JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2008; **20**:291–315
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.372

Research

A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams

C. Bennett[1], D. Myers[1], M.-A. Storey[1,*,†], D. M. German[1], D. Ouellet[2], M. Salois[2] and P. Charland[2]

[1]*Department of Computer Science, University of Victoria, Victoria, BC, Canada*
[2]*Defence R&D Canada–Valcartier, Québec, QC, Canada*

SUMMARY

Sequence diagrams can be valuable aids to software understanding. However, they can be extremely large and hard to understand in spite of using modern tool support. Consequently, providing the right set of tool features is important if the tools are to help rather than hinder the user. This paper surveys research and commercial sequence diagram tools to determine the features they provide to support program understanding. Although there has been significant effort in developing these tools, many of them have not been evaluated using human subjects. To begin to address this gap, a preliminary study was performed with a specially designed sequence diagram tool that implements the features found during the survey. On the basis of an analysis of the study results, we discuss the features that were found to be useful and relate these to the tasks performed. It concludes by proposing how future tools can be improved to better support the exploration of large sequence diagrams. Copyright © 2008 Crown in the right of Canada. Published by John Wiley & Sons, Ltd.

Received 10 January 2008; Revised 30 April 2008; Accepted 7 June 2008

KEY WORDS: sequence diagrams; reverse engineering; software understanding; tool survey; user study

1. INTRODUCTION

Sequence diagrams can be valuable aids to understanding software system behaviour because they highlight the most important requirements of a system [1]. While originally devised as a notation to capture scenarios during analysis and design, sequence diagrams can also aid understanding of

*Correspondence to: M.-A. Storey, Department of Computer Science, University of Victoria, Engineering Office Wing 348, 3800 Finnerty Road (Ring Road), Victoria, BC, Canada V8P 5C2.
†E-mail: mstorey@uvic.ca

Contract/grant sponsor: Copyright © Her Majesty the Queen in Right of Canada as Represented by the Minister of National Defence (2008); contract/grant number: W7701-5-2677/001/QCL

existing software through the visualization of execution call traces. The power of sequence diagrams lies in their ability to represent selected behaviour at a suitable level of abstraction.

Reverse-engineered sequence diagrams can be created through static or dynamic analysis; the advantages of the latter being increased precision, control over inputs and conditional behaviour, as well as resolution of polymorphism and runtime binding in object-oriented languages [2]. Reverse-engineered sequence diagrams based on dynamic call traces are typically very large. The sequence diagram 'size explosion' [2] problem has been approached primarily in two ways: through pre-processing to reduce the size of the initial sequence, and through tool support for user interaction. Pre-processing techniques include reduction at the source through data collection techniques and sampling [2]; collapsing similar sequences using pattern matching (to identify loops, recursion, and non-contiguous repetitions); automatic detection of utility functions (using fan-in/fan-out metrics) [3]; removing abstract operation calls [3]; hiding constructors and getters/setters [4]; limiting the depth of the call tree [3,4]; eliminating self-calls [5]; and selecting relevant areas and hiding details [6,7].

Reverse engineering is a mentally challenging task. Effective cognitive support can improve the performance of reverse engineers by offloading some or most of the cognitive processing onto an external tool (e.g. by reducing the need to remember volumes of detailed information or to perform repetitive calculations) [8]. Such cognitive support can be provided by the features available in a sequence diagram viewer and applied to a range of reverse engineering tasks including design and architecture recovery, feature location, design pattern discovery, and re-documentation at various levels of abstraction. As mentioned above, previous research has addressed techniques for automatically reducing the size of sequence diagrams. However, less work has been done in the area of visualization and user interaction, and most of this work has received little formal evaluation. Indeed, Sharp and Rountev [9] suggest '…what are the actual benefits of the visualization techniques? Experiments with programmers or university students should be used to quantify the benefits of individual tool features'. Our work addresses the important gap in the research by addressing the following research questions:

1. What kinds of interaction and presentation features do state-of-the-art tools provide for exploring sequence diagrams?
2. How can sequence diagrams play a role in understanding the behaviour of a software system during reverse engineering?
3. How can sequence diagram tools be improved?

The following section describes the approach taken to investigate these questions.

2. RESEARCH CONTEXT AND METHODOLOGY

The context for our research lies within a project called Opening up Architectures of Software-Intensive Systems (OASIS) at Defence R&D Canada (DRDC)–Valcartier. An earlier OASIS study [10] concluded that graphical representations generated by software analysis tools could speed up software understanding, providing information that is difficult to obtain using only an

DOI: 10.1002/smr

integrated development environment (IDE) such as Eclipse [11]. However, such visualizations sometimes fail to provide information at an appropriate level of abstraction and can easily overload users with irrelevant low-level details (especially with dynamic analysis data). In the case of trace-generated sequence diagrams, these issues stem primarily from the potentially massive amount of information that must be visualized. Effective visualization tools for dynamic analysis are clearly desirable for future OASIS users, but it is not clear which features are needed to alleviate cognitive issues. The long-term goal of our study is to determine the required features and to design tool support for integration within the OASIS tool set.

To answer the above three research questions, a detailed survey of the literature and commercial tools that support sequence diagram generation and exploration was first conducted. From this, a summary of interaction and presentation tool features was synthesized (Section 3). The summary is a useful first step, as most of these tool features lack user-based evaluations of how they provide cognitive support during program understanding tasks. We also recognize that this feature set may be incomplete or inaccurate and that more research is needed to understand user needs. We performed a user study and held a focus group to address such issues and answer the second and third research questions. We used a mixed-methods methodology that was both explanatory—to validate the features—and exploratory—to extend the set to meet actual user requirements. There are three basic stages to this research:

1. We designed and implemented a sequence viewer to support this research. This was necessary because no existing tool incorporated all the surveyed features. Moreover, of the tools available, some were proprietary and others were not designed to support general reverse engineering tasks (e.g. the *TPTP* tool [12] focuses on performance profiling and captures incomplete traces). Having complete control over tool features implementation also allowed us to integrate with an IDE. This was valuable when observing sequence diagram use within the context of extensive program understanding tasks. Finally, we were able to optimize the performance for larger traces in an attempt to ensure that this was not a barrier to task completion. The design and implementation of this tool are described in Section 4.
2. We performed a focus group with professionals from DRDC Valcartier because they have experience in software reverse engineering and are target users of the OASIS project. This helped in understanding their current methods and expose frustrations they encounter during their work. This information was valuable in the design of the tasks used in the observational experiment. The focus group is detailed in Section 5.
3. We designed and performed an experiment to observe and measure how users exercise the sequence diagram tool features. The experiment involved asking participants to complete a number of software reverse engineering tasks using our sequence tool. During the experiment, participants were observed at work in order to quantify how they exercised each feature. This allowed the identification of when participants had to use other methods to complete the tasks. The effectiveness of each participant on these tasks was also evaluated. These results were then used to validate the initial feature set and discover new features that should be useful. Findings from the experiment were augmented by interviews and a follow-up questionnaire to further explore which features participants found useful, and to identify features they felt were missing. The experiment and its findings are described in Section 6.

Related work is discussed in Section 7. We then conclude this paper with ideas for future work.

DOI: 10.1002/smr

3. SURVEY OF SEQUENCE DIAGRAM TOOL FEATURES

This section provides a survey of sequence diagram tools to discover the features they provide. The majority of tools surveyed implement some form of sequence diagram visualization. Others implement views that offer the same functionality using a different visualization, and hence are valuable for discovering feature requirements. For example, SEAT [13] presents program call traces in the form of a tree view. Sequence diagram user interface features can be divided into two categories: (1) presentation or display facilities, and (2) features that allow the user to interact with and explore the diagram. Note that there may be some overlap between these two feature sets: presentation often being both the result of interaction, and also a necessary precursor to it (e.g. highlighting and hiding could be considered interaction as well as presentation features).

3.1. Presentation features

Presentation features affect the layout of the diagram as well as facilities for showing multiple views, hiding information, and making the most effective use of animation and visual attributes. We found seven such features:

Layout: An important presentation feature is laying out a sequence diagram according to some notational standard. Many tools use their own layout format or some variation on a standard format (e.g. The Unified Modelling Language (UML 2.1)), sometimes adding proprietary extensions to address a specific problem (e.g. how to capture conditional branches). *Scene* [14] produces sequence diagrams according to Rumbaugh's object modelling technique (OMT) notation [15]. *SCED* [16] uses its own UML-like notation that provides constructs for nested sub-scenarios and repetition. *TPTP* [12] also uses UML.

Multiple linked views: It is often necessary to provide multiple views [1] as well as an overview. Views can be of the same type (e.g. to allow comparison of different parts of a trace) or different types (e.g. linked class diagram and sequence diagram views). *Ovation* [17] renders sub-trees using a number of alternative 'charts', including a static class list or a class communication graph. *SCED* supports sequence diagrams and state charts that show transitions within a selected object. Linking these views so that they remain synchronized and can be easily navigated is another useful feature. *SEAT* [13] provides links between sequence and source code views. Similarly, *Scene* links between sequence views and static class diagrams or source code views. An overview is provided by many tools. *ISVis* [18] provides a two-window scenario view consisting of an information mural overview and a temporal message-flow diagram. *Scene* displays a summary call matrix view alongside a sequence view.

Highlighting: Highlighting a section of a sequence diagram is often the expected visible outcome of a user selection or search. Tools that support manual selection of components may use highlighting to indicate selection (e.g. [18]). Highlighting can go beyond single components to show related objects or messages.

Hiding: Hiding information is commonly used to control complexity in sequence diagram tools. Hiding provides abstraction by removing detailed sub-message calls from below a parent call. Components can be hidden following pre-processing, a search (filtering), or a manual selection. *ISVis* supports hiding of classifiers within a subsystem, *SEAT* supports manual hiding, and *VET* [19] hides

DOI: 10.1002/smr

elements following filtering. Similarly, when grouping occurs (described in more detail below), the grouped elements are hidden 'under' a summary element [2]. When components are hidden as a result of filtering, it is important to indicate this so that the user can redisplay these components if required. One technique is to 'grey out' the components, rather than completely hiding them. There should also be an indication of why a set of components was hidden (e.g. as a result of loop detection or pruning of utility functions) [3]. Cornelissen *et al.* [4] propose hiding null return values or abbreviating return values and parameter lists.

Visual attributes: Colour and shape are useful ways to code additional information about a sequence. *Ovation* uses colour to differentiate objects and bevelling to indicate that components are grouped under the bevelled component. *TPTP* uses colour to indicate the length of time spent inside a method execution.

Labels: Classifiers, messages, and return values are usually labelled. Occlusion and legibility are challenges when displaying larger sequences. Techniques to cope with this include hiding labels, replacing them with rectangles when zoomed out (e.g. as implemented by the *VET* tool), or using mouse hovers (e.g. as in *Ovation*).

Animation: Many tools support animation. This comes in two varieties: one that supports stepping through a sequence diagram message by message and another that uses animation to morph between diagram states to help the user maintain context. *Scene* supports single-step animation between trace calls. *AVID* [20] supports animation between component groupings.

3.2. Interaction features

Interaction features allow the user to communicate with the tool to navigate, query, and manipulate the sequence diagram. We found eight such features:

Selection: Manual selection of elements is a prerequisite for further interaction such as manipulation, filtering, and slicing. This is supported by most tools with user interaction.

Component navigation: Rapid, simple movement between components (traversing the call tree) is important to usability [3], as is the ability to move between instances of the same type of pattern (e.g. sub-scenarios) in tools that support grouping of similar patterns (e.g. *SEAT*).

Focusing: Focusing on a specific part of a diagram or behaviour has been identified as a problem when dealing with large traces [2]. Koskimies and Mössenböck [14] note that it can be solved by techniques such as collapsing calls, partitioning sequences into manageable chunks, and selecting an object such that only related messages are shown. Single-step animation can also be used to focus on individual messages.

Zooming and scrolling: Zooming and scrolling [2] are standard techniques to cope with information that is difficult to display all at one time. Semantic zooming reveals more details as the user zooms in, whereas physical zooming simply enlarges the contents. *VET*, *Ovation*, *TPTP*, and *Jinsight* [21] support zooming and scrolling. *Jinsight* supports semantic zooming as well.

Querying and slicing: Queries identify and optionally filter information within a sequence. Scenariographer [22] supports both relational Structured Query Language and set-based Software Modelling Query Language queries on underlying structured data. *ISVis* allows exact, inexact, and wild-card searches. *VET* provides graphical support for selection of objects based on class and name as well as selection of methods by type or time range. Although these are more limited than language-based queries, they provide a much simpler solution. Slicing can be performed on either

DOI: 10.1002/smr

objects or methods and is a specific form of query that selects only entities related to the selected component (a slice through the sequence flow).

Grouping: Grouping can be the result of slicing or it can be done manually (e.g. *AVID*'s manual clustering and *Ovation*'s flattening and underlaying). This is usually indicated by some sort of icon or visual attribute of the summary component (behind which grouped components are hidden). Grouping of objects will result in collapsing the sequence horizontally, but may leave all messages visible (no vertical compaction). However, Cornelissen *et al.* [4] describe a technique to collapse lifelines that would eliminate calls between the merged objects. Grouping at the message level will hide messages called by the grouped messages (vertical compaction). Grouped items can also be annotated with a label (and optionally comments) describing the grouped abstraction. Riva and Rodriguez [23] refer to these approaches as vertical and horizontal abstractions. In addition to pre-processing to detect repeating patterns, interaction support can allow manual selection and collapsing of repeated patterns such as loops. *TPTP* supports grouping of lifelines using pre-determined levels of abstraction (host, process, thread, class, and object), grouping of method calls, and arbitrary user-defined groupings.

Annotating: Annotating can be used for many purposes: to describe why components were grouped [24], to capture user understanding during exploration of a sequence diagram, and to provide waypoints [25] and messages to oneself and others when the diagram is to be shared. Few tools provide annotation mechanisms, *ISVis* [18] being an exception with its facility to describe user-identified sub-scenarios.

Saving views: Saving views, either to share or to revisit, is also important when documenting a user's understanding of the diagram. A tool should be able to save the entire state of the visualization so that it can be restored at a later time. Together with annotations, a saved view can tell a story

Table I. Comparison of features implemented by selected sequence visualization tools.

	ISVIS	Jinsight	Program explorer	SCED/ Shimba	Scene	Together control center	TPTP	VET
Presentation features								
Layout	•	•	•	•	•	•	•	•
Multiple linked views	•	•	•	•	•	•	•	•
Highlighting		•	•		•		•	
Hiding	•	•		•	•		•	•
Visual attributes	•	•	•	•	•	•	•	•
Labels	•	•	•	•	•	•	•	•
Animation					•			•
Interaction features								
Selection	•	•	•		•		•	
Component navigation	•		•		•		•	
Focusing		•	•		•			
Zooming and scrolling	•	•	•	•		•	•	•
Queries and slicing	•	•	•	•	•	•	•	•
Grouping	•	•		•	•			
Annotating	•							
Saving views	•	•	•			•		

DOI: 10.1002/smr

about the diagram being visualized. Hamou-Lhadj and Lethbridge [3] discuss the need to save both the original trace and the transformations that were applied to reduce the complexity of the trace, although saving a record of user interactions is not discussed.

3.3. Feature summary

Table I summarizes these features and maps them to a selection of sequence visualization tools. Note that this feature matrix was derived from literature reviews and limited tool trials. The features listed can be implemented in a variety of ways and we consider even a partial implementation to warrant a mapping from feature to tool (indicated with a bullet in the table). In addition to research tools, several commercial and open-source tools have capabilities to reverse engineer sequence diagrams. For Java, these include IBM's Rational *Rose Enterprise* [26], Omondo's *EclipseUML* [27], and Borland's *Together Control Center* [28]. The latter is included in the table as it is representative of the level of interaction support in commercial tools. The following section outlines how these features are implemented in the OASIS sequence explorer (OSE), a tool developed at the University of Victoria for research into sequence diagram visualization and exploration.

4. THE OASIS SEQUENCE EXPLORER

The OSE was developed based on the features first described in [29] and detailed in Section 2. This tool is implemented in Java as a set of Eclipse views. Choosing optimal (or at least effective) ways to implement tool features is a difficult problem. For the OSE tool, we considered how features had been implemented in previous tools, and, where needed, designed new solutions. OSE is composed of four main views, illustrated as *A–D* in Figure 1. View *A* is the sequence diagram editor, which is divided into three sub-views or panes. The main pane (*A-1*) displays the sequence diagram (or chart) and allows the user to navigate through it. The clone pane (*A-2*) displays the same information but allows the user to scroll to a different horizontal position from the main pane. This makes visible, at the same time, the source and target of method activations when they are separated by large horizontal distances. At the top, a third pane (*A-3*) contains the package hierarchy for the objects/classes in the sequence diagram. All panes are resizable and can be hidden.

A sequence diagram is essentially a tree structure that is visualized using the UML 2 notation. In this case, it is enhanced with elements to ease navigation. Diagrams are laid out as a series of lifelines with object or class names in boxes at the top of each lifeline. When a sequence diagram is too large to view at once, the user may scroll, zoom, fit to window, or specify a legible text layout (the default view). There is also a linked outline view (Figure 1(B)), which displays the complete sequence, with the visible area shown by a draggable blue rectangle that can be used to navigate. An activation may have zero or more child activations. The presence of child's activations is indicated in the chart as an activation box with cyan-green borders. Hovering over an activation box displays a small plus sign. Clicking it expands the activation, revealing child's activations and re-laying out the chart if necessary. Conversely, a minus sign is displayed to collapse the activation. It is also possible to expand or collapse all activations at once, or to selectively expand/collapse

DOI: 10.1002/smr

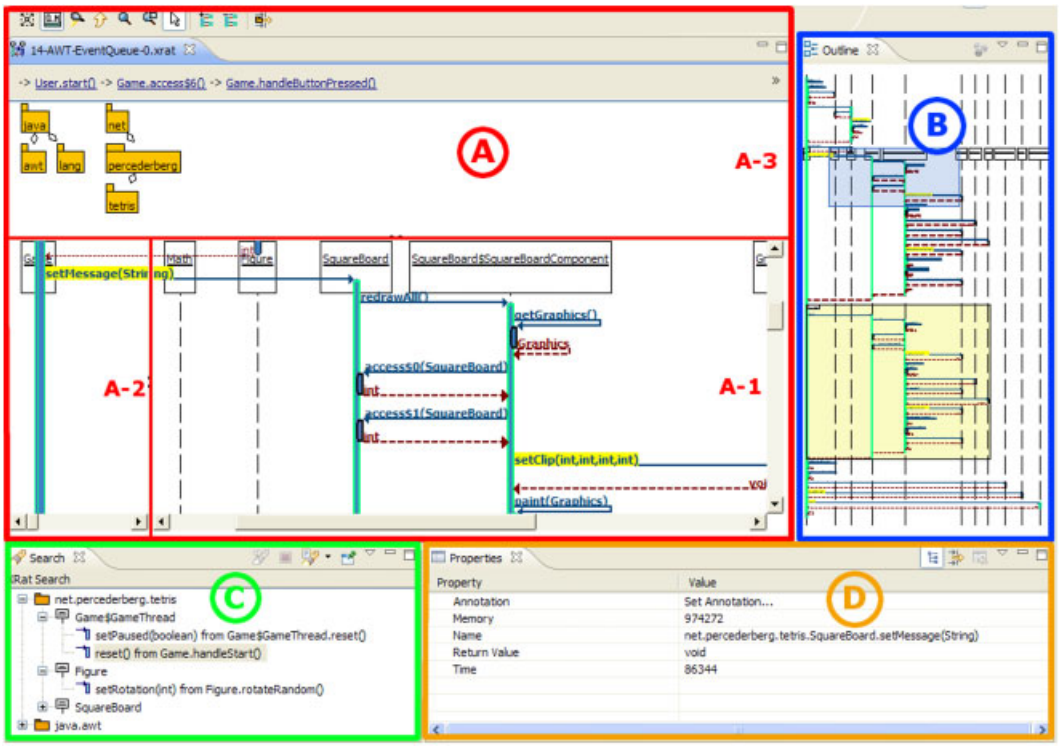

Figure 1. The views of the OASIS sequence explorer.

a single parent activation, a whole lifeline, or a package. Lifelines are not shown unless there is a visible path from the root chart activation to one or more activations on that lifeline. Activations and lifelines are thus grouped and hidden according to their sub-tree in the call hierarchy.

It is possible to 're-root' the chart by focusing on an activation and its children, hiding everything else. A breadcrumb trail from the original root to the currently selected root is displayed at the top of the editor (top of Figure 1(A)). Any level in the call hierarchy can be selected in the breadcrumb trail to reset the chart root. Alternatively, focusing on the parent will navigate one level up the hierarchy.

The sequence diagram editor supports grouping into two other ways. First, lifelines can be grouped together using the package pane. Selecting a lifeline pulls up the package pane that displays containment arcs in the static class hierarchy. A minus sign next to a package group lifelines into a single lifeline under its containing package (Figures 2(a) and (b)).

Grouping is also done through loop recognition. In general, loop recognition is a difficult problem. We do not attempt to find complete or partial matches in sub-trees of the call hierarchy. Instead, we consider activations with the same method signature, which originate from the same activation to be equivalent. Each repetition of the same pattern of activations is considered to be a different iteration of the same loop. Loops are visualized inside a box labelled with the number of iterations and the iteration which is currently displayed. By default, only one iteration is displayed. The user

DOI: 10.1002/smr

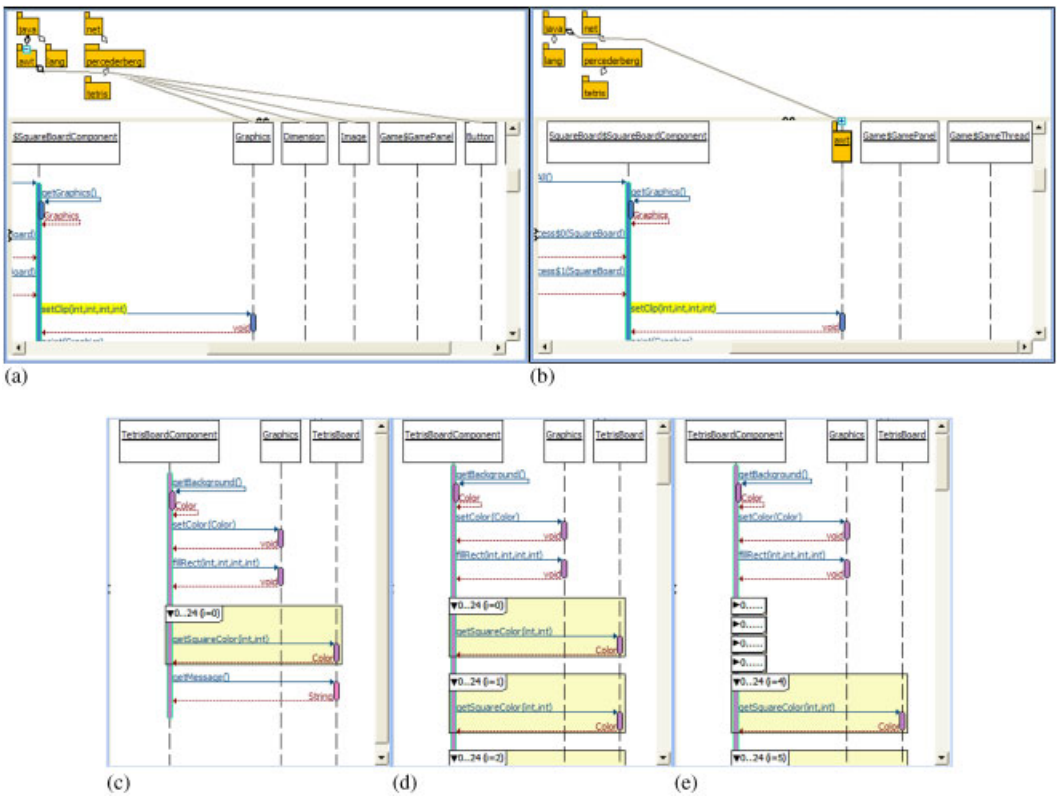

Figure 2. Grouping horizontally and vertically: (a) the ungrouped `java.awt` package in the package pane; (b) the grouped `java.awt` package as a single lifeline; (c) one iteration of the loop is displayed ($i=0$); (d) all iterations are displayed sequentially; and (e) several iterations are collapsed.

can select another iteration by right-clicking on the box label. Alternatively, all loops can be shown or loops can be collapsed into a vertical group. Figures 2(c)–(e) illustrate this feature.

Except for trivial sequence diagrams, it is unlikely that the entire diagram will be both visible and comprehensible on a typical monitor. Therefore, the OSE tool allows the user to search for elements using substring matching or regular expressions. The Eclipse Search view (Figure 1(C)) is populated with search results organized by package, class, activation, and return values. Results are also highlighted in the sequence diagram editor and double clicking on a result will select and focus on the corresponding element in this editor. Some elements of the sequence diagram may have properties that are not visible in the sequence diagram editor (e.g. activations have a time associated with them). When the user selects any diagram element, these properties are displayed in the Eclipse Properties view (Figure 1(D)). It is also possible to annotate any element in the diagram using the properties view. Annotations are saved and can then be searched and edited. A number of other interactions and operations are implemented in OSE. Package filters can be used to remove elements from the display and the search. Hovering over or selecting an activation highlights its entire sub-tree. All visual interactions are animated to maintain context. Double clicking on an

DOI: 10.1002/smr

Table II. Feature implementation in the OASIS sequence explorer.

Feature	Implementation
Layout	Sequence diagram layout; fit-to-screen layout
Multiple linked views	Linking between editor, outline, search, package pane and main/clone panes, properties view, and source editors
Highlighting	Highlighting according to selection and search matches
Hiding	Filtering based on packages; grouping/collapsing elements implicitly hides other elements
Visual attributes	Use of colour to indicate activations that can be expanded/collapsed; '+' and '−' decorations for elements with children; figure shapes modeled after familiar UML notation
Labels	Every element has a label drawn according to familiar UML notation
Animation	All interactions with the diagram are animated to maintain context
Selection	Every element is selectable
Component navigation	Elements can be navigated to using the search view. Also a breadcrumb trail can be used to navigate up the call hierarchy
Focusing	Focus on an activation to make it the root of the diagram
Zooming and scrolling	Standard zoom in/out operations; zoom to marquee; standard scrolling using scrollbars; linked outline view scrolls the sequence diagram editor
Queries and slicing	Ability to search on any element in the sequence
Grouping	Grouping of activations, packages, and repeated patterns
Annotating	Annotating in properties view and within the sequence diagram editor
Saving views	Sequence diagram editor state saved on close

element opens the source code editor for that element if the source code is available. Finally, the state of the sequence diagram editor is saved so that the diagram returns to a familiar state when reopened. Table II maps these functions, operations, and interactions to the features described in Section 3.

5. FOCUS GROUP

The focus group session was carried out with three professional software developers from DRDC Valcartier, all potential users of the OSE tool. The goal of the focus group was to explore reverse engineering tasks that software developers carry out in their normal work, with an emphasis on how sequence diagrams are, or how they could be used to assist in these tasks. This information helped to answer the second research question: 'How can sequence diagrams play a role in understanding the behaviour of a software system during reverse engineering?' It also helped to choose appropriate tasks for the user study.

5.1. Focus group design

In the first part of the focus group, participants were asked to describe the tasks they performed as part of their job, including design, coding, maintenance, code review, bug fixing, and testing. They were also asked to describe the tools and techniques they used in their work (e.g. use of

DOI: 10.1002/smr

debugging and visualizations). More focused questions were asked to determine how participants carried out feature and defect location, as well as other program understanding tasks relevant to software maintenance. In the second part of the focus group, we demonstrated the OSE tool and asked participants to comment on the tool and to describe how they might use it in their work. To conclude, the participants were asked for suggestions on how other technologies could assist in their reverse engineering tasks.

5.2. Focus group findings

Each of the three professionals in the focus group ($F1$, $F2$, and $F3$) had different work roles and varying perspectives on the reverse engineering process. $F1$'s main focus was on software development, debugging, and testing. $F2$'s work involved integration and investigation of open source components. $F3$ primarily performed code and technical reviews. All three participants used Eclipse and Java in their work and were members of teams with four to six people.

$F1$ and $F2$ were concerned with being able to find and understand how features were implemented in software. To do this, $F1$ mostly read the code, documenting his understanding with pen and paper. He did not use a search facility and rarely used a debugger, finding it too difficult to set breakpoints in interesting places. $F2$, on the other hand, used third-party software to create class diagrams. He then used a debugger to understand the functionality of the software. Finally, he drew small sequence diagrams by hand to document this understanding. $F3$ was more concerned with understanding the impact of code changes. For this, he depended on source code repositories, synchronization tools, and a debugger. He first located the code changes and then used the debugger to analyse their impact. $F2$ observed that few of his colleagues used sequence diagrams in their daily work.

The focus group participants also provided feedback on the OSE tool. $F2$ said that he liked the split panes and felt that it could replace some of the other tools he used. He did note it would be preferable to be able to specify the start of the sequence trace, rather than tracing an entire program run. $F3$ noted that the tool could also be used to optimize a program if it displayed profiling information such as method execution duration and memory used. He also suggested that being able to visualize multiple threads and their interactions would be useful. All participants agreed that the ability to annotate diagrams would be useful, but not a critical feature because they typically documented for personal use, not for sharing with others.

5.3. Focus group discussion

The feedback from focus group participants was used to guide the design of our study. Although we recognize that the number of participants in the focus group was low, we learned that not all reverse engineers use sequence diagrams for program understanding. This prompted us to add a basic training session on sequence diagrams to our user study. The participants also indicated that they tended to use sequence diagrams to support their own understanding and did not share their notes or diagrams with other team members. Consequently, for our study we focused on individual tasks rather than collaborative tasks. The focus group also prompted us to include tasks dealing with understanding how specific features were implemented (i.e. tasks $T5$–$T8$ discussed in the following section).

DOI: 10.1002/smr

6. USER STUDY

The user study was the main component in the research to understand the effectiveness of tool features. It consisted of a lab experiment, an interview, and a follow-up questionnaire.

6.1. Study set-up

This subsection describes the preparation and the set-up of the lab experiment, the interview and follow-up questions asked, and the data analysis approach.

6.1.1. Lab experiment

For the lab experiment, six participants with experience in software development and program understanding were recruited using convenience sampling: five computer science graduate students at the University of Victoria and one participant from industry. Each participant was given the same tasks. The tasks were related to one application: a multiplayer distributed Tetris game written in Java and consisting of 10 500 source code statements and 57 classes in eight packages. This application was selected because it was used in a previous study [30], it was developed by a colleague of the researchers (providing a privileged understanding of its architecture and functionality), and was appropriately complex for the study tasks.

The experiment began with a comprehensive 30-min hands-on tutorial on how to use the sequence tool to create a program trace and then navigate, search, and manipulate the resulting sequence diagram. The participant was then asked to create a program trace by running and interacting with the Tetris application. Next, the participant was given 90 min to complete nine assigned study tasks. There were several criteria used to design the tasks. First, we took into consideration the common work tasks identified during the focus group. Next, we considered previously published studies and selected tasks at the level of the large-scale and small-scale questions described in [31,32]. Finally, we presented the tasks in such a way as to guide the participant from a high-level understanding of both the tool and the Tetris application to a more detailed understanding of both. The tasks assigned were as follows:

*T*1. There are three threads used by the program at runtime (numbered 1, 13, and 15). For each thread, describe your understanding of its responsibility.
*T*2. For thread 13, identify the packages that it depends on at runtime and document the number of classes that are called at runtime.
*T*3. Choose one class (from a list of given classes) and document the classes it depends on in the context of thread 1.
*T*4. For the class chosen in *T*3, describe its runtime responsibilities in the context of thread 1.
*T*5. Describe the user-initiated shutdown process of the program.
*T*6. List the methods that new mouse handler(s) would have to call in order to enhance the current keyboard controls with mouse buttons and the scroll wheel.
*T*7. The program currently chooses blocks randomly. What code would need to be modified or replaced in order to change this process so that it is not random?

DOI: 10.1002/smr

$T8$. The program is written using Java Swing and the Abstract Windowing Toolkit (AWT). Describe the coupling of the program to Swing/AWT, and briefly describe what would need to be done to replace Swing with another graphical user interface framework.
$T9$. Identify if and where the command pattern is used in the software.

Task $T1$ required that the participants map overall application behaviour to each thread. Tasks $T2$ and $T3$ involved gaining an understanding of static structure. Task $T4$ focused on understanding the behaviour of a single class and tasks $T5$–$T7$ were chosen as typical program maintenance tasks. Tasks $T8$ and $T9$ were optional, due to their level of difficulty and the time constraints of the study. Participants were asked to use the OSE tool to solve the assigned tasks but were encouraged to browse the source code if they wished. Participants were also allowed to ask questions if they had difficulty using the tool, and were allowed to use any other features provided by the Eclipse's Java development tools. The participants' written solutions were verified by one of the authors, based on information provided by the developer of the Tetris game. The success of each answer was rated from 0 to 2 where 0 was unsuccessful, 1 partially successful, and 2 completely successful.

The sessions were videotaped and the screens were captured for later coding and analysis. The same Windows XP workstation with a dual core AMD Athlon 2.2 GHz CPU, 2 GB RAM, and a 1600 by 1200 resolution monitor was used for each participant. Eclipse Europa 3.3 and the OSE tool were installed on this workstation.

6.1.2. Interview and questionnaire

Immediately following the lab experiment, participants were independently interviewed to help understand their experience while carrying out the tasks. A total of 16 questions were asked to understand: (a) overall experience, (b) frustrations and barriers, (c) usefulness of tool features, (d) usefulness of tool in understanding structure and behaviour of the subject system, (e) task-specific strategies, and (f) their opinions on information overload and tool performance.

To examine features whose use could not be effectively evaluated during the experiment, a questionnaire was sent to all participants following the experiment. The questionnaire asked participants to rate the usefulness of seven features on a scale of 1–5 where 1, useless; 2, not very useful; 3, somewhat useful; 4, useful; and 5, very useful. The questionnaire also asked each participant to provide a qualitative assessment of each feature. Features surveyed and their associated questions are listed in Table III.

6.1.3. Data analysis approach

Using screen captures from the experiment, we coded 35 separate user operations. These operations included such things as making selections inside the OSE tool, collapsing, activations, resizing windows, and using tools supplied by Eclipse, but not a part of the OSE tool. Operations were then grouped by the features that they support. The multiple linked views feature, for example, is supported by nine different user operations including interacting with the different panes supplied by the OSE tool, and using the search view to make a selection. Using the observations alone, some features could not be mapped to operations; hence, we inquired about these using the survey.

DOI: 10.1002/smr

Table III. Questionnaire.

Feature	Implementation
Layout	The sequence diagram was laid out according to invocation time for activations, size of labels, etc. How useful was the layout to you?
Multiple linked views	There were several views available to you: the sequence diagram; the overview; the search view; the properties view; the package hierarchy; the clone pane; and the source code. Did you find the linking useful?
Highlighting	During your interaction with the sequence diagram, highlighting was used in several ways: elements were made bold when they were selected or moused over; matches to searches were highlighted in yellow. Was this feature useful to you?
Hiding	It was possible to filter some calls from the sequence viewer. How useful did you find this?
Visual attributes	The visual attributes of the various views were used to attempt to indicate how you could interact with them. For example, activation boxes that had sub-activations (children) were outlined in green. When you moused over elements that could be expanded/collapsed, a $+/-$ symbol was displayed. Packages had different colours and shapes that classes. Were these visual attributes useful to you?
Labels	How useful was it to you that you could clearly see the textual labels for the elements in the viewer?
Animation	While you interacted with the sequence viewer, the diagram animated to react to your interactions. Was this useful to you?

We tracked when users performed operations outside the tool because these operations indicate something of an 'anti-feature.' That is, they indicate instances when the provided tool was not sufficient for the user to complete the task.

6.2. User study findings

This subsection describes the results from the lab experiment, interview, and follow-up questionnaire. These findings are further discussed in Section 6.3 and summarized in Table IV.

6.2.1. Lab experiment

To begin the experiment, users captured their own dynamic sequence traces of the Tetris program. The class files for the program were statically instrumented beforehand in order to make the tracing process as non-intrusive as possible. There was one instance of a trace that was too large for the tool to handle. It was unfortunately destroyed during the experiment; hence, its size could not be recorded. The size metrics of the traces are maximum call count (23 227), minimum call count (3049), mean call count (11 663.075), maximum call depth (12), minimum call depth (5), and mean call depth (10.55).

During the experiment, participant interactions were captured to determine how many times a tool feature was used to accomplish each of the assigned tasks. Although the primary purpose of this experiment was to understand the process followed and features used by participants, we also

DOI: 10.1002/smr

Table IV. Findings and discussion summary.

Feature	Usefulness	Improvements
Layout	Critical—it was determined that the UML layout is workable	None identified
Multiple linked views	Critical—the sequence diagram does not provide a complete picture on its own	Link to static structure view (e.g. class diagram). In the package pane, do not collapse sub-packages at the same time. View multiple trace threads simultaneously
Highlighting	Useful	None identified
Hiding	Critical—needed to manage complexity of large traces	Allow filtering of all classes in a package; Users need a way to know what has been hidden in a diagram
Visual attributes	Useful	Make better use of colour to understand what activations are, or can be, collapsed and to indicate relationships between elements in the diagram
Labels	Critical—without names, methods and classes make no sense	Improve placement of labels. Method execution labels should be nearer the target
Animation	Useful	None identified
Selection	Critical—a prerequisite for related features such as grouping	None identified
Component navigation	Useful—in particular for specific understanding tasks	Permit jumping between child and parent activations or activations in a single lifeline. Note this could be thought of as a sort of 'semantic scrolling'
Focusing	Little used—possibly because highlighting achieves a similar purpose	Focus on a single class, showing related messages to and from the class (the most requested feature)
Zooming and scrolling	Critical—scrolling Little used—zooming was seldom used due to automatic sizing	None identified
Queries and slicing	Useful—in particular for specific understanding tasks	Search across threads. Show a portion of the execution from a specified start point to a specified end point (note that this could perhaps best be implemented during trace capture rather than during user interaction)
Grouping	Useful	None identified
Annotating	Not evaluated by tasks	Allow tagging. Show indication of annotation in sequence diagram view
Saving views	Useful	Extend this idea with a new feature: saving of session state
New features		
Saving of session state	Not evaluated	Save the state of the session to help the user recover or go back to a previous setting
Integrated static analysis	Not evaluated	Augment dynamic information with static analysis

DOI: 10.1002/smr

evaluated participant success on the assigned tasks. Two participants completed all the tasks and two completed the mandatory tasks and part of the first optional task. The remaining two participants did not complete the mandatory tasks. All participants were able to complete $T1$ successfully. Disregarding those who were unable to complete a task, participants performed better for tasks $T4$–$T7$ (functionality-related tasks) than they did for tasks $T2$ and the first part of $T3$ (structure-related tasks). The participant with the best success for task $T8$ worked outside the OSE tool more than inside it for this task.

Of the 15 features described in Section 3, nine features were measured by counting explicit user operations. Highlighting, layout, visual attributes, labels, saving views, and animation were implicitly involved in many or all operations and are discussed in Section 6.2.3. Annotating was not required by any of the tasks. The total count of features used by all tracks, from most to least used, is zooming/scrolling (700), multiple linked views (473), grouping (465), hiding/expanding (458), selection (231); working outside the tool (193), queries/slicing (66), component navigation (59), and focusing (25).

Both these numbers (and those of Figure 3) were determined from operation counts and are independent of participant success on a given task. These numbers show that the most frequently used feature was zooming/scrolling. Operation counts showed that scrolling represented the vast majority of these operations, with only 10 zooming operations executed. The second most used feature was multiple linked views. 'Working outside the tool' is related to this feature because the view that users most commonly linked to was the source code editor, which was not provided by the OSE tool. Grouping and hiding/expanding operations, typically used to reduce the amount of information presented, showed significant use. The selection feature was the next most used and was typically used when considering a specific activation, class, or package and its related entities. Queries, component navigation, and focusing received significantly less use.

Figure 3 shows the relative use of features per task, revealing that participants working on the thread responsibility task ($T1$) made the most use of several features including grouping and hiding/expanding features. This might also be explained by the fact that, since this was the first task, participants spent more time playing with the tool to familiarize themselves. Focusing was used fairly evenly across tasks, and the grouping and hiding/expanding features have nearly identical use (in the OSE implementation, one is almost always caused by the other).

6.2.2. Interview

After the experiment, participants were interviewed individually to gain a better understanding of their experiences with the tool. Five out of six participants said that the OSE tool helped them understand the functionality of the software. One of these five noted that it was also necessary to view the source code. The sixth participant said that the tool 'came close' in helping to understand the functionality of the software. This participant also liked the tool's presentation of the program as a 'visual stack trace'. Three participants found the tool effective for understanding the structure of the software and one of these three suggested that there were other visualization tools better suited for understanding structure.

When asked to list the most useful tool features, five participants mentioned the package pane and the ability to expand and collapse packages into a single lifeline. One of these, however, said that this feature could be improved if it did not collapse sub-packages at the same time. Among the

DOI: 10.1002/smr

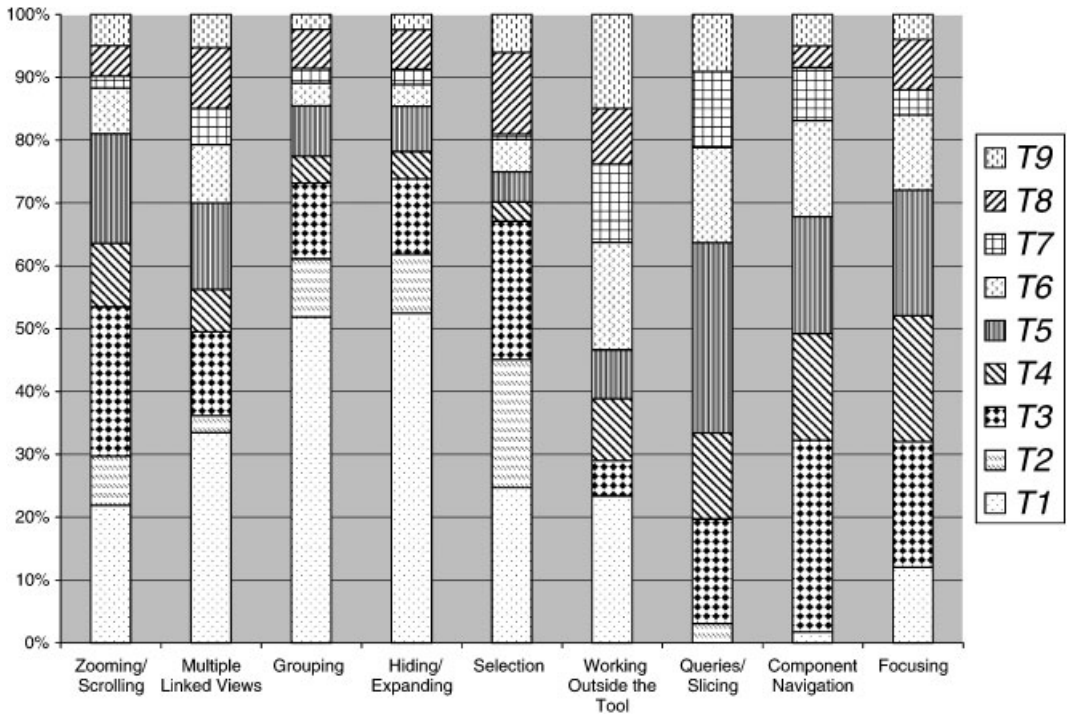

Figure 3. Percentage feature use by task.

least useful features, three participants said that they did not use zooming, preferring the default zoom level.

6.2.3. Follow-up questionnaire

For some tool features, it was difficult to determine how often they were used or how useful they were from observations alone. This is because some features are either pervasive, occurring as a result of most other actions (e.g. animation), or are implicit, occurring without conscious user intervention (e.g. saving of views). To address this, we provided a questionnaire that asked participants to evaluate and discuss the effectiveness of these features. Only one feature, filtering, was rated poorly (2 out of 5 mean score). All other features scored 4 out of 5. This might be because people did not feel comfortable criticizing the tool with a potential tool designer (a typical bias in this kind of study).

All participants found the search feature and selection highlighting feature useful, but requested more of a visual difference between expanded and collapsed sub-sequences. There were several comments in the survey related to filtering. Although the default package filtering was found to be effective, some participants would have liked more advanced and intuitive filtering at different levels. A common request related to filtering was to support focusing on a class, by filtering unrelated details. Two participants complained about tool performance but others found it good or adequate.

DOI: 10.1002/smr

Although labels were found to be very helpful by all participants, more than one person suggested that activation labels should be placed closer to the target activation. One participant also found that method activations were, at times, too small to easily click on. Finally, most participants agreed that animation between view changes helped maintain context.

6.3. User study discussion

In this subsection, some conclusions are drawn to answer the third research question about how sequence diagrams tools can be further improved, recognizing the limitations of the study (which are outlined below). In general, tasks with lower success rates (e.g. $T2$ and $T3$) were those that required structural understanding (for which the sequence diagram is clearly not an ideal tool). Although structure can be inferred from dynamically created sequence diagrams, they are unlikely to provide sufficient details and will be more difficult to use than views that explicitly show the structure. This was also confirmed by the interview where only half the participants thought that the tool helped with structural understanding. Two participants did not complete the mandatory tasks. This appeared to be due, in one case, to capturing an abnormally large execution trace, and in the other because the participant chose a more complex class for tasks $T3$ and $T4$.

In our study, task success does not appear to be correlated to the use of specific features. Perhaps this is because of the small number of participants and the diversity of their approaches to reverse engineering tasks. Ironically, the heavy use of a feature (e.g. repeated scrolling) may not indicate that a tool feature is helpful in solving a problem, but may actually be a sign that the user is unable to find what he or she is looking for. It could also indicate that it is such a basic feature that we should not record its use. The count numbers and those of Figure 3 were not normalized for task duration. The order of tasks and consequent learning effects may have influenced task performance and duration for the later tasks. Task $T1$ (thread understanding) made the most use of several features but this may be in part due to it being a larger and more difficult task than many of the others. $T5$ (understanding the shutdown process) was also a difficult task and required comparatively more operations. Interestingly, $T3$ (class dependencies) also required a lot of effort, possibly because it was a structural analysis task that was not well suited to the tool.

Tool performance was not found to be a limiting factor in the tasks performed. Even for substantial traces consisting of more than 20 000 calls, the tool remained responsive and usable. Instead, cognitive overload, as shown by participant frustration and repeated comments, appeared to be the major barrier to task completion.

6.3.1. Usefulness of features

Scrolling was measured as the number one activity, probably due to the size of the traces. Although this feature was obviously necessary, it would be good to reduce the need to do this through improved searching, filtering, and focusing support. The feature list combines zooming and scrolling into a single feature as both mechanisms enable users to view large diagrams in a limited space. The study participants, however, much preferred to scroll through the logical area rather than zoom. The layout of the diagram is what allows the diagram to be displayed in a meaningful way; hence, it is a critically important feature. Participants never attempted to change the layout of the diagram, although one did comment that he/she liked the 'visual stack trace' nature of the layout. This makes

DOI: 10.1002/smr

it difficult to judge any advantages of using UML as a basis for layout, but it does indicate that it is adequate enough to fulfill this critical feature.

It is clear that multiple linked views were vital in accomplishing the study tasks. The linked package pane view was selected by five of the six participants as being the most useful feature. The combination of navigation to source code and working outside the tool (typically browsing source code) were the second most common activities after scrolling. Hence, integration within an IDE is clearly important, as has been noted by other researchers [33].

Tasks $T5$–$T9$ required participants to investigate specific parts of the application. Figure 3 indicates that for these very specific and focused tasks, querying, component navigation, and use of multiple linked views (primarily source code viewing) are the most important. Although focusing received little use, it could be argued that selection, which highlights related entities (but does not hide unrelated ones), served the same purpose for some participants.

Several features that were implicit in the tool's design were also found to be very useful. Labeling was considered important, as was saving of views. Animation between layouts was also found to help participants retain the context. The usefulness of such features is difficult or impossible to measure directly, emphasizing the importance of qualitative analysis.

The annotation feature was not used, probably because none of the tasks required annotation. However, the potential usefulness of annotations was indicated by one participant's request for tagging. Filtering was also underutilized, perhaps because participants were reluctant to remove important information. This appeared to leave participants in a conundrum because they also felt that the diagrams were too large and hard to understand when everything was visible.

Table IV summarizes the usefulness of features, using a subjective scale ranging between 'little used', 'useful', and 'critical'.

6.3.2. Feature improvements and new features

From participant requests and our observations in the user study, a number of variations to or improvements on existing features were identified. The user study also helped identify two new features: saving of session state and integrated static analysis. These feature improvements and proposed new features are summarized in Table IV.

Saving the state of a session was requested by participants so that they could explore a diagram and try different filters or searches, but be able to easily recover from unwanted results. This could be provided by a web browser style 'Back' button, by saving snap-shots of the diagram, or through an undo facility. Although it could be argued that this feature belongs under the 'Saving Views' feature, providing an edit history and navigation is a very different form of cognitive support from simply saving a view for later use. This is analogous to the difference between a version control system and the ability to save a file to disk. Saving state is not new to information visualization. Schneiderman [34] mention 'History' as one of the seven tasks in their task by data-type taxonomy. However, this functionality is not common exploration tools and is not found in the surveyed sequence diagram tools.

The fact that all participants found it necessary to repeatedly navigate between source code and sequence diagrams points to another opportunity for improved cognitive support. Sequence diagrams, as implemented, do not clearly show the logic of why a particular sequence is repeated, perhaps due to a loop or branch in the code. Guéhéneuc and Ziadi [35] propose augmenting dynamic

DOI: 10.1002/smr

sequence diagrams to show such branch and loop constructs through repeated trace capture or using information extracted from the static analysis. Based on our observations, this would be a useful feature.

6.3.3. Limitations

The current findings should be considered only as a first step towards developing a more in-depth understanding of required tool support for sequence diagram viewers. The small number of participants in the user study and their non-random selection (i.e. convenience sampling) limit our ability to generalize the results. Although the majority of participants had more than a year of professional programming experience, most were graduate students. This and their varying levels of experience with reverse engineering tasks add to the difficulty in generalizing results. A larger and less homogenous user group might have enabled us to find patterns of feature usage, which were not evident from the current results. It is also likely that different tool usage patterns might be observed in professional programmers with extensive reverse engineering experience. Finally, due to time constraints and the varying experience level of the participants, not all tasks were completed by all developers.

Although an effort was made to select realistic and challenging tasks, the time frame of the experimental sessions constrained us to select an application smaller than typical commercial systems. The generated traces were substantial—averaging over 10 000 calls—but traces from a more complex system would likely result in deeper call hierarchies and more complexity. Even with larger applications, trace size can be constrained by selecting a shorter trace capture interval.

The technology and tool used in the study also limit generalization of the results. Tool features were evaluated using a Java application, explored by a single visualization tool in an Eclipse environment.

7. RELATED STUDIES

This section considers related studies where user interaction with sequence diagram visualization tools has been evaluated. It does not explore the extensive evaluation of automated and semi-automated techniques for reducing trace complexity as this has been done elsewhere [11,12,23]. Some of the sequence diagram tools mentioned in Section 2 have been evaluated. *ISVis* [36] was the subject of a case study in which a single analyst used the tool to construct a partial architectural model consisting of 15 components by identifying approximately 50 interaction patterns in the NCSA Mosaic web browser. The intent of creating this model was to understand an existing area of the application to support a feature request related to that area. This understanding was achieved over 9 h and five separate sessions. The participant made use of a subset of the features including selecting, grouping-related utility methods, hiding low-level details, browsing a related overview, searching for patterns, and browsing related source codes.

The designers of *Shimba* and *SCED* [16] evaluated these tools' effectiveness in a series of tasks focused on understanding the class diagram editor of the *Fujaba* round trip engineering system. Questions asked covered behaviour of the software, runtime usage and dynamic control flow of a method, and various state-related questions. They do not provide a detailed description of the tool

DOI: 10.1002/smr

features used in the study but do mention the value of grouping repeated patterns and scrolling (both findings supported by our study). Despite techniques to reduce complexity, the size of the diagrams was sometimes overwhelming.

Pacione [31] and Pacione *et al.* [32] performed a comparison of a number of dynamic visualization tools including Borland's *Together*, *Jinsight*, and *jRMTool*, evaluating their capabilities to answer a series of large- and small-scale reverse engineering questions. A single user carried out the evaluation using the *JHotDraw* application. The overall conclusion was that no one tool was capable of supporting all tasks and some tasks were beyond the capabilities of all users regardless of the tools used (e.g. design pattern location). Our observations agree with these conclusions. Pacione *et al.* also found that tools that abstracted at the method and class/object level were most effective in answering the questions. The authors do not explicitly describe the use of tool features, implying that this should be the subject of further study. They do, however, mention some difficulties that hint at useful tool features. These include the inability to focus on a sub-sequence in the *Together* tool and being unable to link to a static view or drill down to see object state in several of the tools. Our study supports these observations, noting the importance of the ability to drill down into source code and the desire for a linked static view.

Zayour [33] evaluated both a technique and a tool (DynaSee) against what he referred to as a manual 'slicing' task. The author performed two user studies to evaluate the usefulness of the tool as a whole. Five software engineers who had some familiarity with the target application were assigned a set of comprehension tasks. The author found that bookmarks (annotations) greatly facilitated locating events in code following analysis that participants made use of search facilities even on smaller traces, and that method naming had a major influence on how useful a trace was for analysis tasks (poor naming forced the participant to browse code). Pattern detection to support abstraction, ranking of routines to allow filtering of less relevant ones, and removal of repetitions (e.g. loops) were also useful features.

8. CONCLUSION AND FUTURE WORK

This paper presents the results of a tool review and user study of sequence diagram tool features that support exploration of large reverse-engineered sequence diagrams. The contributions of the work are as follows:

- a summary of state-of-the-art sequence diagram tool features;
- a pluggable and extensible tool—the OSE—that implements these features;
- an experiment and focus group that formally evaluate the importance of these features in a variety of reverse engineering tasks; and
- a discussion of ideas to improve cognitive support in reverse-engineered sequence diagram tools.

This study addressed three research questions. The summary of state-of-the-art tools was directed towards answering question 1 'What kinds of interaction and presentation features do state-of-the-art tools provide for exploring sequence diagrams?' We discovered that current tools tend to implement many different features in varied ways, but it is rare for those features to be formally evaluated. We developed the OSE and conducted an experiment to answer Question 2 'How can

DOI: 10.1002/smr

sequence diagrams play a role in understanding the behaviour of a software system during reverse engineering?' We discovered that the features intuited by previous researchers are useful in reverse engineering tasks, but we also found that heavy use of a feature does not necessarily mean it (or the tool) helps to solve a task. Repeated use may actually be a sign of frustration on the part of the user. The user study also helped to answer Question 3 'How can sequence diagram tools be improved?' Our participants requested several improvements including the ability to save the state of the session in order to recover from an operation (similar to an undo), or to return later to it (similar to a bookmark). It was also obvious from observing the participants that they needed to explore the source code and the sequence diagrams at the same time. We believe that tight integration between source code and sequence diagrams is an important topic that needs to be explored.

We expect to extend our tool to address the received feedback. We also hope to integrate this study with complementary research to work towards building tools that provide richer options for multiple linked views and more powerful processing of reverse-engineered software. Finally, we would like to perform a larger study with more professional reverse engineers on larger, more realistic tasks.

We believe that the work presented in this paper provides useful insights on how individuals deal with large traces, and will guide us—and others in the community—as we design, implement, and evaluate this class of potentially powerful tools.

ACKNOWLEDGEMENTS

We are grateful to Chris Callendar for supplying the target application for our study and to the DRDC Valcartier employees and all the participants who took part in the study. Their help was and is invaluable to us. Copyright © Her Majesty the Queen in Right of Canada as Represented by the Minister of National Defence (2008). This copyrighted work has been created for Defence Research and Development Canada, an agency within the Canadian Department of National Defence; contract/grant number: W7701-5-2677/001/QCL.

REFERENCES

1. Kruchten P. The '4+1' view model of software architecture. *IEEE Software* 1995; **12**(6):42–50.
2. Hamou-Lhadj A, Lethbridge TC. A survey of trace exploration tools and techniques. *CASCON '04*: *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press: Markham, Canada, 2004; 42–55.
3. Hamou-Lhadj A, Lethbridge TC. Challenges and requirements for an effective trace exploration tool. *IWPC '04*: *Proceedings 12th IEEE International Workshop on Program Comprehension*. IEEE Computer Society: Bari, Italy, 2004; 70–78.
4. Cornelissen B, van Deursen A, Moonen L, Zaidman A. Visualizing test suites to aid in software understanding. *CSMR '07*: *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society: Amsterdam, Netherlands, 2007; 213–222.
5. Kern J, Garrett C. Effective Sequence Diagram Generation: Effective Use of Options with Borland Together Technologies. http: www.borland.com/resources/en/pdf/white_papers/20263.pdf June 2003 [7 July 2008].
6. Eisenbarth T, Koschke R, Simon D. Aiding program comprehension by static and dynamic feature analysis. *International Conference on Software Maintenance* (*ICSM*). IEEE Computer Society: Florence, Italy, 2001; 602–611.
7. Chen K, Rajlich V. Case study of feature location using dependence graph. *Proceedings of the 8th International Workshop on Program Comprehension* (*IWPC*). IEEE Computer Society: Limerick, Ireland, 2000; 241–247.
8. Walenstein A. Cognitive support in software engineering tools: A distributed cognition framework. *PhD Thesis*, Simon Fraser University, BC, Canada, May 2002.
9. Sharp R, Rountev A. Interactive exploration of UML sequence diagrams. *Third International Workshop on Visualizing Software for Understanding and Analysis* (*VISSOFT*). IEEE Computer Society: Paris, France, 2005; 8–13.

10. Charland P, Dessureault D, Lizotte M, Ouellet D, Nécaille C. Using software analysis tools to understand military applications: A qualitative study. *Technical Memorandum DRDC Valcartier TM 2005-425*, Defence Research & Development Canada—Valcartier, QC, Canada, June 2006; 73.
11. The Eclipse Foundation. Eclipse—An Open Development Platform. http://www.eclipse.org [7 July 2008].
12. The Eclipse Foundation. Using UML2 Trace Interaction Views. http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.tptp.platform.doc.user/tasks/tesqanac.htm [7 July 2008].
13. Hamou-Lhadj A, Lethbridge TC, Fu L. SEAT: A usable trace analysis tool. *IWPC '05*: *Proceedings of the 13th International Workshop on Program Comprehension*. IEEE Computer Society: St Louis MO, U.S.A., 2005; 157–160.
14. Koskimies K, Mössenböck H. Scene: Using scenario diagrams and active text for illustrating object-oriented programs. *Proceedings of the 18th International Conference on Software Engineering*. IEEE Computer Society: Berlin, Germany, 1996; 366–375.
15. Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W. *Object-oriented Modeling and Design*. Prentice-Hall: Englewood Cliffs NJ, U.S.A., 1990; 500.
16. Systä T. Understanding the behavior of java programs. *Proceedings of the 7th Working Conference on Reverse Engineering* (*WCRE*). IEEE Computer Society: Brisbane, Australia, 2000; 214–223.
17. Pauw WD, Lorenz D, Vlissides J, Wegman M. Execution patterns in object-oriented visualization. *Proceedings of the Conference on Object-oriented Technologies and Systems* (*COOTS '98*). USENIX: Santa Fe, U.S.A., 1998; 219–234.
18. Jerding DF, Stasko JT, Ball T. Visualizing interactions in program executions. *Proceedings of the 19th International Conference on Software Engineering*. ACM Press: Boston MA, U.S.A., 1997; 360–370.
19. McGavin M, Wright T, Marshall S. Visualisations of execution traces (VET): An interactive plugin-based visualisation tool. *Proceedings of the 7th Australasian User Interface Conference* (*AUIC '06*). Australian Computer Society: Hobart, Australia, 2006; 153–160.
20. Walker RJ, Murphy GC, Freeman-Benson B, Wright D, Swanson D, Isaak J. Visualizing dynamic software system information through high-level models. *SIGPLAN Notices* 1998; **33**(10):271–283.
21. Pauw WD, Jensen E, Mitchell N, Sevitsky G, Vlissides JM, Yang J. Visualizing the execution of java programs. *Revised Lectures on Software Visualization*, *International Seminar*. Springer: London, U.K., 2001; 151–162.
22. Salah M, Denton T, Mancoridis S, Shokoufandeh A, Vokolos FI. Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences. *ICSM '05*: *Proceedings 21st IEEE International Conference on Software Maintenance*. IEEE Computer Society: Budapest, Hungary, 2005; 155–164.
23. Riva C, Rodriguez JV. Combining static and dynamic views for architecture reconstruction. *CSMR '02*: *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*. IEEE Computer Society: Budapest, Hungary, 2002; 47–55.
24. Hamou-Lhadj A, Lethbridge T. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. *ICPC '06*: *Proceedings of the 14th IEEE International Conference on Program Comprehension*. IEEE Computer Society: Athens, Greece, 2006; 181–190.
25. Storey MA, Cheng LT, Bull I, Rigby P. Shared waypoints and social tagging to support collaboration in software development. *CSCW '06*: *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*. ACM Press: Banff, Canada, 2006; 195–198.
26. IBM. Rational Rose Enterprise. http://www-306.ibm.com/software/awdtools/developer/rose/enterprise [7 July 2008].
27. Omondo. EclipseUML. http://www.eclipsedownload.com [7 July 2008].
28. Borland. Together. http://www.borland.com/together [7 July 2008].
29. Bennett C, Myers D, Storey MA, German D. Working with 'monster' traces: Building a scalable, usable sequence viewer. *Proceedings of the 3rd International Workshop on Program Comprehension Through Dynamic Analysis* (*PCODA*). *Technical Report TUD-SERG-2007-022*. Delft University of Technology: Vancouver, Canada, 2007; 1–5.
30. Christl A, Koschke R, Storey MA. Automated clustering to support the reflexion method. *Information and Software Technology* 2007; **49**(3):255–274.
31. Pacione MJ. A novel software visualisation model to support object-oriented program comprehension. *PhD Thesis*, University of Strathclyde, Glasgow, Scotland, November 2005.
32. Pacione MJ, Roper M, Wood M. A comparative evaluation of dynamic visualisation tools. *Proceedings of the 10th Working Conference on Reverse Engineering* (*WCRE*). IEEE Computer Society: Victoria, Canada, 2003; 80–89.
33. Zayour I. Reverse engineering: A cognitive approach, a case study and a tool. *PhD Thesis*, University of Ottawa, Ottawa, Canada, May 2002.
34. Shneiderman B. The eyes have it: A task by data type taxonomy for information visualizations. *IEEE Visual Languages*. IEEE Computer Society: Silver Spring MD, U.S.A., 1996; 336–343.
35. Guéhéneuc YG, Ziadi T. Automated reverse-engineering of UML v2.0 dynamic models. *Proceedings of the 6th ECOOP Workshop on Object-oriented Reengineering* (*WOOR*), 2005. Unavailable in print. Available at: http://www.iro.umontreal.ca/~ptidej/Publications/Documents/ECOOP05WOORb.doc.pdf [7 July 2008].
36. Jerding D, Rugaber S. Using visualization for architectural localization and extraction. *Proceedings of the Fourth Working Conference on Reverse Engineering* (*WCRE*). IEEE Computer Society: Amsterdam, The Netherlands, 1997; 56–65.

DOI: 10.1002/smr

Chris Bennett is a Masters student and research assistant in the Computer-Human Interaction and Software Engineering Lab at the University of Victoria. His research focuses on developing effective tool support for reverse engineering tasks including understanding of assembly language and general program understanding.

Del Myers is a research assistant with the Computer-Human Interaction and Software Engineering Lab at the University of Victoria. His research varies in the area of software engineering and he has contributed to works advancing knowledge about tool support for novice programmers, model-independent visualizations, and social tagging in software engineering. His most recent work is in the area of creating effective, reusable sequence diagram visualizations for reverse engineering tasks.

Dr. Margaret-Anne Storey is an associate professor of computer science at the University of Victoria, a Visiting Scientist at the IBM Centre for Advanced Studies in Toronto and a Canada Research Chair in Human Computer Interaction for Software Engineering. Her research goal is to understand how technology can help people explore, understand and share complex information and knowledge. She applies and evaluates techniques from knowledge engineering, social software and visual interface design to applications such as collaborative software development, program comprehension, medical ontology development, and learning in web-based environments.

Dr. Daniel M. German is associate professor at the University of Victoria. His main areas of research are software evolution and open source software engineering.

DOI: 10.1002/smr

David Ouellet is a computer scientist at Defence Research and Development Canada – Valcartier in the System of Systems Section. He received his BEng degree in Computer Engineering (Software) from the Royal Military College of Canada. His research interests are architecture recovery, software comprehension and visualisation.

Martin Salois graduated in computer science from Laval University, Quebec, Canada, in 1997 and finished a Masters degree there in 1999, in collaboration with Defense Research and Development Canada – Valcartier, where he has been working since as a scientist. The subject of his Masters thesis and his current work is the detection and prevention of malicious code in software. He is also interested in software visualization and understanding.

Philippe Charland is a Defence Scientist at Defence Research and Development Canada – Valcartier, in the System of Systems Section. He received his BSc and MSc degrees in Computer Science from Concordia University. His research interests are software architecture recovery, program comprehension, and software maintenance.

DOI: 10.1002/smr