when such objects are created. We propose to extend sequence diagrams capture object creation using the aforementioned ways. Our object creation extension does not introduce new notational components to sequence diagrams but rather utilizes the existing UML components (i.e., lifelines and arrows). Fig. 4 shows interactions carried out in the following statement using our proposed extension:

```
A a2 = a1.getObj();
a2.getVal();
```

We observe that the returned value of the invoked method `getObj` of the object `a1` creates the lifeline `a2`, which is then used to call the method `getVal`. On the other hand, in the standard sequence diagram, it is not clear how and through which integration the object `a2` was created.

IV. Experimental Evaluation

We define different comprehension tasks that we aim to use for measuring the added valued by our proposed sequence diagram extensions to improve program comprehension of method calls. We use the Greenfoot[2] Java project as a case study for our experiment. Greenfoot includes various method call scenarios that cover most of our extensions.

A. Research questions and hypotheses

Based on our selected case study, we define the following research questions:

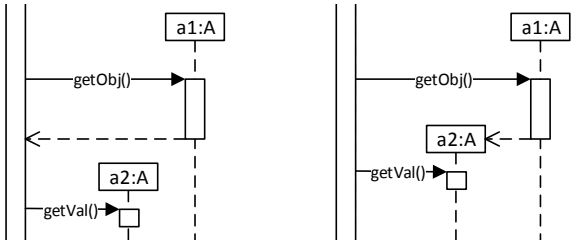
1) Does the availability of our proposed extensions to the sequence diagram reduce the time that is needed to achieve the comprehension tasks?
2) Does the availability of our proposed sequence diagram extensions increase the correctness of the answers provided during those tasks?
3) Is representing programs using our proposed sequence diagram extensions less complex and more precise than that with the use of the standard sequence diagram?

We associate the first three research questions with three null hypotheses, formulated as follows:

- $H1_0$: The use of our proposed sequence diagram extensions does not affect the time needed to complete each comprehension task.
- $H2_0$: The use of our proposed sequence diagram extensions does not affect the correctness of responses given during those tasks.
- $H3_0$: Our proposed extensions to the sequence diagram represent programs in a more complex and imprecise way.

After that, we have stated the alternative hypotheses used in the experiment, as follows:

[2]https://www.greenfoot.org



(a) Standard Sequence Diagram (b) Extended Sequence Diagram

Fig. 4: Notation representing lifelines of returned objects

- $H1$: The use of our proposed sequence diagram extensions decreases the time needed to complete each comprehension task.
- $H2$: The use of our proposed sequence diagram extensions improves the correctness of answers given during those tasks.
- $H3$: Our proposed sequence diagram extensions precisely represent programs using a simple (i.e., not complex) notation.

The first alternative hypothesis is motivated by the fact that our sequence diagram extensions explicitly demonstrate method call scenarios of the subject system. UML standard of sequence diagrams, on the other hand, require participants to implicitly infer method call scenarios of the system. The rationale behind the second alternative hypothesis is the inherent precision of our notational components used to differentiate between the various method call scenarios. Such a hypothesis results in a deeper understanding of program interactions at the method level and, hence, more accurate answers. The third alternative hypothesis is induced by the way and style our extensions are represented. The design of the style of the extended in intended to reflect the actual flow of information in programs using the least number and size of components.

To test the $H1_0$, $H2_0$, and $H3_0$ hypotheses, we define a set of comprehension tasks. Such tasks are implemented by both a control group and experimental group. The control group uses standard UML sequence diagrams, whereas the experimental group uses sequence diagrams supplied with our proposed extensions. A between-subjects design is maintained to allow each subject to be either in the control group or in the experimental group.

B. The object of the experiment

The system that our experiment is based on is Greenfoot, a Java environment that simplifies the development of two-dimensional graphical applications and is meant for educational purposes of programming languages. Generating reverse-engineered sequence diagrams for the overall functionality of Greenfoot will for sure result in obtaining more complex and disappointing diagrams for the subjects to achieve the tasks. Therefore, we have selected only a specific scenario of Greenfoot used for browsing classes. This scenario is based on a class called $ClassBrowser$, which is responsible for drawing and laying out the classes on the user interface. The

resulting diagrams contain more than 50 method calls between around 20 objects/classes.

We choose Greenfoot as our experimental object for the following reasons:

- Greenfoot is an open source software project. The availability of source code also helps in verifying and replicating of the experiment conducted in this paper [57].
- Greenfoot is a modular software system, which enables to perform the analysis and modeling of its method invocation scenarios easily.
- The selected scenario of Greenfoot for our case study encompasses all types of method call scenarios supported by our sequence diagram extensions.

We use the Visual Paradigm[3] tool to produce the reference reverse-engineered sequence diagram using the UML standard. In addition, we use our tool [47], [58], [59] to reverse engineer the extended sequence diagram. Both diagrams contain the same number of lifelines but different message lines between lifelines. However, our extended diagram generates lifelines on demand lifelines (i.e., at the place where a specific class/object is used). We exported both diagrams into PDF files to allow participants to search for certain terms or to zoom in/out while responding to the tasks. Details about the reverse-engineered diagrams and the experimental tasks used in our study can be found in our online appendix [56].

C. Task design

Prior controlled experiments for program comprehension employed a comprehension framework proposed by Pacione *et al.* [50], who classified the comprehension tasks of software visualization into nine primary activities. However, we find that strictly following such a framework may not expose all the capabilities of our proposed extensions. Therefore, we use different question types in our our tasks. Each type of questions requires a different kind of user input.

1) Category C1: Searching for the number or names of certain program components:

- **Task T1 (Recursive Calls)**: Write the name(s) of all recursive methods, if any?

2) Category C2: Writing code representing a certain sub-diagram:

- **Task T2.1 (Chained Calls)**: Write the Java code that corresponds to the excerpt of a sequence diagram (see Appendix [56]:T2.1)?
- **Task T2.2 (Chained Calls + Nested Calls)**: Write the Java code that corresponds to the excerpt of a sequence diagram (see Appendix [56]:T2.2)?
- **Task T2.3 (Lifelines of returned objects)**: Write the Java code that corresponds to the excerpt of a sequence diagram (see Appendix [56]:T2.3)?

[3]https://www.visual-paradigm.com

3) Category C3: Snapping the sub-diagram representing some certain code:

- **Task T3 (Multi-Nested Calls)**: Identify and screenshot the portion of the sequence diagram that reflects the following code:

```
cb.quickAddClass(newClassView(
cb,newGCoreClass(Actor.class,project)));
```

4) Category C4: Rating diagrams produced using the standard and extended diagrams for the same method call scenarios.

- **Task T4.1 (Chained Calls)**: Rate the Complexity and Precision of a sub-diagram (see Appendix [56]:T4.1) in representing the following code:

```
this.getRootPane().revalidate();
```

- **Task T4.2 (Nested Calls)**: Rate the Complexity and Precision of the sub-diagram (see Appendix [56]:T4.2) in representing the following code:

```
BorderFactory.createTitledBorder(null,
Config.getString("BBworld"));
```

- **Task T4.3 (Recursive Calls)**: Rate the Complexity and Precision of the sub-diagram in (see Appendix [56]:T4.3) in representing the following code:

```
void createClassHierarchyComponent(
Collection roots, boolean isRecursiveCall) {
 createClassHierarchyComponent(children,true);}
```

For tasks of categories C1, C2, and C3, we use open-ended questions in our tasks to make it harder for participants to guess the answers, which generates more reliable and representative comprehension situations. Feedback obtained from the tasks of the category C4 is not graded further, since such tasks already expect rating values by participants. A single evaluator awarded points to the answers to ensure a uniform and fair grading based on a solution model.

D. The subjects of the experiment

The subjects in this experiment are 8 PhD candidates, 12 MS students, and 16 BS senior students. The PhD and MS students were in the same program at the computer science department. The resulting number of subjects is of 36 subjects. Subjects are from 7 different different nationalities and are working on different areas of computer science and software engineering. All subjects have prior experience with the UML sequence diagram but none of them has had previous knowledge about our proposed extensions. The participation in the experiment was completely voluntarily.

We distribute the subjects based on their knowledge of Java, software modeling, sequence diagrams and reverse engineering. Considering that all undergraduate students were working on senior projects in software engineering, they were just evenly distributed into two groups of eight students, one as control and another as experimental. MS and PhD students were distributed based on their experience in software engineering. We measured participants' experience by the kind and number of courses

they have taken at the software engineering program. We asked informal questions to each subject to assess the experience with sequence diagrams. As a result, MS and PhD students were evenly assigned to the groups (i.e., four PhD and six MS students per each group). In total, each group consisted of 18 students: four PhD, six MS and eight BS students.

E. Experimental procedure

We conduct our experiment through two sessions, each of which has taken place at a computer lab in the computer science department. Both sessions were conducted on workstations with the same Internet connection and specifications, i.e., all of them are of Intel Core i3 - 2.93 GHz CPUs, 4 GB RAM, and screen resolutions of 1440 x 900. The first session involved the MS and PhD students of both groups, whereas the second session was for the BS students. A 5-minute recall tutorial on sequence diagrams was given to both groups, highlighting our proposed extensions, and how can they reflect Java code. In addition, we conducted a 10-minutes presentation showing our proposed extensions to the standard sequence diagram. Both sessions were supervised, allowing the subjects to pose clarification questions and preventing them from communication with each other. We have been requiring subjects to motivate their answers at all times. Subjects were encouraged to take a short break if they started to get bored or confused.

F. Variables and analysis

The availability of our extensions for the sequence diagram notation in the experiment is regarded as the independent variable to the UML sequence diagram during all the tasks.

The first dependent variable is the ***time spent*** on each task and is measured by recording the the time a user spent on each task. In addition, we disabled the 'Back' button on each page to prevent the subjects from navigating back to earlier tasks. The second dependent variable is the ***correctness*** of the given answers. We measured the correctness of answers using a model answer that associates scores to each expected answer. Two of the authors assessed the correctness of the answers. Then, we resolved any disagreements using an open discussion with the third author.

To test our hypotheses, we first tested the sample distributions using the $Kolmogorov - Smirnov$ test [60] to see whether they are normal. In addition, we used the $Levene$'s test [61] to check whether the sample distributions have equal variances. In the cases where statistical tests passed successfully, the Student's t-test was used to evaluate the hypotheses. Following our alternative hypotheses, we employed a one-tailed variant of each statistical test. For the time as well as the correctness variables, a typical confidence level of 95% was maintained (α = 0.05).

G. Pilot studies

Before conducting the experimental sessions, we carried out two pilot studies to refine several experimental parameters, such as the number and kind of tasks, their feasibility, clarity, and the amount of time would be required. The pilots for the control and experimental groups were performed by one BS, one PhD, and two MS students at the computer science department. Pilots were also given tutorial about UML sequence diagrams and our proposed extensions. Pilots have not participated in the actual experiment. The results of the pilots helped us to (i) eliminate three complicated and time-consuming tasks, (ii) change the categories of two tasks, (iii) make the remaining tasks clearer and easier to understand, and (iv) improve our tutorial.

V. Experimental Results

Table I presents a set of descriptive statistics of the questionnaire results based on aggregated measurements over the eight tasks, which are basically based on grading the answers of participants and the time spent on each task.

Based on the individual results of each task, we observe that our data has no outliers to be removed. However, as a key factor for both time and correctness, we have noticed that two subjects (one from each group) were not very interested in conducting the questionnaire as we have noticed that they did not respond to the provided tasks properly. For example, one of them has written some zeros as responses for some of the tasks of the category C2 that required writing code, while the other has entered similar rating values for all both criteria and both diagrams in the tasks of the category C4. Subsequently, we disregarded the entire input provided by these two particular subjects (i.e., we ended up with having responses of 17 subjects from the control group and 17 subjects from the experimental group).

A. Results of the time spent on tasks

We have started by testing the null hypothesis $H1_0$ described in section IV-A that stated that the time needed to complete comprehension tasks is not impacted by the availability of our proposed sequence diagram extensions. Fig. 5a shows the total time spent by the subjects on the first eight tasks using a box plot. It can be also indicated from Table I that, on average, extended diagram group required 25.20 percent less time.

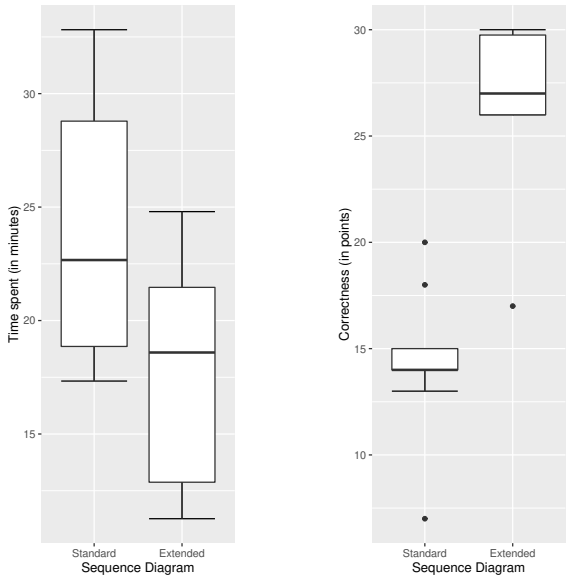
The distributions of the samples are normal and they have equal variances as well. This has been was proven by the Kolmogorov-Smirnov and Levene tests, which have succeeded for the timing results shown in Table I). This concludes that Student's t-test can be used to test $H1_0$. As presented in Table I, a statistically significant result has been yielded from the t-test, which is represented by the p-value of 0.0237 that is less than 0.05. The average time spent by the extended sequence diagram group was visibly lower, which means that $H1_0$ can be rejected in support of the alternative hypothesis $H1$, implying that the use of our sequence diagram extensions could decrease the time needed to achieve different comprehension tasks.

B. Reasons for different time requirements

There are several factors that contributed to the lower time requirements for the extended sequence diagram participants. First, most of program interactions are explicitly represented using special and expressive notation, which helps in finding certain information by just having an outlook to the provided

TABLE I: Computed statistics of the questionnaire results

	Time (in minutes)		**Correctness (in points)**	
	Standard Sequence Diagram	*Extended Sequence Diagram*	*Standard Sequence Diagram*	*Extended Sequence Diagram*
Mean	23.81	17.81	14.40	26.80
Difference		-25.20%		+86.11%
Min	17.33	11.26	7	17
Max	32.82	24.80	20	30
Median	21.45	18.59	14	27
Stdev.	5.72	5.10	3.37	3.85
Kolmogorov-Smirnov	0.594	0.597	0.070	0.005
Levene F		0.6405		0.7525
Student's t-test				
df		17.76		17.69
t		2.47		-7.66
p-value		0.0237		0.0001



(a) Time spent (b) Correctness

Fig. 5: Box plots for the overall spent time and correctness

diagram. Participants who used the standard UML sequence diagram, on the other side, tended to look for certain pointers that might assist them inferring the locations of certain program information. Second, as most of the program information were either not, wrongly or inappropriately presented in the standard sequence diagram, participants tended to search for answers to the questions even more than once in some portions of the diagram, which for sure results in having a cognitive load.

On the other hand, there might be several factors that led to having a negative impact on the time requirements of the participants who used the extended sequence diagrams. The main important factor is the unfamiliarity of these extensions to the participants as it was the first time for participants to see such extensions. This has led to having the participants requesting a copy of the tutorial presented while they were conducting the questionnaire. Therefore, referring to the tutorial for every particular sequence diagram extension in some of the questions contributed to spending a certain amount of time as overhead for recalling its meaning. This could be solved by incorporating the proposed extensions into standard UML as well as the tool that generate it.

C. Results of the correctness of answers

We test the null hypothesis $H2_0$, which states that the use of our sequence diagram extensions does not affect the accuracy of the answers given by participants during the comprehension tasks.

Fig. 5b demonstrates the points obtained by the subjects on the first eight tasks by means of a box plot. Notice that we take into consideration the overall points rather than individual ones (points per task are discussed in subsections V-E and V-F). The correctness difference is obviously seen from the box plot, and is even more pronounced than that for the timing results. Answers provided by the extended diagram-based subjects were more accurate by 86.11 percent (refer to Table I), that is obtained through averaging 26.8 out of 32 points compared to 14.40 points for the standard diagram group. Similar to the timing results, Table I also shows the results of the Student's t-test for response correctness, in which the requirements for the use of the t-test were met as well. The p-value of 0.0001 implies statistical significance, which means that $H2_0$ can be rejected in support of our alternative hypothesis $H2$, which states that the availability of our sequence diagram extensions improve the correctness of answers provided throughout the conducted comprehension tasks. Such results also imply that $H3_0$ can be rejected, since obtaining precise answers indicates the simplicity of our extensions.

D. Reasons for response accuracy differences

We regard the added value of our proposed extensions for correctness to several factors. The design of our extensions expresses the code behind them. Participants were confident and thus able to capture the correct answer of most of the provided questions. Finally, the questionnaire results presented in Table I shows that the extended sequence diagram group utilized their allotted diagram most of the time. However, in some tasks, participants could provide correct answers but spent a bit more time. Such results have been further confirmed by the ratings

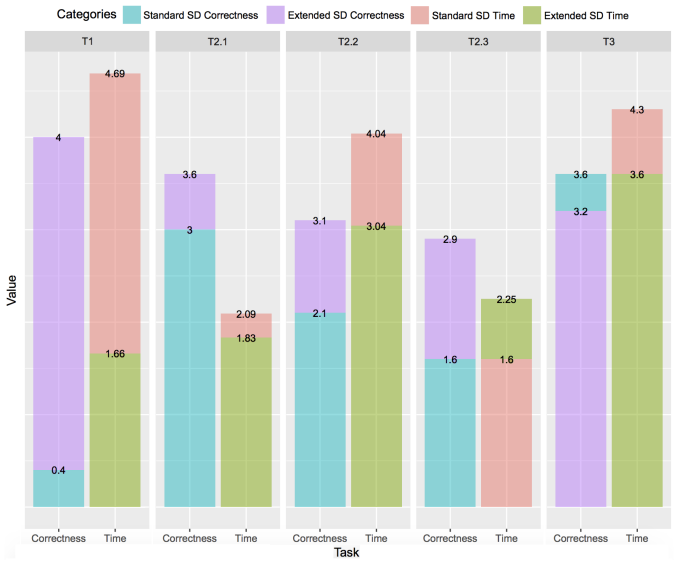


Fig. 6: Mean values of the correctness (in points) and the time spent (in minutes) of the tasks of Categories C1, C2, and C3

of participants (Tasks of Category C4), which indicate that our extensions were precise while they were complex to some extent.

E. Performance of the tasks of Categories C1, C2, and C3

We examine the performance of the subjects per each task independently in more detail. Fig. 6 demonstrates the average time spent and points obtained by each group. Although our experiment composes eight tasks, only five tasks are considered in this evaluation (i.e., tasks of Category C4 have a separate evaluation discussed in subsection V-F). Looking at Fig. 6, we observe that, for the majority of the tasks (i.e., four out of five tasks), participants who used the extended diagram were able to answer questions faster than those who used the standard diagram. Moreover, the answers of participants who used the extended diagram were more accurate than those who used the standard diagram in four out of five tasks. Such results indicates that the standard diagram requires more time to understand program interactions and may eventually lead to inaccurate answers.

1) Task T1: Recursive calls: In this task, participants found it easier to capture recursive calls using our extended diagram. All experimental participants achieved the full points (i.e., 4 out of 4) and required less than half the time required by the control group. The main reason of having only two subjects out of eleven who could capture recursive methods is the fact that self and recursive messages are represented using the same notation in the UML standard of sequence diagrams. Therefore, such participants consumed too much time tracking all self-messages for the sake of identifying whether they are being executed or not. Nevertheless, some control participants responded that not recursive calls exist.

2) Task T2.1: Chained calls: Timing results of this task indicate that control participants spent less time in writing the code representing a chained call UML sequence diagram than experimental participants. However, answers of the experimental group obtained higher scores. This result may indicate that, despite the simplicity of the standard sequence diagram, it could lead to wrong interpretation of program interactions. Participants who used the extended diagram were able to recognize the correct flow of messages with the price of the time that was mostly spent on recalling the meaning of the new notation by referring to the provided tutorial.

3) Task T2.2: Chained Calls + Nested Calls: This task also requires writing a code snippet that generates the excerpt of the diagram. As the flow of messages here was relying on chained and nested calls, the diagram excerpts of both the standard and extended diagrams were somewhat complicated. However, participants of the experimental group could write the code faster and more precise than those of the control group. The mean time spent by the experimental group was about 1.0 minute less and the precision score was 1.0 point more than the control group.

4) Task T2.3: Lifelines of returned objects: In this task, we clearly observe that the time spent by the experimental group was greater than that spent by the control one. The diagram excerpt used for this task was fairly simple using both the standard and extended sequence diagrams. This caused the standard participants responding faster but, due to the limitation of the UML sequence diagram in creating lifelines for objects once they are returned from a method call, most participants could not recognize that the message provided in the excerpt returns an object to a named variable, which as a result led to wrong answers. On the other hand, the extended diagram participants were able to identify the returned object and could answer the question better but with the price of time spent.

5) Task T3: This is the only task that represents category C3. Here, subjects are provided with source code and requested to search for the portion of the supplied diagram representing that code snippet. Again, the diagram excerpt representing that code was relatively simpler and participants go catch quickly. However, we observe the significance of the time invested for performing this task compared with the other tasks, which is caused by having participants to search, snapshot, save the snipped image, and then upload that image as a response to this task. However, we observe that such time is less than that of the control group. In addition, we observe that answers provided by the experimental were less accurate in comparison with the standard group participants who achieved slightly better scores (only 0.3 more than the experimental group). While investigating the root cause of that, we observed that there was another excerpt of the diagram that is somehow identical to the one requested in this task. Everything was similar in those two excerpts except the name of one of the classes used as a parameter to one of the methods called.

F. Performance of the tasks of Category C4

In the tasks of Category C4, we ask participants to evaluate the complexity and precision of our extensions in comparison to the standard notation. To this end, we define the following two criteria:

- **Complexity:** this criterion measures how complicated a diagram is for understanding the actual flow of a given programming scenario, from the participants' point of view. This applies to both the standard and the extended notation of sequence diagrams.
- **Precision:** this criterion measures how precise a diagram is in representing a given programming scenario, from the participants' point of view. This involves the capability of the diagram to cover all aspects of the given code, such as the types of messages, the composition of different messages, and the interactions between lifelines.

Participants rate the provided diagrams of tasks T4.1, T4.2, and T4.3 using the aforementioned criteria in a scale of $0-4$. After collecting the results of all tasks of this kind, we have aggregated them using the median rather than the mean. This is because that the mean will not appropriately represent the overall complexity and criteria as it averages the inputs. Questions of Category C4 are asked to participants of both the control and experimental groups.

1) Task T4.1: Task T4.1 comparatively measures the complexity and precision of the standard and extended sequence diagrams in representing chained calls. In Fig. 7, we observe that our extended notation is 25% less complex than the standard notation in representing chained calls. In addition, we observe that representing chained calls using our extended notation is $1.3x$ more precise than the standard notation of sequence diagrams.

2) Task T4.2: We observe that the standard notation of sequence diagrams is nearly as twice complex as our extended notation in representing nested calls. Such a result indicates that the standard notation may complicate the interaction by generating multiple separate messages that relate to a single nested method invocation. Such a representation may lead users to mistakenly think of a different interaction calling scenario that may not reflect the actual program behavior. As a result, software maintainers may find it confusing to trace or inspect the code when using a standard sequence diagram.

3) Task T4.3: Our extended notation for recursive calls appears to significantly enhance the representation of recursive methods and their corresponding method invocations. At the same time, we observe that it is more precise than the standard notation as it renders the actual recursive flow of a given programming scenario.

4) Summary: After a deeper interpretation of the performance results obtained per each individual task, we analytically generalize our discussion. From the results obtained, we observe that our extensions to the standard sequence diagram are of great help in grasping the interactions executed throughout the program or even within a certain kind of control structures. In addition, our extensions help participants visually distinguish patterns (e.g., a particular block of interactions). This has led to gain more accurate information about the flow of control between method invocations than using the standard sequence diagram. Despite the simplicity of the standard sequence diagram in representing some method invocation scenarios, the responses obtained of participants indicate that such simple representation has led to a misinterpretation of the actual flow of data/control between methods. We have explored such a trade-off using the tasks of Category C4.

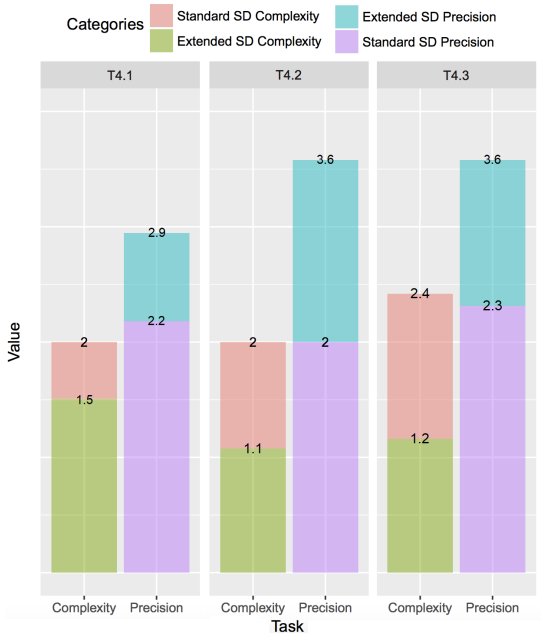


Fig. 7: Mean complexity and precision of the tasks of Category C4

VI. Threats to Validity

In this section, we discuss the validity threats and how we addressed them in our experiment. Such validity threats have been classified into two different categories, namely internal validity and external validity.

A. Internal validity

This type of validity refers to the cause-effect inferences made throughout the analysis. It includes the threats related to subjects, tasks, and other variables.

*1) **Subjects**:* Subjects may be of different levels of expertise. To mitigate this threat, we ensured that to have a fair distribution of the subjects to the two groups, depending on their experience in software engineering in general and in sequence diagrams in particular. To do this, we first asked subjects from the MS and PhD level (informal) questions to infer their comparative experience in the topic of the experiment. For undergraduate students, we have selected students from the same level (i.e., the senior level) and particularly the ones who were working on similar senior projects with under the supervision of one professor. Second, we gave a tutorial to give all participants some background about sequence diagrams and our extensions. Participants also had access to the tutorial as a reference throughout the experiment.

*2) **Tasks**:* The comprehension tasks may have been biased toward our proposed extensions. To mitigate this threat, we have designed our tasks in a way that evens the difficulty of

tasks over the standard and extended diagrams. In other words, there were tasks in which the standard diagram was supposed to be easier to answer than the extended diagram. In addition, the design of the tasks could have been too complex. We have mitigated this threat by performing pilot studies, which in turn enabled us to refine the tasks and exclude the harder or time-consuming tasks. Moreover, the answers of subjects could have been graded wrongly. We mitigated this threat by (a) using a model answer, (b) grading by two persons, and (c) resolving grade disagreements in a discussion with a third person.

*3) **Miscellaneous**:* Our statistical analysis may not have been completely accurate due to having three students with blank-like answers or similar rating points. In order to mitigate this threat, we removed the responses of these three students on all tasks from our analysis. Another threat to validity could be because participants of the two groups were give two different diagrams. We have mitigated this threat by generating the diagrams from the same source code and exporting them as PDF files.

B. External validity

External threats to validity are concerned with the possibility of generalizing the results to different contexts, and the limited representativeness of the tasks, the subjects and the use of $Greenfoot$ as an object. For example, involving more participants (e.g., professionals from the industry or students from various levels) could be a possible threat. Unfortunately, it was quite difficult to invite more than the involved number of participants to the experiment, since participation was completely voluntarily. In addition, inviting people from industry was also a bit challenging, since they are always concerned about how to spend their time efficiently. We plan to extend the number of participants in our experimental evaluation to include software maintainers from diverse backgrounds and different levels of experience.

VII. Related Work

Prior research has investigated techniques to support program comprehension of software behavior using visualization [62], [63]. Such techniques may employ a static, dynamic, or hybrid program analysis and represent program interactions in different forms of visualization.

A. Studies on using standard sequence diagrams

State-of-the-art techniques were proposed to support program comprehension using reverse-engineered sequence diagrams [38], [54], [64]. Techniques were proposed to compact sequence diagrams [10], [14], [15], [18], [19], [32], merge sequence diagrams [65], make interactive sequence diagrams [4], [13], [16], provide better performance [12], [30], or address other software domains [7], [9], [21], [33]. Still, the standard notation of sequence diagrams does not have enough expressive power to precisely represent complex program interactions [38], [66], such as compound method invocations.

B. Studies on using extended sequence diagrams

Prior studies considered extending the existing notation and primitives of the UML sequence diagram in order to model security patterns [42], [43], model thread creation, waiting and notification [44], or show concurrent and distributed interactions [9], [12]. However, only a few studies attempted to simply extend the standard notation of sequence diagrams. For example, Rountev *et al.* [2] extended sequence diagrams to support the presentation of intraprocedural control flow of programs. However, there was no analysis in prior research on how such extensions can improve program understanding. To our knowledge, only two studies [27], [34] evaluated the effectiveness of their proposed sequence diagrams for program comprehension. Nevertheless, such evaluations were not conducted under a controlled environment and only involved a limited number of participants and questions.

C. Studies on using supplementary diagrams

Prior studies proposed to use supplementary diagrams to add more information about program control flow, such as State Chart Diagram [28], [29], [67], Markov Chains [12] as well as Activity and Class Diagrams *et al.* [5]. Using multiple diagrams distracts users and reduces program understandability.

D. Studies on using non-standard forms of visualization

Prior studies proposed techniques to represent program interactions using visualizations forms other than sequence diagrams. For instance, Cornelissen *et al.* [45] proposed a way of representing program interactions using a circular bundle that presents an overall view of software behavior. Fittkau *et al.* [68] proposed to utilize of city metaphor to show the interactions of software entities. However, such kind of techniques provide interaction views that deviate from the standard and hide much details of the program control flow.

VIII. Conclusion

This paper proposes sequence diagram extensions to enhance the visualization of complex method invocation scenarios, such as nested, chained, and recursive method calls. Understanding method invocations using our extended reverse-engineered sequence diagrams allows software maintainers to better track the values or objects of the invoked methods. We have evaluated the effectiveness of our extensions for recognizing the actual behavior of method invocations. To this end, we have conducted a controlled experiment in which participants perform a set of comprehension task. Our results indicate that UML sequence diagrams may give users wrong indications about how the program works. In addition, we observe that our proposed sequence diagram extensions for compound method invocations help participants achieve comprehension tasks with less time and high ratios of correct responses.

We aim in the future to address other limitations of the standard sequence diagrams in which the actual flow of control/data of programs is not well demonstrated, such as static initialization blocks, type casting, exception handling clauses, and method calls that appear as part of the specifications of loops and conditions.

REFERENCES

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

[2] Atanas Rountev, Olga Volgin, and Miriam Reddoch. Static control-flow analysis for reverse engineering of UML sequence diagrams. *ACM SIGSOFT Software Engineering Notes*, 31(1):96–102, 2005.

[3] Atanas Rountev, Scott Kagan, and Jason Sawin. Coverage criteria for testing of object interactions in sequence diagrams. In *International Conference on Fundamental Approaches to Software Engineering*, pages 289–304. Springer, 2005.

[4] Richard Sharp and Atanas Rountev. Interactive exploration of UML sequence diagrams. In *Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005. 3rd IEEE International Workshop on*, pages 1–6. IEEE, 2005.

[5] Elena Korshunova, Marija Petkovic, MGJ van den Brand, and Mohammad Reza Mousavi. CPP2XMI: reverse engineering of UML class, sequence, and activity diagrams from C++ source code. In *13th Working Conference on Reverse Engineering 2006 (WCRE'06)*, pages 297–298. IEEE, 2006.

[6] Liliana Martinez, Claudia Pereira, and Liliana Favre. Recovering sequence diagrams from object-oriented code: An ADM approach. In *Evaluation of Novel Approaches to Software Engineering (ENASE), 2014 International Conference on*, pages 1–8. IEEE, 2014.

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

[8] Paolo Tonella and Alessandra Potrich. Reverse engineering of the interaction diagrams from c++ code. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 159–168. IEEE, 2003.

[9] Lionel C Briand, Yvan Labiche, and Johanne Leduc. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transactions on Software Engineering*, 32(9):642–663, 2006.

[10] Hassen Grati, Houari Sahraoui, and Pierre Poulin. Extracting sequence diagrams from execution traces using interactive visualization. In *17th Working Conference on Reverse Engineering (WCRE), 2010.*, pages 87–96. IEEE, 2010.

[11] Rainer Oechsle and Thomas Schmitt. JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java debug interface (JDI). In *Software Visualization*, pages 176–190. Springer, 2002.

[12] Matthias Rohr, André van Hoorn, Jasminka Matevska, Nils Sommer, Lena Stoever, Simon Giesecke, and Wilhelm Hasselbring. Kieker: Continuous monitoring and on demand visualization of Java software behavior. In *Proceedings of the IASTED International Conference on Software Engineering*. ACTA Press, 2008.

[13] Takashi Ishio, Yui Watanabe, and Katsuro Inoue. AMIDA: A sequence diagram extraction toolkit supporting automatic phase detection. In *Companion of the 30th International Conference on Software Engineering*, pages 969–970. ACM, 2008.

[14] Koji Taniguchi, Takashi Ishio, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Extracting sequence diagram from execution trace of java program. In *Principles of Software Evolution, Eighth International Workshop on*, pages 148–151. IEEE, 2005.

[15] Yui Watanabe, Takashi Ishio, Yoshiro Ito, and Katsuro Inoue. Visualizing an execution trace as a compact sequence diagram using dominance algorithms. *Program Comprehension through Dynamic Analysis*, page 1, 2008.

[16] Kai Koskimies and Hanspeter Mossenbock. Scene: Using scenario diagrams and active text for illustrating object-oriented programs. In *Proceedings of the 18th International Conference on Software Engineering, 1996*, pages 366–375. IEEE, 1996.

[17] Tim Souder, Spiros Mancoridis, and Maher Salah. Form: A framework for creating views of program executions. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 612. IEEE Computer Society, 2001.

[18] Philippe Dugerdil and Julien Repond. Automatic generation of abstract views for legacy software comprehension. In *Proceedings of the 3rd India software engineering conference*, pages 23–32. ACM, 2010.

[19] Juanjuan Jiang, Johannes Koskinen, Anna Ruokonen, and Tarja Systa. Constructing usage scenarios for API redocumentation. In *15th IEEE International Conference on Program Comprehension (ICPC), 2007.*, pages 259–264. IEEE, 2007.

[20] Tewfik Ziadi, Marcos Aurélio Almeida Da Silva, Lom-Messan Hillah, and Mikal Ziane. A fully dynamic approach to the reverse engineering of UML sequence diagrams. In *16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), 2011.*, pages 107–116. IEEE, 2011.

[21] Muhammet Ali Sag and Ayça Tarhan. Measuring COSMIC software size from functional execution traces of Java business applications. In *Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2014 Joint Conference of the International Workshop on*, pages 272–281. IEEE, 2014.

[22] Kunihiro Noda, Takashi Kobayashi, Tatsuya Toda, and Noritoshi Atsumi. Identifying Core Objects for Trace Summarization Using Reference Relations and Access Analysis. In *Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual*, volume 1, pages 13–22. IEEE, 2017.

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

[24] David Lo and Shahar Maoz. Specification mining of symbolic scenario-based models. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 29–35. ACM, 2008.

[25] David Lo, Shahar Maoz, and Siau-Cheng Khoo. Mining modal scenario-based specifications from execution traces of reactive systems. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 465–468. ACM, 2007.

[26] Giovanni Malnati, Caterina Maria Cuva, and Claudia Barberis. JThread-Spy: teaching multithreading programming by analyzing execution traces. In *Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, pages 3–13. ACM, 2007.

[27] Kunihiro Noda, Takashi Kobayashi, and Kiyoshi Agusa. Execution trace abstraction based on meta patterns usage. In *19th Working Conference on Reverse Engineering (WCRE), 2012.*, pages 167–176. IEEE, 2012.

[28] Tarja Systa. On the relationships between static and dynamic models in reverse engineering java software. In *Proceedings. Sixth Working Conference on Reverse Engineering, 1999*, pages 304–313. IEEE, 1999.

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

[30] Yvan Labiche, Bojana Kolbah, and Hossein Mehrfard. Combining Static and Dynamic Analyses to Reverse-Engineer Scenario Diagrams. In *29th IEEE International Conference on Software Maintenance (ICSM), 2013.*, pages 130–139. IEEE, 2013.

[31] Brian A Malloy and James F Power. Exploiting UML dynamic object modeling for the visualization of C++ programs. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 105–114. ACM, 2005.

[32] Del Myers, Margaret-Anne Storey, and Martin Salois. Utilizing debug information to compact loops in large program traces. In *14th European Conference on Software Maintenance and Reengineering (CSMR), 2010.*, pages 41–50. IEEE, 2010.

[33] Andrew R Dalton and Jason O Hallstrom. A toolkit for visualizing the runtime behavior of TinyOS applications. In *The 16th IEEE International Conference on Program Comprehension (ICPC)*, pages 43–52. IEEE, 2008.

[34] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *14th IEEE International Conference on Program Comprehension (ICPC)*, pages 181–190. IEEE, 2006.

[35] Madhusudan Srinivasan, Jeong Yang, and Young Lee. Case studies of optimized sequence diagram for program comprehension. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–4. IEEE, 2016.

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

[37] Tatsuya Toda, Takashi Kobayashi, Noritoshi Atsumi, and Kiyoshi Agusa. Grouping objects for execution trace analysis based on design patterns. In *Software Engineering Conference (APSEC), 2013 20th Asia-Pacific*, volume 2, pages 25–30. IEEE, 2013.

[38] Taher Ahmed Ghaleb, Musab A Alturki, and Khalid Aljasser. Program comprehension through reverse-engineered sequence diagrams: A system-

atic review. *Journal of Software: Evolution and Process*, 30(11):e1965, 2018.

[39] Shaohua Xie, Eileen Kraemer, and RE Kurt Stirewalt. Empirical evaluation of a UML sequence diagram with adornments to support understanding of thread interactions. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pages 123–134. IEEE, 2007.

[40] Vahid Garousi, Lionel C Briand, and Yvan Labiche. Control flow analysis of UML 2.0 sequence diagrams. In *Model Driven Architecture–Foundations and Applications*, pages 160–174. Springer, 2005.

[41] Elizabeth Burd, Dawn Overy, and Ady Wheetman. Evaluating using animation to improve understanding of sequence diagrams. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 107–113. IEEE, 2002.

[42] Alexander van den Berghe, Riccardo Scandariato, Koen Yskout, and Wouter Joosen. Design notations for secure software: a systematic literature review. *Software & Systems Modeling*, pages 1–23, 2015.

[43] Jan Jürjens. Towards development of secure systems using UMLsec. In *Fundamental approaches to software engineering*, pages 187–200. Springer, 2001.

[44] Cyrille Artho, Klaus Havelund, and Shinichi Honiden. Visualization of concurrent program executions. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 2, pages 541–546. IEEE, 2007.

[45] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J Van Wijk, and Arie Van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *15th IEEE International Conference on Program Comprehension (ICPC)*, pages 49–58. IEEE, 2007.

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

[47] Taher Ahmed Ghaleb, Khalid Abdullah Aljasser, and Musab A Alturki. Reverse engineering method, system and computer program thereof, February 2020. US Patent 10,552,286.

[48] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The unified modeling language user guide*. Pearson Education India, 1999.

[49] Bas Cornelissen, Andy Zaidman, and Arie van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 37(3):341–355, 2011.

[50] Michael J Pacione, Marc Roper, and Murray Wood. A novel software visualisation model to support software comprehension. In *Proceedings of the 11th Working Conference on Reverse Engineering, 2004.*, pages 70–79. IEEE, 2004.

[51] Florian Fittkau, Santje Finke, Wilhelm Hasselbring, and Jan Waller. Comparing trace visualizations for program comprehension through controlled experiments. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 266–276. IEEE Press, 2015.

[52] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: a controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 551–560. ACM, 2011.

[58] Taher Ahmed Ghaleb, Khalid Abdullah Aljasser, and Musab A Alturki. Software engineering method including tracing and visualizing, May 2020. US Patent App. 16/778,167.

[53] Massimiliano Di Penta, RE Kurt Stirewalt, and Eileen Kraemer. Designing your next empirical study on program comprehension. In *15th IEEE International Conference on Program Comprehension (ICPC)*, pages 281–285. IEEE, 2007.

[54] Chris Bennett, Del Myers, M-A Storey, Daniel M German, David Ouellet, Martin Salois, and Philippe Charland. A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):291–315, 2008.

[55] Shaohua Xie, Eileen Kraemer, RE Kurt Stirewalt, Laura K Dillon, and Scott D Fleming. Assessing the benefits of synchronization-adorned sequence diagrams: two controlled experiments. In *Proceedings of the 4th ACM symposium on Software visualization*, pages 9–18. ACM, 2008.

[56] Online Appendix. https://taher-ghaleb.github.io/papers/vissoft_2020/appendix.html.

[57] Taher Ahmed Ghaleb. The role of open source software in program analysis for reverse engineering. In *Open Source Software Computing (OSSCOM), 2016 2nd International Conference on*, pages 1–6. IEEE, 2016.

[59] Taher Ahmed Ghaleb, Khalid Abdullah Aljasser, and Musab A Alturki. Method including collecting and querying source code to reverse engineer software, July 2020. US Patent App. 16/778,127.

[60] Frank J Massey Jr. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[61] Howard Levene. Robust tests for equality of variances. *Contributions to probability and statistics: Essays in honor of Harold Hotelling*, 2:278–292, 1960.

[62] Abdelwahab Hamou-Lhadj and Timothy C Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 42–55. IBM Press, 2004.

[63] Michael J Pacione, Marc Roper, and Murray Wood. A comparative evaluation of dynamic visualisation tools. In *20th Working Conference on Reverse Engineering (WCRE), 2013.*, pages 80–89. IEEE Computer Society, 2003.

[64] Matthias Merdes and Dirk Dorsch. Experiences with the development of a reverse engineering tool for UML sequence diagrams: a case study in modern Java development. In *Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 125–134. ACM, 2006.

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

[66] Taher Ahmed Ghaleb. Extending sequence diagrams for better comprehension of program control-flow. Master's thesis, King Fahd University of Petroleum and Minerals, 2015.

[67] Swaminathan Jayaraman, Bharat Jayaraman, et al. Towards program execution summarization: Deriving state diagrams from sequence diagrams. In *Seventh International Conference on Contemporary Computing (IC3), 2014.*, pages 299–305. IEEE, 2014.

[68] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. Live trace visualization for comprehending large software landscapes: The ExplorViz approach. In *First IEEE Working Conference on Software Visualization (VISSOFT), 2013.*, pages 1–4. IEEE, 2013.