Working with 'Monster' Traces: Building a Scalable, Usable Sequence Viewer

Chris Bennett
University of Victoria
cbennet@uvic.ca

Del Myers
University of Victoria
delmyers.cs@gmail.com

Margaret-Anne Storey
University of Victoria
mstorey@uvic.ca

Daniel German
University of Victoria
dmg@uvic.ca

Abstract

In this position paper, we survey and identify tool features that provide cognitive support for reverse engineering and program comprehension of very large reverse engineered sequence diagrams. From these features we synthesize user requirements for a sequence diagram viewer, to which we add system requirements such as memory and processing scalability. We briefly describe a pluggable sequence viewer that meets these requirements and discuss some open questions that we are currently exploring.

1. Introduction

Sequence diagrams are an aid to understanding system behaviour in the form of scenarios (the +1 view in Krutchen's 4+1 architectural view model [1]). While originally devised as a notation to capture scenarios during analysis and design, sequence diagrams can also aid understanding of existing software through visualization of execution call traces. Their power lies in their ability to represent selected behaviour at a suitable level of abstraction. As Kruchten notes [1], scenarios illustrate how elements from the four primary architectural views come together, highlighting the most important requirements of a system.

Reverse engineered sequence diagrams based on call traces are typically huge, sometimes running to thousands or even hundreds of thousands of calls. Designing tools that help the user cope with the size and complexity of such traces is a major problem. In addition, tools need to be able to physically handle such traces within the memory and processing constraints of typical computers. Approaches to address these issues include reducing information overload through pre-processing, support for presentation and user interaction, and techniques to deal with partial sequences. Automatically reducing arbitrary traces to a manageable size is probably not realistic. Consequently, effective user interaction that allows the user to reduce clutter, navigate the sequence, and focus on relevant details is critical.

This position paper is structured as follows. Section 2 provides a brief background to research on reverse engineered sequence diagrams. In Section 3 we describe presentation and interaction features of sequence diagram viewers as derived from the research literature. In Section 4 we identify related cognitive support requirements and categorize features in terms of the cognitive support they provide. In Section 5 we address system requirements (such as scalability and performance issues) that arise from the huge volume of information contained in a reverse engineered sequence trace. We end with a brief discussion of a sequence viewer we designed (called Zest) and propose future work.

2. Background

Sequence diagrams used in reverse engineering can be abstracted at various levels including statement, object, class, architectural, and inter-thread [2]. Statement-level diagrams include intra-procedural calls and typically make use of an extended notation that supports conditions, loops, and branches (e.g., see [3]). High-level sequence diagrams are typically used as an aid to program understanding (e.g., see the work of [4] on filtering utility methods to reduce trace complexity). Sequence diagrams can be created through static or dynamic analysis, the advantages of the latter being increased precision, control over inputs, as well as resolution of polymorphic behaviour and runtime binding in object-oriented languages [2]. Regardless of the creation method or abstraction level represented on a diagram, there is a need to cope with large amounts of reverse engineered data. This 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), and automatic detection of utility functions (using fan-in/fan-out metrics) [5]. Other pre-processing techniques include removing abstract operation calls [5], hiding constructors and getters/setters [6], and limiting the depth of the call tree [5,6]. While pre-processing may be necessary to reduce the complexity of a sequence, considerable tool support is needed to help the user explore and understand the resulting diagram. We refer to this category of tool support as "cognitive support" - support that allows the user to offload some of their cognitive processing, such as their need to memorize details or to perform tedious calculations that the tool could do for them [8].

3. Presentation and Interaction Features

We divide sequence diagram user interface features into two categories 1) presentation or display facilities, and 2) features that allow the user to interact with and explore the diagram. We note that there may be overlap between presentation and interaction features, presentation often being both the result of interaction and a necessary precursor to it (e.g., highlighting and hiding could be considered interaction as well as presentation features). In the next subsections, we survey a number of reverse engineering tools that display sequence diagrams, summarizing their common presentation and interaction features.

3.1. Presentation

We first consider the presentation features these tools provide. Presentation concerns 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.

3.1.1. Layout. Perhaps the most important presentation feature is the layout of sequence diagrams according to some notational standard. Many research tools use their own layout format or some variation on a standard format (e.g., UML 2.1), perhaps adding proprietary extensions to address a specific problem (e.g., how to capture conditional branches). *Scene* [9] produces sequence diagrams according to Rumbaugh's OMT notation [10]. *SCED* [11] uses its own UML-like notation that provides constructs for nested sub-scenarios and repetition. *TPTP* [12] also uses UML.

3.1.2. Multiple Linked Views. It is often necessary to provide multiple views [1] as well as an overview of an underlying model. 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* [13] adopts an approach to viewing sub-trees, whereby a subtree may be rendered using a number of alternative 'charts', including a static class list or a class communication graph. *SCED* supports sequence diagrams as well as 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* [7] 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* [14] provides a two-window scenario view consisting of an information mural overview and a temporal message-flow diagram and *Scene* displays a summary call matrix view alongside a sequence view.

3.1.3. 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 usually use highlighting to indicate selection. Highlighting can go beyond single components to show related objects or messages.

3.1.4. Hiding. Hiding selected information is commonly used for controlling complexity in sequence diagram tools. Hiding supports 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* [15] hides 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. 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) [5]. The authors in [6] propose hiding null return values or abbreviating return values and parameter lists.

3.1.5. 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 (hidden) under the bevelled component. *TPTP* uses colour to indicate the length of time spent inside a method execution.

3.1.6. 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*).

3.1.7. 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 and *AVID* supports animation between component groupings.

3.2. Interaction

Interaction features allow the user to communicate with the tool while they navigate, query, and manipulate the sequence diagram to improve their understanding.

3.2.1. Selection. Manual selection of elements is a prerequisite for further interaction such as

manipulation, filtering, and slicing. This is supported by most tools.

3.2.2. Navigation. Rapid, simple movement between components (traversing the call tree) is important to usability [5] as is the ability to move between instances of the same type of pattern (e.g., subscenarios) in tools that support grouping of similar patterns (e.g., *SEAT*).

3.2.3. Focusing. User focusing has been identified as a problem when dealing with large traces [2]. The authors of the *Scene* tool 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.

3.2.4. Zooming and Scrolling. Zooming and scrolling are standard techniques to cope with more information than can be legibly shown in a single window. *VET*, *Ovation*, *TPTP* and *Jinsight* [16] support zooming and scrolling [2].

3.2.5. Queries and Slicing. Queries identify and optionally filter information within a sequence. *Scenariographer* [17] supports both relational SQL and set-based SMQL (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 method type or time range. While these are more limited than language-based queries they provide a much simpler solution. Slicing can be performed on either objects or methods and is a specific form of query that selects only objects or methods related to the selected component (a slice through the sequence flow).

3.2.6. 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.* [6] 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 summary message (vertical compaction). Grouped items can also be annotated with a label (and optionally comments) describing the grouped abstraction. Riva and Rodriguez propose a technique to collapse packaging activations within these packages [18]. In addition to preprocessing to detect repeating patterns, interaction support can allow manual selection and collapsing of repeated patterns such as loops. *TPTP* supports grouping of life lines using pre-determined levels of abstraction (host, process, thread, class, and object), grouping of method calls, and arbitrary user-defined groupings.

3.2.7. Annotating. Annotating can be used for many purposes: to describe why components were grouped [4], to capture user understanding during exploration of a sequence diagram, and to provide waypoints [19] and messages to oneself and others when the diagram is to be shared. Few tools provide annotation mechanisms, but our initial experiences show this to be a useful feature.

3.2.8 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 it can be restored at a later time. Together with annotations, a saved view can tell a story about the diagram being visualized. In [5] the authors discuss the need to save both the original trace and the transformations that were applied to reduce its complexity, although saving a record of user interactions is not discussed.

4. Cognitive support requirements for tools that present very large sequence diagrams

Even after preprocessing, interacting with and understanding a reverse engineered sequence diagram can be a daunting task. Tools should provide cognitive support for the user to effectively and efficiently explore and interact with the sequence diagram view. Through our experiences developing and using customized sequence diagram views, and an extensive review of the literature, we have synthesized six high level cognitive support requirements that these tools should meet: (1) The tool needs to **present a diagram that is intuitive and coherent** to the end user. Since these diagrams are typically large and screen space is limited, the layouts need to use available visual attributes, such as position, size and color effectively and efficiently. (2) The tool should present **multiple perspectives** of the underlying sequence. It may be necessary to display a related static view (e.g., a class diagram) in addition to the dynamic sequence view, or some combination of the two. (3) The user needs to be able to **navigate the diagram** and explore a focus area or navigate to other elements on or off the screen. During navigation, the tool should help the user maintain context and help build and maintain a mental model of the navigated sequence. (4) Since sequence diagrams are typically very large, the user needs tool assistance as they **filter and drill down** on the salient features they wish to understand. Filtering can be supported through interactive querying techniques and presentation facilities for hiding information. (5)

Related to filtering, the user may need to **abstract details** in the viewer**.** This will remove visual details but maintain some visual cues on the abstractions created during the understanding process. (6) **Documenting the user's understanding** for future use or to share with colleagues is also an important feature.

Hamou-Lhadj *et al.* have also discussed high level user requirements, specifically requirements to support exploration, abstraction and filtering [2].

In Table 1 we map the tool features identified in Section 3 with these cognitive support requirements. The main advantage of this approach is that it organizes requirements into different groups, linking each tool feature with a clear cognitive support goal. This mapping may also be useful when comparing tools that might not have the same feature set, but attempt to solve similar problems. In particular, we have used this table to identify and prioritize the features of the Zest sequence diagram viewer (described below).

Cognitive Support Requirements	Presentation and Interaction Tool Features
1. Visualize diagram	• Layout (positioning) • Visual attributes such as Colour and shape • Labels
2. Multiple perspectives	• Multiple and linked views (e.g., overview views, split panes, static and dynamic views)
3. Navigating (while maintaining context)	• Selection • Highlighting • Focusing • Multiple and linked views • Zooming • Scrolling
4. Filtering	• Querying • Hiding information
5. Abstracting	• Grouping • Annotating
6. Documenting (e.g., for sharing)	• Annotating • Saving views

Table 1: Mapping presentation and interaction features to the cognitive support requirements for sequence diagram views

5. System requirements for coping with very large sequence diagrams

While computer systems continue to increase in processing and memory capabilities, large diagrams of any kind can be taxing on even very powerful machines. This leads to the question of whether it is even possible to render the diagrams that we would like to see. With the right optimizations, many of the interaction features previously described can reduce memory load and improve performance. Techniques such as lazy-loading of visual elements can be combined with grouping and filtering. However, trade-offs between performance and memory requirements must then be made and it is difficult to find an optimal solution.

Large diagrams require massive amounts of memory to render – sometimes more than is available with, for example, a Java virtual machine. Caching visible pages for the display can help, but it is not obvious if it is useful to display more information than a modern machine can handle at one time. The cognitive load on the human may be the limiting factor.

6. The Zest Sequence viewer

In the previous sections, we synthesized a list of features and requirements that are needed to build a general, scalable sequence diagram viewer that can be used across different applications. In order to explore the effectiveness and completeness of this list, we developed and are now evaluating the Zest Sequence Viewer (see Figure 1).

The Zest Sequence Viewer was designed from the outset to be easily pluggable into various end-user applications. The viewer is written in Java, using the SWT framework [20], so it can be plugged into any SWT application. We have explored using it as a viewer for visualizing dynamic program traces and for visualizing debug stack states. The Zest Sequence Viewer has been used to load upwards of a thousand objects, but trace size is limited by the memory required

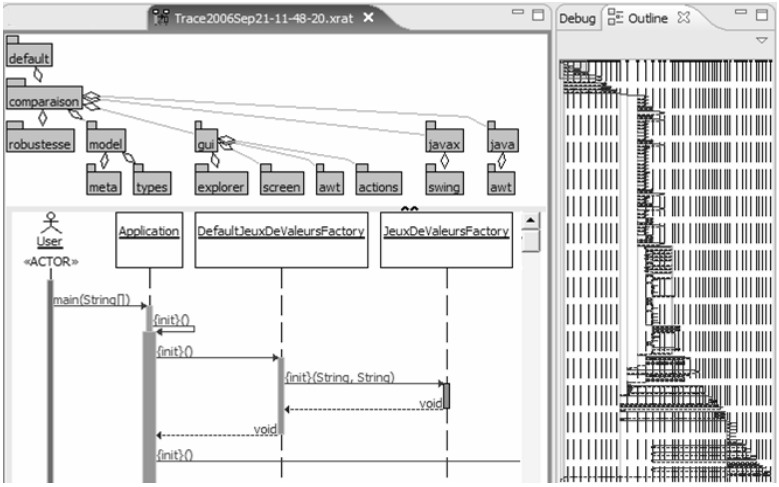

Figure 1: A portion of a sequence diagram in the Zest Sequence Viewer with an overview of the sequence on the right

to render large drawing areas. Such graphs can require hundreds of megabytes of memory, and may be larger than the Java virtual machine will allow.

7. Discussion

Our preliminary exploration has demonstrated the usefulness of the Zest Sequence Viewer. It has also helped us understand important requirements and tool features. However, more research must be done on the limitations of visualizing large sequences. A number of questions need to be resolved, e.g., what is the limiting factor: computer memory or human cognitive load? What kinds of visual inconsistencies can users cope with when displaying an incomplete sequence (e.g., changes in layout, hiding of visual elements)? Are humans able to understand and/or remember what elements have been hidden from the view? If not, what additional support can we provide for this? We are currently designing a case study that will involve observing professionals in their reverse engineering tasks using the Zest Sequence Viewer. We wish to observe their response to the viewer so that we can evaluate its usefulness and determine human factors in understanding sequence traces. We expect the results from this case study to further inform the cognitive support requirements for sequence diagram viewers.

Acknowledgments

We are grateful to Martin Salois, David Ouellet and Philippe Charland of Defence Research and Development Canada (DRDC) for their input into and review of this work. This work was funded by DRDC contract W7701-52677/001/QCL.

References

[1] P. Kruchten, “The 4+1 view model of architecture” *IEEE Software*, vol. 12, no. 6, Nov. 1995, pp. 42-50.

[2] A. Hamou-Lhadj and T. C. Lethbridge, “A survey of trace exploration tools and techniques”, in *Proceedings of the 2004 Conf. of the Centre For Advanced Studies on Collaborative Research,* IBM Press, 2004, pp. 42-55.

[3] A. Rountev, O. Volgin, and M. Reddoch, “Static control-flow analysis for reverse engineering of UML sequence diagrams”, *SIGSOFT Softw. Eng. Notes* 31, 1, Jan. 2006, pp. 96-102.

[4] A. Hamou-Lhadj and T. Lethbridge, “Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system”, in *Proceedings of the 14th IEEE international Conference on Program Comprehension*, Washington, DC, 2006, pp. 181-190.

[5] A. Hamou-Lhadj, T.C. Lethbridge, and L. Fu, "Challenges and requirements for an effective trace exploration tool", in *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, Washington, D.C., 2004, pp. 70- 78.

[6] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman, "Visualizing test-suites to aid in software understanding", in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, 2007, pp. 213-222.

[7] A. Hamou-Lhadj, T. C. Lethbridge, and L. Fu, “SEAT: a usable trace analysis tool”, in *Proceedings of the IEEE 13th international Workshop on Program Comprehension*, Washington, DC, 2005, pp. 157-160.

[8] A. Walenstein, “Cognitive support in software engineering tools: a distributed cognition framework”, Ph.D. dissertation, Simon Fraser University, B.C., Canada, 2002, p. 87.

[9] K. Koskimies and H. Mössenböck, “Scene: using scenario diagrams and active text for illustrating object-oriented programs“, *Proceedings of the IEEE 18th international Conference on Software Engineering* , Washington, DC, 1996, pp. 366-375.

[10] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, 1990, Prentice Hall.

[11] T. Systä, “Understanding the behavior of Java programs”, *Proceedings of the Seventh Working Conference on Reverse Engineering* , IEEE Computer Society, Washington, DC, 2000, p. 214.

[12] The Eclipse Foundation, “Help—eclipse SDK: using UML2 trace interaction views”, http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.tptp.platform.doc.user/tasks/tesqanac.htm [Sept. 2007]

[13] W. DePauw, D. Lorenz, J. Vlissides, and M. Wegman, “Execution patterns in object-oriented visualization”, in *Proceedings Conference on Object-Oriented Technologies and Systems* , USENIX, 1998, p. 9.

[14] D. Jerding, J. Stasko, and T. Ball, “Visualising interactions in program executions”, in *Proceedings of the 19th International Conference on Software Engineering*, Boston, USA, 1997, pp. 360-370.

[15] M. McGavin, T. Wright, and S. Marshall, “Visualisations of execution traces (VET): an interactive plugin-based visualisation tool”, in *Proceedings of the 7th Austr-alasian User interface Conference–Vol. 50*, 2006, pp. 153-160.

[16] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang, “Visualizing the execution of Java programs”, *Lecture Notes In Computer Science; Vol. 2269*, Springer-Verlag, London, 2001, pp.151-162.

[17] M. Salah, T. Denton, S. Mancoridis, A. Shokoufandeh, and F. I. Vokolos, “Scenariographer: a tool for reverse engineering class usage scenarios from method invocation sequences”, 2*1st IEEE Int. Conference on Software Maintenance*, 2005 pp. 155-164.

[18] C. Riva and J. V. Rodriguez, “Combining static and dynamic views for architecture reconstruction”, *Proceedings of the 6th European conference on Software Maintenance and Reengineering*, 2002, pp. 47-55.

[19] M. Storey, L. Cheng, I. Bull, and P. Rigby, “Shared waypoints and social tagging to support collaboration in software development”, *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work,* 2006, pp. 195-198.

[20] S. Northover and M. Wilson, *SWT: The Standard Widget Toolkit, Volume 1 (The Eclipse Series),* New York: Addison-Wesley Professional, 2004.