Abstract—Reverse-engineered sequence diagrams are promising tools to comprehend the runtime behavior of object-oriented programs. To improve the readability and understandability of massive-scale sequence diagrams, various techniques for effectively exploring or compressing sequence diagrams have been proposed in the literature.

When researchers analyze the effectiveness of these approaches through user studies, it is important to reveal not only what techniques can improve developer productivity but also how developers explore reverse-engineered sequence diagrams and how exploration and compression features are utilized.

We developed a feature to record interactions between a developer and recovered sequence diagrams in our tool, SDExplorer. We show how the recorded interaction data can be used for in-depth analysis of developer activities, toward the evaluation of visualization approaches to helping behavioral comprehension.

Index Terms—interaction data analysis, scalable sequence diagram explorer, visualization, program comprehension

I. INTRODUCTION

Correctly understanding the runtime behavior of a system is an essential part of software maintenance. UML sequence diagrams are helpful to achieve this goal by visualizing the interactions among objects in sequential order. Unfortunately, in real-world software development, due to the fast speed of software evolution, such design diagrams/documents are usually outdated or non-existent [1].

Design recovery techniques give possible solutions to this problem. Runtime information of an object-oriented system can be recorded by utilizing tracing tools (e.g., SELogger [2]); thereby, we can visualize the runtime behavior of a system in a sequence diagram format, called reverse-engineered sequence diagram [3]–[5]. This visualization approach is promising for comprehending the actual behavior of a system; however, owing to the massiveness of execution traces, reverse-engineered sequence diagrams are often afflicted by scalability issues.

Therefore, researchers have made much effort to improve the readability and understandability of massive-scale sequence diagrams. While forward-designed diagrams ignore implementation details and focus on the critical interactions of key objects or classes, reverse-engineered diagrams contain all runtime events and information including trivial elements such as implementation details [6]. Numerous techniques for effectively exploring or compressing sequence diagrams have been proposed in the literature [3], [7]–[15]. These approaches reduce comprehension cost by, for example, interactive exploration, summarizing repetitive behavior, or ignoring less important events.

However, we consider that the effectiveness of such exploration and compression techniques in the actual development tasks is not well evaluated yet; it needs to clarify the advantages and disadvantages of utilizing those techniques. When researchers analyze the effectiveness of exploration and compression features for massive-scale sequence diagrams through user studies, it is important to reveal not only what features can improve developer productivity, or how developers feel on each feature, but also how developers explore diagrams, and how exploring and compressing features are utilized. Moreover, it likewise needs to evaluate the effect of combining multiple exploring and compressing features in terms of program comprehension.

To this end, we develop a feature to record interactions between a developer and reverse-engineered sequence diagrams in our tool, SDExplorer [15]. The SDExplorer supports popular features of existing sequence diagram visualization tools (e.g., searching, filtering, compressing, etc).

We conduct a user study for recording interaction logs about how diagrams are explored and how features are utilized in program comprehension tasks. Then, we show how the interaction data can be used for in-depth analysis of developer activities, toward evaluating visualization approaches to helping behavioral comprehension; by visualizing the recorded logs, we try to capture behavioral patterns of developers in program comprehension tasks, and then show some first results and interpretations thereof.

II. RELATED WORK

Numerous trace visualization approaches to helping behavioral comprehension have been proposed in the past decades. Cornelissen et al. reviewed the history of them [16]. From the report, we know it is 30 years since Kleyn and Gingrich pointed out the value of visualizing runtime behavior [17]. Pauw et al. presented several kinds of visualization of the runtime behavior of Java programs [18], and introduced some preliminary compressing techniques [19]. JAVAIS was then proposed for an educational purpose [20]. SEAT and OASIS were developed in 2004 for software comprehension [5], [21]. SEAT team then focused on techniques for summarizing the
message sequence in traces [8], while OASIS team made a survey on features of sequence diagram visualization tools [3]. JIVE was first proposed in 2007 [22], and Jayaraman et al. added compaction features to it [13]. Extravis uses massive sequence and circular bundle views to visualize the runtime behavior and call relationships of objects [23]. Diver provides a feature to compress repetitive behavior in a recovered sequence diagram by utilizing source code and debug information [24].

In the research field of program behavior visualization, there are four types of evaluation approaches: case study, preliminary, user feedback, and user study [16]. Unfortunately, little existing work conducted user studies; we found two in the above-mentioned related work.

Bennett et al. surveyed features of sequence diagram visualization tools [3]. They conducted a user study to evaluate the effectiveness of tool features. In the study, participants were asked to complete program comprehension tasks with their tool, OASIS. As a result, they concluded the usefulness of tool features, and then made a suggestion on feature improvements and new features.

Cornelissen et al. did a controlled experiment to measure the usefulness of their tool, Extravis [16]. Participants performed software comprehension tasks with and without Extravis, and the time and correctness for each task were measured. They concluded that utilizing Extravis has a positive effect on software comprehension.

As for the evaluation of sequence diagram compression techniques, a common quantitative evaluation measure is compression ratio [7], [9], [10], [13], [24], while a few studies prepared ground truths [7], [9], [10], [13], [24], while a few studies prepared ground truths and evaluated performance [12].

III. SDExplorer IN A NUTSHELL

SDExplorer [15] is a browser-based visualization tool. Fig. 1 shows a snapshot of SDExplorer. It provides rich features for effectively exploring sequence diagrams and achieves high scalability with data and UI virtualization. We showed that SDExplorer could smoothly explore a sequence diagram of over 2,000,000 messages and 3,000 objects; the average rendering time (including the time to load trace data from a database) for such a large-scale sequence diagram was 490 msec, and most operations could be done within 1 sec [15].

In what follows, we briefly describe the architecture and features of SDExplorer. The more details of SDExplorer are described in our previous paper [15].

A. Architecture

SDExplorer takes execution traces and optional group/loop information as its input. Users need to prepare the input data in a JSON format.

SDExplorer consists of two components: controller and renderer. The controller loads the input data from a database (or directly from a JSON file) onto a memory, and the renderer renders a portion of the memory data as a sequence diagram on a display.

To achieve high scalability, SDExplorer applies data and UI virtualization strategies while handling a massive-scale trace data. Only a fragment of input trace data around a window is fetched from a database, and the smaller part is actually (re-)rendered. User operations, such as zooming and moving, trigger an update request, so that the part to visualize is recalculated and the sequence diagram is updated on demand.

SDExplorer supports vertical and horizontal compaction (i.e., grouping messages and lifelines) according to input data. Thus, existing techniques for trace compression can be easily combined with our tool. Moreover, because SDExplorer is provided as a JavaScript library, it can be easily integrated with other tools or embedded into documents [25].

B. Features

Based on the investigation of the characteristics of reverse-engineered sequence diagrams and popular features of existing tools, we implemented the following features in SDExplorer.

- **Zooming/Moving.** Users can zoom-in/out and move a diagram with a mouse.
- **Loop summarization.** SDExplorer supports folding/unfolding messages by summarizing loops (repetitive behavior). Users can specify loop information as the input of SDExplorer. A built-in loop detector can be used if users do not provide any loop information.
- **Lifeline folding/unfolding.** SDExplorer supports hierarchical object (lifeline) grouping. Clicking on a folded group (lifeline) will unfold it, and then hidden objects (lifelines) will appear. Likewise, clicking on an unfolded object group will fold objects into the group.
- **Searching.** While exploring large sequence diagrams, it is almost impossible to find messages and lifelines of interest only with zooming and moving. Users can query a database for finding messages (and caller/callee thereof) of interest, and move to the positions of found messages.
- **Filtering.** To satisfy the demand for focusing only on the interactions between certain objects of interest, a filtering functionality is provided. If users specify several objects of interest, a filtered sequence diagram would be displayed.
TABLE I
DETAILS OF OPERATION LOGS.

<table>
<thead>
<tr>
<th>Operation Type</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>Window location</td>
</tr>
<tr>
<td>Zoom</td>
<td>Window location; Scale</td>
</tr>
<tr>
<td>Compress (Fold loops)</td>
<td>Window location</td>
</tr>
<tr>
<td>Decompress (Unfold loops)</td>
<td>Window location</td>
</tr>
<tr>
<td>Grouping (Fold object groups)</td>
<td>Group id</td>
</tr>
<tr>
<td>Ungrouping (Unfold object groups)</td>
<td>Group id</td>
</tr>
<tr>
<td>Search</td>
<td>Search query; Items selected in a result-set</td>
</tr>
<tr>
<td>Filter</td>
<td>Filter query</td>
</tr>
<tr>
<td>Filter-cancel</td>
<td>-</td>
</tr>
<tr>
<td>Hintbox</td>
<td>Message id</td>
</tr>
<tr>
<td>Nearby</td>
<td>Message id</td>
</tr>
<tr>
<td>InitSize (Reload the page)</td>
<td>Window size</td>
</tr>
</tbody>
</table>

- **Hintbox.** A hintbox will pop up by a double click on a message, which shows the caller, callee, and signature of the message.
- **Showing nearby objects.** By clicking the nearby button, the horizontal order of lifelines are re-arranged to make the view more readable; objects interacting with one another are arranged in close proximity.

C. Logging User Operations

For interaction-based analysis, SDEExplorer can record developer operation logs. Every log entry has an operation type, attributes, and an event timestamp.

The recorded operations and attributes thereof are listed in Table I. For example, a “Move” type log is `{type: Move, attribute: [-179,100], timestamp: 2019-01-02 10:25:30}`.

With operation logs, we can accurately know how users read a sequence diagram to complete comprehension tasks.

IV. INTERACTION DATA ANALYSIS

We conduct a user study to investigate how reverse-engineered sequence diagrams are explored and how tool features are utilized. In the study, we design 4 program comprehension tasks on JHotDraw, which are based on specific types of reverse engineering tasks proposed by Pacione et al. [26]. After that, we generate a sequence diagram from one execution of JHotDraw. Then, we invite 14 graduate students, who each have Java experience of at least more than 2 years, to solve the comprehension tasks. They can only read the sequence diagram, and source code is not available.

In the following, we show how we can analyze subject behavior by using interaction data with some examples.

A. Visualizing Interaction Data

Handling a large number of raw operation logs is troublesome. Inspired by Minelli et al. [27], we take an approach to visualizing the operation logs.

Reviewing collected operation logs, we classify the operation events (see Table I) in SDEExplorer into the following three types.

- **Shifting.** Both moving and zooming belong to this type. These operations frequently appear in logs because a reverse-engineered sequence diagram is usually too large to display in a screen. Shifting operations can only move a small distance; users tend to continually perform shifting operation until messages of interest are found (see the blue ‘blocks’ in Fig. 2). 
- **Navigating.** Searching and filtering are categorized into this type. By these operations, users try to find important information where they start comprehension. Unlike shifting operations, navigating operations need input (i.e., searching or filtering queries). When users perform this type of operations, they tend to have some expectations or hypotheses about the behavior of a subject system.
- **Supporting.** This category contains other types of operations, such as lifeline folding/unfolding and nearby, which are used to facilitate user understanding of sequence diagrams displayed.

Fig. 2 shows an example of visualized operation logs. In visualized logs, “shifting” (resp. “supporting”) operations are marked in blue (resp. red). “Navigating” operations are displayed separately because they play different roles in finding important information.

In addition to those types of operations, non-operation periods are likewise essential. During the study, users could either read sequence diagrams or take notes if they did not perform any operations. In other words, a long-time non-operation period usually means that users are understanding the contents displayed on the screen. We mark the periods of non-operations in yellow.

B. Patterns in Operation Logs

First, we summarize the typical patterns of shifting and navigating operations. Fig. 3 shows four most frequently appearing patterns.

For shifting operations, the pattern ① shows user actions of moving around. This pattern usually appears when users
Critical parts. As a result, even if (a) failed to fully understand the roles of the class important for the tasks s/he undertook, it was possible to list up some important methods. However, (b) paid attention to totally wrong places; thus, s/he resulted in a score of 0 points.

As described above, through visualization of operation logs with critical bars, we can interpret the details of developer activities and the degree of their understanding; we consider that this can be a valuable tool for detailed evaluation of visualization approaches to helping behavioral comprehension.

V. CONCLUSION

When researchers analyze the effectiveness of techniques for effectively exploring or compressing massive-scale sequence diagrams through user studies, it is important to reveal not only what techniques can improve developer productivity, but also how developers explore sequence diagrams and how exploration and compression features are utilized.

In this paper, we introduced a feature of SDExplorer to record developer interactions on reverse-engineered sequence diagrams. Through visualization of operation logs with critical bars, we found some behavioral patterns of developers, which enables us to interpret the degree of their understanding; the interaction data can help in-depth evaluation of sequence diagram-based visualization approaches to helping behavioral comprehension.

We plan further and more detailed evaluation of sequence diagram exploration and compression features based on recorded interaction data, subject feedbacks, and scores; we try to reveal what types of features are effective for what types of comprehension tasks in what types of contexts.

Acknowledgment: This work was partly supported by MEXT KAKENHI Grant Numbers JP15H02683, JP18H03221 and JP18J15087.
REFERENCES


