SDExplorer: a generic toolkit for smoothly exploring massive-scale sequence diagram

Kaixie Lyu
Tokyo Institute of Technology, Japan
lvkaixie@sa.cs.titech.ac.jp

Kunihiro NODA
Tokyo Institute of Technology, Japan
knhr@sa.cs.titech.ac.jp

Takashi KOBAYASHI
Tokyo Institute of Technology, Japan
tkobaya@cs.titech.ac.jp

ABSTRACT

To understand program's behavior, using reverse-engineered sequence diagram is a valuable technique. In practice, researchers usually record execution traces and generate a sequence diagram according to them. However, the diagram can be too large to read while treating real-world software due to the massiveness of execution traces.

Several studies on minimizing/compressing sequence diagrams have been proposed; however, the resulting diagram may be either still large or losing important information. Besides, existing tools are highly customized for a certain research purpose. To address these problems, we present a generic toolkit *SDExplorer* in this paper, which is a flexible and lightweight tool to effectively explore a massive-scale sequence diagram in a highly scalable manner. Additionally, *SDExplorer* supports popular features of existing tools (i.e. search, filter, grouping, etc.). We believe it is an easy-to-use and promising tool in future research to evaluate and compare the minimizing/compressing techniques in real maintenance tasks.

SDExplorer is available at https://lyukx.github.io/SDExplorer/.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;

KEYWORDS

scalable sequence diagram explorer, reverse engineering, visualization, program comprehension

ACM Reference Format:
Kaixie Lyu, Kunihiro NODA, and Takashi KOBAYASHI. 2018. SDExplorer: a generic toolkit for smoothly exploring massive-scale sequence diagram. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Conference on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3196321.3196366

1 INTRODUCTION

Correctly understanding the runtime behavior of a system is an important part of software maintenance. Sequence diagrams as design documents are helpful to achieve this goal as they visualize the interactions of objects in a sequential way. However, in real-world software development, due to the fast speed of software evolution, such design documents are usually out-of-date or even do not exist.

Reverse-engineered sequence diagrams, which are generated by visualizing execution traces in a sequence-diagram style, can then be useful to address this problem [3, 14]. Furthermore, as reverse-engineered sequence diagrams are generated from real execution traces, they can capture runtime bindings and thread interactions, which makes it a potent technique for debugging [1].

The main challenge of this technique is to deal with the "size explosion" problem, for the reason that execution traces will increase rapidly at runtime. Due to the massiveness of execution traces, reverse-engineered sequence diagrams would be too huge to read. Lots of previous studies aim to solve this problem by compressing the sequence diagram. Typically, sequence diagrams can be compressed in 2 directions: horizontally and vertically. Horizontally, grouping objects will cut the number of lifelines. A novel approach is identifying the most important objects (i.e. core objects) and grouping objects with a focus on them [11]. The grouping operation usually decreases the vertical size as well because interactions between objects in the same group will be ignored after grouping. Vertically, summarizing loops decrease the number of visible messages. The loop information is provided in a pre-process [9], or detected in traces [6].

Although compressing technique can be effective in cutting the size of a reverse-engineered diagram, important information might be eliminated (e.g., detailed iterations of a loop are necessary in debugging situations; important messages may be hidden by grouping their caller/callee objects). Hence, only visualizing and exploring the compressed diagram by using normal UML sequence diagram tools are not sufficient for totally comprehending program's behavior.

Almost all the previous studies developed special tools to address the "size explosion" problem. However, these existing tools have following issues:

- The tools will crash when loading huge sequence diagrams. During Bennett *et al.*'s survey, they destroyed one instance because of the huge size [1]. However, real-world software can produce ultra-size traces in a very short time.
- Each tool is highly customized for a certain purpose in each research team, which becomes an obstacle to evaluate and compare the effectiveness of existing techniques. E.g., *JIVE* [6] is a comprehensive debugging support tool with an awesome sequence diagram module; however, since it is a Java debugger plugin of the IDE *Eclipse*, non-Java programs are not supported, and even non-Eclipse-based Java projects have to pay extra efforts to use this tool. Besides, to

ICPC '18, May 27–28, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5714-2/18/05.
https://doi.org/10.1145/3196321.3196366

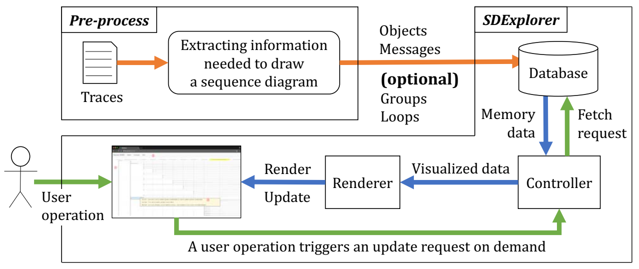


Figure 1: Architecture of *SDExplorer*

evaluate a new proposed approach, it is necessary to implement another customized tool from scratch, which brings a high cost to researchers.

To address these issues, in this paper, we present a tool, *SDExplorer*, such that:

- **Highly scalable:** We use *virtualization* technique to display the diagram: only a small fragment of a diagram is actually rendered in a display window (e.g., a display window includes only around 100 messages and objects while the entire diagram contains millions of events). The display window will be moved by user operations on demand so that it is able to smoothly explore massive-scale sequence diagrams containing millions of events.
- **Publicly available for generic use:** Our *SDExplorer* is a browser-based tool, which makes it an easy-to-use and platform-independent tool. *SDExplorer* is also programming-language-independent because the tool takes its input (i.e., execution trace information) from generic databases. Besides, *SDExplorer* supports major features of existing tools (i.e. zooming, searching, filtering, folding/unfolding, etc.). *SDExplorer* (including its source code) is publicly available at https://lyukx.github.io/SDExplorer/.

Moreover, we now continue on enhancing *SDExplorer* to additionally support the feature of recording user operations in the near future (details are explained in section 5). We believe *SDExplorer* will become a promising aid to evaluate the effectiveness of trace summarization and effective exploration techniques in program comprehension tasks with reverse-engineered sequence diagrams.

2 SDEXPLORER IN A NUTSHELL

2.1 Architecture

Real-world software usually generates massive-scale traces in a rather short time. *SDExplorer* supports virtualization with a database, which achieves very high scalability.

Figure 1 shows the architecture of *SDExplorer*.

The pre-process part in Figure 1 generates the input of *SDExplorer*. Users of *SDExplorer* need to prepare execution traces and optionally groups/loops information as an input of *SDExplorer*. *SDExplorer* receives the input in a standard *json* format to store the data into the database. Techniques of previous studies on trace summarization (e.g., [10–13]) can be then easily combined with our tool.

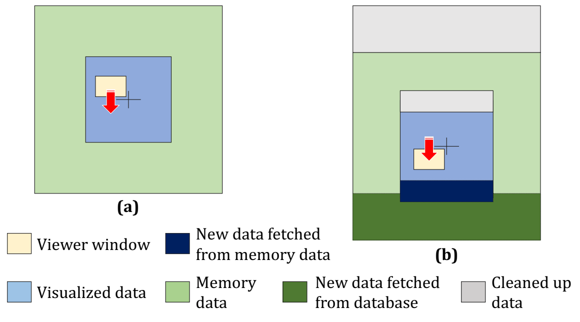

(a)
(b)
Viewer window
New data fetched from memory data
Visualized data
Memory data
New data fetched from database
Cleaned up data

Figure 2: View-update strategy of *SDExplorer*

The input data is stored in a database. *Controller* loads the input data onto a memory (the loaded-data is called *memory data*), then *Renderer* renders a part of the *memory data* as a sequence diagram on a display.

To explore the sequence diagram smoothly, we implemented a virtualization mechanism. Figure 2 shows the view-update strategy of *SDExplorer*. The *viewer window* is an area where users can see a (part of) sequence diagram on a display, and moves according to users operations like scrolling or executing loop compaction. Sequences contained in the area of the *visualized data* is pre-rendered by *Renderer*. Sequences contained in the area of the *memory data* is pre-loaded onto a memory by *Controller*. Once *viewer window* passes through the center line of *visualized data*, as shown in Figure 2 (b), *SDExplorer* will clean up 25% data in *visualized data* and fetch new data of the same size from *memory data*. According to this data loading, the location of *visualized data* will move downwards. Likewise, 25% of the *memory data* will be cleaned up, and new data will be loaded. The location of *memory data* will also be updated.

2.2 Features

In this section, we elaborate the features of *SDExplorer* that are listed below. **Interactive folding/unfolding** and **loop summarization** are key features to support trace compaction techniques that are necessary for dealing with massive-scale reverse-engineered sequence diagram.

- **Zooming**. Users can zoom in/out a diagram by using a mouse wheel.
- **Interactive folding/unfolding**. *SDExplorer* supports (hierarchical) object grouping. Clicking on a folded group will unfold it and the hidden messages will appear. Likewise, clicking on an unfolded group will fold objects in the group and eliminate messages among them. (see Figure 3 ④⇔⑤.) The motivation of providing this feature is that though grouping is a powerful technique to reduce the size of a sequence diagram, some important information might be hidden by it. For instance, in Figure 3 (a) and (b), since `B` and `C` belong to one group `BC`, grouping `B` and `C` will eliminate the message `important()`.
- **Loop summarization**. Clicking the `Compress` button on the toolbar (Figure 5 ②), repetitive messages are summarized with UML combined fragments (Figure 3 ⑥). Although users are allowed to provide loop information as an optional

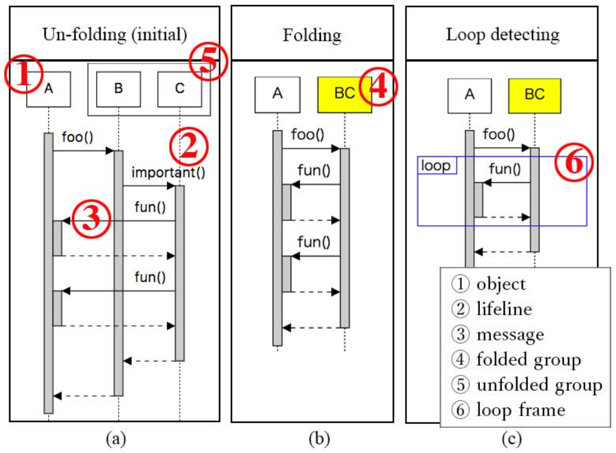


Figure 3: *SDExplorer*'s features to support trace compaction techniques

input of *SDExplorer*, a built-in loop detector, which uses a simplified version of Jayaraman et al.'s algorithm [6], is implemented in *SDExplorer* for improving the users' convenience.

- **Searching and filtering**. `Search` is used to locate one message. Figure 4 shows how `Search` works. For the better user experience, *SDExplorer* supports interactive search with a fuzzy query (Figure 4 (b)). Figure 4 (c) shows a search result. Clicking one of the search result items, the display window will be moved to the clicked message's location. `Filter` takes a set of objects/groups as filter queries, and then draw a sub-diagram consisting of the messages among objects/groups specified as the queries.
- **Hintbox** While reading a large-scale sequence diagram, it is useful to zoom out and read the outline of entire interactions. In this scenario, texts of messages might be too small to read, and the caller/callee objects might be out of the screen. It could be annoying to move or zoom-in several times just for seeing messages' texts. To avoid the bothersome situation, we implemented *Hintbox* feature: double-clicking on a message, a tooltip, which is describing the details of caller/callee objects and method signature, will pop out. Note that this hintbox always has a fixed size (i.e., the size never change by zooming).

3 DEMONSTRATION

To evaluate the usability and scalability of *SDExplorer*, we made a demonstration on `jpacman`, a Java implementation of a traditional Pac-Man game.

As the pre-process of *SDExplorer*, we collected an execution trace and grouping information as follows. We repeated the approach of Noda et al. [11]: recording traces by using `SELogger` [7] and obtaining grouping information by identifying core objects. After the pre-process, we got 3,109 objects, 29 object-groups, and 3,454,948 messages. Then we eliminated self-calls as a common way for generating an efficient sequence diagram [1]. Finally, 2,554,604 messages remained.

Table 1: Response time of features [sec]

	render[1]	zoom	loop[2]	fold	unfold	search	filter
max	0.52	0.96	1.11	0.11	0.48	6.08	22.68
min	0.43	0.80	0.03	0.02	0.02	1.95	20.06
average	0.49	0.87	0.70	0.06	0.17	3.32	21.20

[1] the time elapsing from "loading trace data from the database" to "finish rendering a sequence diagram on the display"
[2] includes both of the built-in loop detection and the loop compaction

3.1 Scalability

The size of *Memory data* was set as 5,000 messages and 5,000 objects, and the size of *visualized data* was set as 360 messages and 252 objects (the sizes are configurable if necessary). The runtime environment was macOS 10.13.3, 2.3GHz Intel i5 CPU, 8GB of RAM, MongoDB v3.4.9, and Chrome version 63.0.3239.132. As an evaluation of the scalability, we measured the response time of *SDExplorer*'s features on a reverse-engineered sequence diagram of `jpacman`. The result is shown in Table 1.

For the **load & render**, we recorded the response time in 5 different locations in the diagram and calculated the max, min, and average value. Similarly, we repeated **zooms** 5 times from the largest scale to the smallest scale and measured the response time. As for the **loop**, the response time depends highly on the displayed messages (e.g., summarizing a loop including 15,000 messages means loading from database 3 times). We randomly chose 5 locations in the diagram, then applied the loop compression to them and measured the response time. The response time of **fold** and **unfold** also depends highly on the number of objects in a group and the number of interactions within the group. Choosing 5 groups (including the largest one of 497 objects and smallest one of 1 object), we measured the response time of **fold** and **unfold**. As for **search**, we picked five query words randomly and executed the search for each query word to find messages containing the query. For **filter**, we repeated the following procedure 5 times and measured the response time: we randomly chose 10 objects as a filter query and executed the filtering (i.e., only 10 objects were left in the diagram after filtering).

From the result shown in Table 1, most of the features can be executed within around 1 second; this shows the high scalability of *SDExplorer* when treating the massive-scale traces including millions of events. The response time of **search** and **filter** is relatively large compared with other features. The bottleneck is the response time of a database query, which we can not make improvement unless we develop a special database for *SDExplorer*. However, we think users do not execute the **search** and **filter** features so frequently; thus, the response time is expected to be acceptable during practical maintenance tasks.

3.2 Trace summarization

SDExplorer provides the features to support visualization of the results of major trace summarization approaches: loop compaction

Table 2: Trace summarization performance

Raw #msg.	LC[1] #msg.	OG[2] #msg.	LC & OG #msg.
2,559k	1,334k (47.88%)	2,555k (0.17%)	65k (97.45%)

[1] LC means loop-compaction was applied
[2] OG means object-grouping was applied
Each number in the parentheses means space savings:
(1 - (compressed size) / (uncompressed size)) * 100 [%]

and object grouping. In this section, we demonstrate the features for trace summarization by applying trace summarization to whole *jpacman* data. For loop summarization, we used a built-in loop detection feature. As for grouping objects, we utilized grouping information obtained by applying the technique of Noda et al. [11].

The summarization result is as shown in Table 2. Table 2 shows the number of messages in the diagram before/after the trace summarization. From the result, we can see that a large number of messages disappears by the trace summarization features. Although the compression ratio is very high, the resulting diagram is still large; thus, the features for smooth exploration provided by *SDExplorer* are necessary. We believe *SDExplorer* will be a promising aid to evaluate and compare the usefulness of summarization techniques and summarized diagrams in future research.

4 RELATED WORK

There are many approaches to compress a sequence diagram, which can be used as the pre-process part in *SDExplorer*.

To reduce the vertical size, summarizing loops is a popular approach [6, 9, 12, 13]. Jayaraman et al. and Taniguchi et al. developed original loop detecting algorithms to find continuous repeats [6, 13]. Bohnet et al. proposed novel metrics to measure the similarity between function-calls for loop summarization [2]. However, performing loop detection on trace data involves some false positives. A guaranteed way to identify loop structures is marking the entry and exit events of every loop while recording traces. Unfortunately, this approach usually imposes large overhead to record traces. Myers et al. proposed an optimized way that used debug information to support locating loops [9]. Another approach is detecting loops from event patterns in a trace.

Apart from summarizing loops, Hamou-Lhadj et al. presented a trace summarizing approach based on the removal of implementation details [5]. Bohnet et al. applied a pruning algorithm to simplify execution traces [2]. These techniques are also helpful to reduce the vertical size, and they can be combined with loop summarization techniques to get a better result.

As for the horizontal reduction of a sequence diagram, a major approach is object grouping. For a Java project, a simple way is using package information to group objects [1, 6]. Another way is to group strongly co-related objects by identifying core objects[11] or design patterns [10].

In the aspect of visualization, most existing researches use the standard UML sequence diagram [3, 6, 12]. Bennett et al. and Jayaraman et al. expanded the standard with several features to support their research [1, 6] (e.g., Bennett et al. introduced `groups` based on package structure of Java projects). Besides, Cornelissen et al. used circular bundle views to visualize execution traces [4]. Bohnet et al. applied different techniques to visualize execution traces including call graphs, call sequences, call matrices, etc. [2]

5 CONCLUSION AND FUTURE DIRECTION

This paper presents *SDExplorer*, a browser-based and light-weight tool for smoothly exploring massive-scale sequence diagrams. *SDExplorer* takes trace and summarization data as its input and visualizes it in a sequence diagram format. Our tool achieves a high scalability and solves the freeze/crash problem of existing tools while loading a large sequence diagram. The tool also provides useful features like folding/unfolding, loop compaction, filtering, etc. We believe it is an easy-to-use and promising tool in future research to evaluate and compare the minimizing/compressing techniques in real maintenance tasks.

As a part of future work, *SDExplorer* will support recording user operations in the near future. Minelli et al. analyzed user interactions of IDE and quantified program comprehension while reading source code [8], learning from which we plan to explore the effectiveness of existing trace summarization techniques in the aspect of program comprehension by using operation logs of *SDExplorer*.

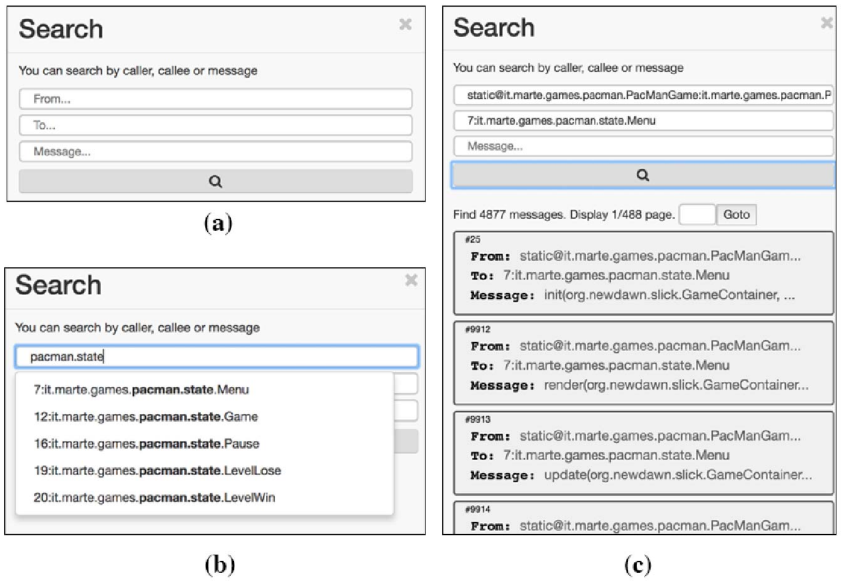
ACKNOWLEDGMENTS

This work was partly supported by MEXT KAKENHI Grant Numbers JP26280021 and JP15H02683.

REFERENCES

[1] Chris Bennett, Del Myers, M-A Storey, Daniel M German, David Ouellet, Martin Salois, and Philippe Charland. 2008. A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. *Journal of Software: Evolution and Process* 20, 4 (2008), 291–315.
[2] Johannes Bohnet, Martin Koeleman, and Jürgen Döllner. 2009. Visualizing massively pruned execution traces to facilitate trace exploration. In *Proc. VISSOFT.* 57–64.
[3] Lionel C Briand, Yvan Labiche, and Johanne Leduc. 2006. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transactions on Software Engineering* 32, 9 (2006), 642–663.
[4] Bas Cornelissen, Andy Zaidman, and Arie van Deursen. 2011. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering* 37, 3 (2011), 341–355.
[5] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. 2006. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proc. ICPC.* 181–190.
[6] S Jayaraman, Bharat Jayaraman, and Demian Lessa. 2017. Compact visualization of Java program execution. *Software: Practice and Experience* 47, 2 (2017), 163–191.
[7] Toshinori Matsumura, Takashi Ishio, Yu Kashima, and Katsuro Inoue. 2014. Repeatedly-executed-method viewer for efficient visualization of execution paths and states in java. In *Proc. ICPC.* 253–257.
[8] Roberto Minelli, Andrea Mocci, Michele Lanza, and Takashi Kobayashi. 2014. Quantifying program comprehension with interaction data. In *Proc. QSIC.* 276–285.
[9] Del Myers, Margaret-Anne Storey, and Martin Salois. 2010. Utilizing debug information to compact loops in large program traces. In *Proc. CSMR.* 41–50.
[10] Kunihiro Noda, Takashi Kobayashi, and Kiyoshi Agusa. 2012. Execution Trace Abstraction Based on Meta Patterns Usage. In *Proc. WCRE.* 167–176.
[11] Kunihiro Noda, Takashi Kobayashi, Tatsuya Toda, and Noritoshi Atsumi. 2017. Identifying Core Objects for Trace Summarization Using Reference Relations and Access Analysis. In *Proc. COMPSAC.* 13–22.
[12] Madhusudan Srinivasan, Jeong Yang, and Young Lee. 2016. Case studies of optimized sequence diagram for program comprehension. In *Proc. ICPC.* 1–4.
[13] Koji Taniguchi, Takashi Ishio, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2005. Extracting Sequence Diagram from Execution Trace of Java Program. In *Proc. IWPSE.* 148–154.
[14] Tewfik Ziadi, Marcos Aurélio Almeida da Silva, Lom Messan Hillah, and Mikal Ziane. 2011. A fully dynamic approach to the reverse engineering of UML sequence diagrams. In *Proc. ICECCS.* 107–116.

A APPENDIX



Search

You can search by caller, callee or message

From...

To...

Message...

(a)

Search

You can search by caller, callee or message

pacman.state

7:it.marte.games.**pacman.state**.Menu

12:it.marte.games.**pacman.state**.Game

16:it.marte.games.**pacman.state**.Pause

19:it.marte.games.**pacman.state**.LevelLose

20:it.marte.games.**pacman.state**.LevelWin

(b)

Search

You can search by caller, callee or message

static@it.marte.games.pacman.PacManGame:it.marte.games.pacman.P

7:it.marte.games.pacman.state.Menu

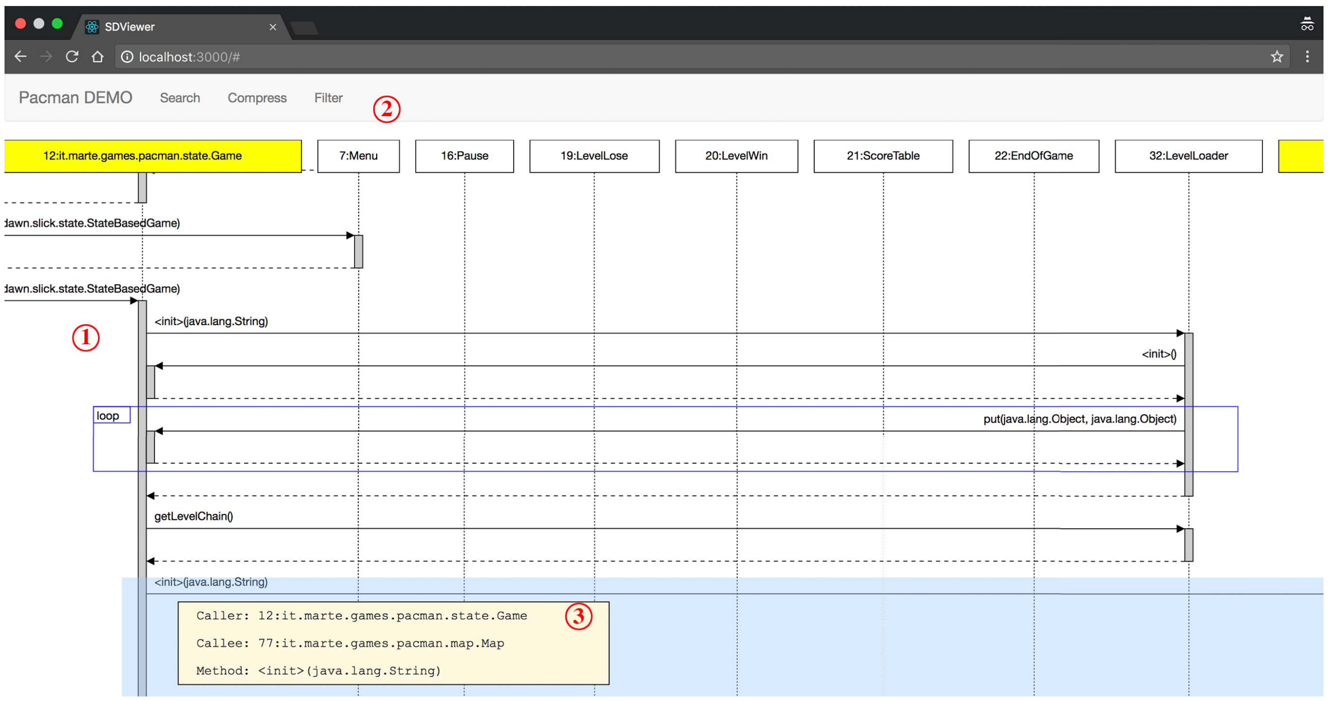
Message...

Find 4877 messages. Display 1/488 page. Goto

```
#25
From: static@it.marte.games.pacman.PacManGam...
To: 7:it.marte.games.pacman.state.Menu
Message: init(org.newdawn.slick.GameContainer, ...

#9912
From: static@it.marte.games.pacman.PacManGam...
To: 7:it.marte.games.pacman.state.Menu
Message: render(org.newdawn.slick.GameContainer...

#9913
From: static@it.marte.games.pacman.PacManGam...
To: 7:it.marte.games.pacman.state.Menu
Message: update(org.newdawn.slick.GameContainer...

#9914
From: static@it.marte.games.pacman.PacManGam...
```

(c)

Figure 4: An example of searching messages



```
Caller: 12:it.marte.games.pacman.state.Game
Callee: 77:it.marte.games.pacman.map.Map
Method: <init>(java.lang.String)
```

Figure 5: A snapshot of *SDExplorer*