Challenges and Requirements for an Effective Trace Exploration Tool*

Abdelwahab Hamou-Lhadj
University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, Canada
ahamou@site.uottawa.ca

Timothy C. Lethbridge
University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, Canada
tcl@site.uottawa.ca

Lianjiang Fu
University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, Canada
lfu@site.uottawa.ca

Abstract

Building efficient tools for the analysis and exploration of large execution traces can be a very challenging task. Our experience with building a tool called SEAT (Software Exploration and Analysis Tool) shows that there is a need to address several key research questions in order to overcome these challenges. SEAT is intended to integrate several filtering techniques to tackle the size explosion problem that make traces hard to understand. However, the incorporation of these techniques into one efficient tool raises many issues. This paper focuses on these issues and underlies future research directions to advance the area of dynamic analysis of large software systems.

1. Introduction

Dynamic analysis is crucial for understanding the behaviour of a large system. Understanding an object-oriented system, for example, can be very hard if one relies only on the source code and static analysis. Polymorphism and dynamic binding, in particular, tend to obscure the relationships between the system artefacts.

Dynamic information is typically represented using execution traces. Although, there are different kinds of traces, this paper focuses on traces of routine (or method) calls. We use the term routine to refer to a function, a procedure, or a method in a class.

Most of the existing tools for analysing large execution traces of routine calls [4, 5, 13, 14, 20, 21] rely on specific visualization schemes to overcome the size explosion problem. They implement a set of techniques to facilitate the exploration of the trace content. These techniques include searching for specific components; hiding some interactions and so on. However, it is totally up to the user to combine them to get the desired understanding of the behaviour that is represented by the trace. We call these capabilities: Trace Exploration Techniques.

Our experience with analysing traces shows that they can contain thousands of calls even if they are generated from a small system. Such traces require advanced mechanisms for understanding them besides simple exploration.

Consider, for example, asking a designer to draw a sequence diagram that corresponds to a particular scenario and to compare the result with an execution trace that is generated from the system that implements this scenario. We will almost always see a surprising difference between the two representations in terms of the number of interactions that appear.

The research we are currently conducting at the University of Ottawa, with the collaboration of QNX Software Systems, is based on a different approach than simple trace exploration. We are more interested in discovering what can be *removed* from an execution trace to extract moderately sized high-level representations close to the sequence diagrams a designer would create. We call this process trace compression. The reader should not, however, confuse this notion of compression with standard data compression algorithms: The latter are purely concerned with saving space, and produce output that is unintelligible until uncompressed. Trace compression, while coincidentally saving space, has the goal of improving the intelligibility of its output.

In our previous work, we have presented several techniques [9, 10] that aim at achieving trace compression. However, we only experimented with them to assess the quantitative gain they attain.

Currently, we are working on assessing the qualitative aspects of these techniques. We would like to answer the following questions:

* This research is sponsored by NSERC, NCIT and QNX Software Systems

- Which compression techniques would be useful to software engineers to gain a rapid understanding of any given scenario?
- Which compression techniques can extract the most accurate high-level representations?
- If more than one technique is needed, how can we combine them to best achieve the desired goal?

For this purpose, we are prototyping a tool called SEAT (Software Exploration and Analysis Tool) that will be used for the experiments. However, during the analysis and design of SEAT, we have uncovered a number of requirements that raise very interesting research challenges.

This paper presents these challenges that need to be addressed in order to build an effective tool for dynamic analysis of large software systems.

The rest of this paper is organized as follows: We introduce SEAT in Section 2. In Section 3, we describe in more detail the features we would like to integrate in SEAT and the challenges that are derived from them. Finally, we present our conclusions and future directions.

2. Introduction to SEAT

SEAT is a prototype tool for exploring large execution traces. Figure 1 shows the overall flow of information using SEAT. The tool takes traces of routine calls represented in CTF (Compact Trace Format) [11] as an input and displays them using visualization techniques in a tree-like control window. To help the user extract useful information, SEAT implements several trace compression techniques. Some of these require the presence of the source code.

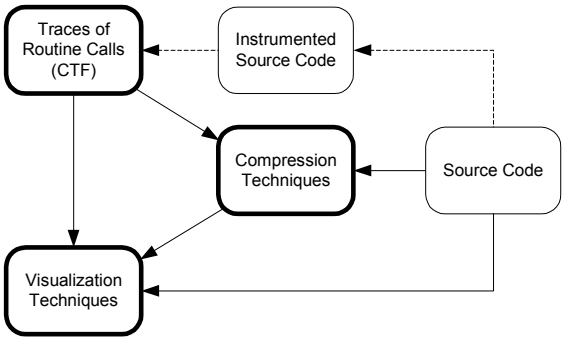

Figure 1. SEAT Data Flow

The dashed lines refer to one possible way for generating traces of routine calls, which is based on source code instrumentation. In practice, there are other techniques that can achieve the same goal. For example, one can instrument the execution environment in which the system runs (e.g. the Java Virtual Machine). This technique has the advantage of not modifying the source code. It is also possible to run the system under the control of a debugger, in which case breakpoints are set at strategic locations to generate events of interest. This last technique has shown to considerably slow down the execution of the system.

A snapshot of SEAT's graphical user interface is shown in Figure 2; it consists of three views: the exploration view; the trace view and the source code view. The interaction between these views is based on the twin-hierarchy user interface technique that was used in a tool called TkSee [26]. Figure 3 illustrates these interactions.

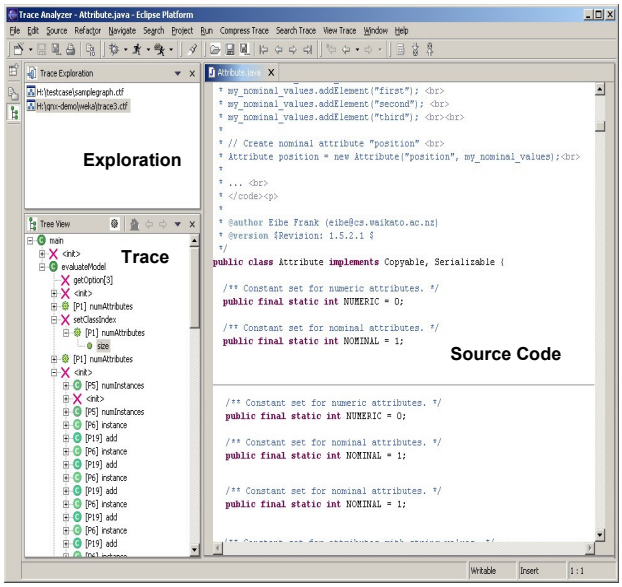

Figure 2. SEAT GUI

The analyst uses the exploration view to start a new exploration or navigate through the previous ones. Each exploration consists of analysing a particular trace which will be viewed as a tree structure using the trace view. At any time, the analyst can map the trace components to the source code (if it is available) to get more details

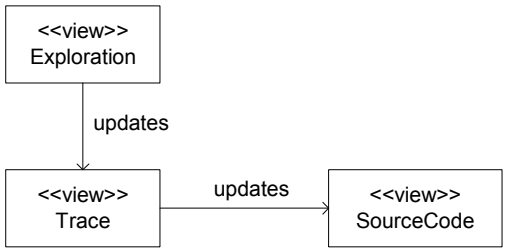

Figure 3. Interaction between SEAT views

Using SEAT, an analyst can compress the traces to the level where he or she can understand important aspects of their structure. This is done by applying several compression algorithms such as repetition removal, ignoring order of calls etc. The next section explains SEAT's actual and intended capabilities, along with the challenges in implementing them.

3. Challenges in building SEAT

In this section, we present the features that need to be integrated with SEAT and the challenges in building them with respect to three mains aspects, which we list here and explain in more details in what follows:

- Integrating different compression techniques
- Input/output mechanisms
- User interface design and usability issues

3.1 Compression Techniques

The aim of the compression techniques is to hide implementation details to reveal the main behavioural properties of the system. There are basically three types of compression techniques. Some of them are already implemented in SEAT and others represent significant challenges. There will be the ones that we will focus on in this subsection.

- Techniques based on pattern matching
- Techniques based on automatic detection of utility routines
- Techniques based on automatic detection of abstract operations

Pattern matching:

Pattern matching techniques are probably the most common techniques for overcoming the overwhelming size of traces. The idea consists of grouping similar sequences of events in the form of execution patterns (also called behavioural patterns). To compute similarity, several matching criteria have been used such as ignoring the order of calls, ignoring the number of repetitions, comparing two sequences of calls up to a certain depth and so on. A more detailed list of pattern matching criteria is described by De Pauw et al. in [5]. Implementing these criteria should not be an issue and most of them are supported by SEAT.

However, these techniques can only reduce the size of a trace if its subtrees satisfy the given matching criteria. This excludes subtrees that still exhibit signs of similarity but they do not fall into any matching scheme. For this purpose, we decided to augment this concept by integrating similarity metrics. Examples of these metrics include edit distance [22] and MoJo [24]. In fact, we want the tool to be flexible enough to be able to handle any metric. However, this new feature represents several challenges. One of the main challenges is to find the appropriate threshold for similarity.

To help the engineers to find an adequate threshold, an effective tool should be able to compute in advance an interval of which one bound indicates high similarity or even identity, while the other bound indicates very low similarity or complete dissimilarity. To illustrate this concept, let consider the following scenario: the user selects a subtree (e.g. by selecting its root) and requests from the system to show similar subtrees according to a chosen similarity metric. The result will be an interval, displayed in a usable way to the user, which he or she can dynamically adjust until the similarity value results in an appropriate amount of trace detail being displayed.

However, this technique may require extensive computation, since the space of the subtrees that need to be compared is very large. This problem gets worse if one considers the amount of disk access it might involve to retrieve parts of the trace.

One possible solution is to reduce the search space by considering only the subtrees that are initiated by the same routine. The motivation behind this is based on the observation that patterns are generally used to discover high-level domain concepts [13]. These concepts are usually triggered by the same routine. The tree structure that is derived from this routine may differ due to exceptional behaviours that occur during the execution of the scenario. However, this does not solve completely the problem and further research is needed.

Detecting utilities:

Many software systems use utilities that help the implementation of the system's functionality. A utility could be a routine, a class or even a subsystem. In a recent study conducted by Zayour [23] involving a large real world procedural telecommunication system, the author showed that not all of the procedures have the same degree of importance. Some procedures can be simple utilities (e.g. sorting an array) and removing them would not in most cases affect the comprehension process.

However, as the system documentation becomes obsolete, utilities go undetected and become mixed with the main system functionality. Therefore, there is a need to automatically detect them. As we are not aware of any tool that incorporates this capability, we started working on a metric for detecting utilities. This metric is based on

statistical data that needs to be extracted using the source code as well the execution traces. An example of such information includes routine call fan-in and fan-out, extracted from a static call graph. Other properties can be extracted from traces; examples include the number of occurrences of the routine in the trace, its position (whether it is a leaf or not), and the number of different parents a routine has (its dynamic fan-in). The rationale behind this is that something that is called from a lot of places is likely to be a utility, whereas something that itself makes many calls to other components is likely to be too complex and highly coupled to be considered a utility

Combining static and dynamic information is probably the best way to detect utilities. However, the extraction of such properties can be very expensive and sometimes impossible, which poses a real challenge for implementing an efficient algorithm for detecting utilities. Indeed, building an accurate static call graph for object-oriented systems is a hard task [8] due to the need for static resolution of polymorphic calls or the use of function pointers. On the other hand, relying only on dynamic information can be insufficient due to its incompleteness.

Detecting abstract operations:

One of the main activities in reverse engineering consists of extracting high-level views by abstracting out low-level implementation details. SEAT adheres to this principle by working towards extracting sequence diagrams (or other similar representations) from execution traces.

One of the key techniques that we use for this purpose is the automatic detection of abstract operations. We define an abstract operation as an operation that can be implemented in different ways depending on the context where it is defined. Polymorphism is a typical way for implementing abstract operations. Given the possibility to detect abstract operations, the analyst can reduce the size of the trace by hiding its implementation details. In [10], we showed that this technique can result in a significant reduction of the trace size.

However, if the concept of generalization is used as an implementation convenience rather than for representing abstract operations, which tends to be very common in practice, this can affect tremendously the results of the analysis. In addition to this, we want to make the technique applicable to procedural software systems as well. For these reasons, we started investigating other alternatives. By exploring the execution traces of several systems, we noticed that the routines that implement abstract operations tend to have similar names. For example, the following three routines are found in traces of a drawing editor, called XFig (a procedural system under UNIX): init_spline_drawing, init_ellipse_drawing and init_circle_drawing. According to their names, it is obvious that these routines are initializing drawing parameters for each of the mentioned shapes. The information about the fact that the "init" operation is applied to three different shapes can be very valuable to the user who is only concerned with what the system is doing instead if how it does it. Therefore, we decided to consider using naming conventions in addition to polymorphism (where it applies) to automatic detection of potential abstract operations

However, naming conventions come with a set of challenges. Sneed showed that software engineers do not necessary follow the naming conventions [18]. On the other hand some other researchers who have used naming conventions for other types of research such as recovering the system architecture using file names showed very promising results [1, 2, 3]. An efficient tool should consider these different parameters and assess the validity of naming conventions. In [2], Anquetil and Lethbridge presented a framework for assessing the relevance of a naming convention that can be used for this purpose.

3.2 SEAT Input/Output Mechanisms

Tools need to be able to efficiently input and output data in order to be adopted. SEAT uses a metamodel called CTF [11] for representing traces. The key idea behind CTF is that any rooted ordered labelled tree can be transformed into an acyclic ordered directed graph by representing identical subtrees only once [11], a technique that was first introduced by Downey et al. in [6] to develop efficient tools that work on tree structures. An efficient algorithm that does the transformation is presented in [7]. Our preliminary results show that this technique can achieve very high compression ratios [9]. However, when using CTF we encountered some interesting issues which we will discuss next.

Saving the data and the transformation rules:

We refer to data transformation as the process of taking input data in a certain format, applying to it a set of transformation rules and outputting a new set of data in the same format but with a different content. In the context of trace compression, the transformation rules are the compression techniques. Although, CTF is capable of supporting the trace that results after an original trace has undergone several compression schemes, it is not designed to save the transformation rules. That is, any tool that reads CTF can read the resulting trace but cannot retrieve the original. This is because the generic form of CTF places an obstacle to representing what is specific to

SEAT. This problem leads to interesting research questions with regards to metamodelling and exchange formats, which consist of the following:

- How can we make an exchange format general enough for the purpose it is designed for and specific enough to carry specific information?
- A metamodel represents the structure of the data that is manipulated; if this structure turns out to be dynamic, how can we represent it?

One solution to this problem is to agree on a set of operations that can be performed on a trace and design a more general metamodel for representing all the needed data. Assuming that the agreement takes place, this approach has the obvious drawback that new techniques will always require reviewing the metamodel.

Another approach is to have two metamodels. The first one will represent general information about traces of routine calls, which will be CTF as it is now. The second will be specific to SEAT and will be used to save the transformation rules. The disadvantage of this approach is that it prevents the tools from further integration and sharing features.

Supporting a variety of sources and destinations:

Considering the possible environments in which the user may use a reverse engineering tool, a variety of I/O sources and destinations need to be supported besides basic disk files. A very popular scenario is that the data comes from a stream, such as TCP/IP connection over a network. This allows dynamic analysis of executable programs: The program under study transmits the stream to the trace analysis tool, and the trace is never saved to disk. Another possible I/O exchange is using shared memory where the analysis tool reads data directly from memory to which the program under study writes. Supporting all these data sources and destinations adds a lot of flexibility to the analysis tool and can help the tool's adoptability, but represents a challenge for tool builders. In addition to this, an efficient tool should be flexible enough to read other file formats and allow converters to be plugged-in easily.

I/O Performance:

Performance of I/O is critical to the success of a trace analysis tool, since user will tend to be intolerant of systems with a poor response time. Since traces are very large, performance will always be a concern so that ever larger traces can be handled.

One of the performance issues relates to input and output of trace data. Although, the CTF metamodel is independent of any syntactic form, the tendency in the last few years converges towards using GXL [12] as a standard syntax for reverse engineering tools. The idea is that if all the tools can utilize GXL as a general format for their inputs and outputs, more interactions between them can be achieved.

However, a general XML format, such as GXL, often requires more processing than a tuned special format. XML will ultimately be parsed and processed to create an internal model. Performance of parsing XML poses an obstacle, especially for large data set. One study related to the performance of XML parsing showed that XML loading is 26 times slower than flat file with delimited format [16]. The situation is worsened with a DTD or schema validation. Therefore, compromise between generalization and efficiency needs to be decided.

3.3 User Interface Design Challenges

The final set of challenges we will explore relate to how to design a usable trace analysis tool.

I/O constraints on the user interface

As discussed earlier, trace data can be formidably large, so loading and manipulating it in memory can take excessive time unless some careful design choices are made.

Most user interface elements for displaying large amounts of information build a complete representation of the display in memory, and then make sections of it visible as the user scrolls using the scrollbar (or pages up and down). Standard list widgets in languages like Java work this way. Unfortunately, for the trace browsing tools we are creating, such a design is not workable. To start with, there can be just too many trace nodes to quickly create a list widget that the user can scroll. Secondly, when the user changes the parameters of any of the compression mechanisms discussed in Section 3.1, the entire display will need to be re-created, despite the fact that only a tiny fraction will be visible.

What is needed, therefore, is a new type of browsing widget that will generate the display for only that part of the trace that can currently be viewed. When the user scrolls or uses page up/down keys, a new part of the display must be quickly generated.

Our current solution in SEAT is to create a widget with a tree–like data structure representing what is visible (i.e. the real, not the virtual display), plus a small amount of additional context. This structure has links to the model data (our internal representation of CTF). When scrolling occurs in the widget, when the user 'opens' a subtree, or when parameters of the compression algorithms are

changed, some or all of this tree structure is regenerated. The objective is to give the user the impression he or she is navigating in a normal tree browser widget.

Because the compression techniques discussed earlier can easily filter out many nodes or even the whole currently-visible subtree, it is necessary to keep track of the 'most recent' nodes visible to the user. The widget also needs to support long scroll jumps (e.g. using the scrollbar), always quickly extracting data from model.

Rendering compressed nodes

Displaying the result of applying several compression schemes on the same subtree raises interesting usability issues. For example, suppose the user selects a subtree (e.g. by selecting its root) and then decides to hide the utilities it contains, ignore the order of calls of its subtrees and hide implementation details of its abstract operations. An efficient tool should be able to visually render the subtree to indicate these transformations. This typically involves the use of color-coding techniques or icons. However, considering the number of compression techniques and the way they can be combined, the choice of appropriate rendering techniques can represent a real challenge.

Another issue related to the problem discussed above is with regards to restoring the trace's original state. Imagine a user who first hides the implementation details of all abstract operations and then requests to hide all utilities. Some of the utilities were already hidden because they were among the implementation details of the abstract operations. The problem occurs when the user decides to restore all utilities without restoring the abstract operation details. It is obvious that some utilities stay hidden, which might very misleading to the user.

The reader can certainly imagine several other scenarios like this one. The overlapping nature of the compression techniques poses a real challenge for developing usable and efficient rendering techniques to keep the user informed of what is happening.

We currently do not have complete solutions to this class of problems. We will be experimenting with a variety of alternatives using SEAT.

Integration with development environments such as Eclipse

One factor that can significantly increase usability is integrating the trace browsing facilities with a standard integrated development environment. This allows software engineers who are doing maintenance to discover information using the trace browsing tool, and immediately edit the code based on the information they have found.

We have chosen to incorporate our tool into the Eclipse environment. Eclipse is a universal platform that can be extended by plugins [25]. It provides a framework for UI level integration and a large number of APIs to enable tools to work together. There has been much effort to improve the usability and accessibility of Eclipse's interface, and a set of guidelines are being developed for its developers. We discovered that, making a tool seamlessly integrate with other components of the platform so the user can have a common look and feel is a demanding task. We had to call upon an Eclipse expert to criticise our earlier UI designs; this allowed us to adjust some of the decisions we made so our tool would feel more Eclipse-like. There were some situations, however, where some decisions we made to specifically help those browsing traces, contradicted the advice we received about Eclipse integration. One example of this was ideas we had for setting up control panels to change the parameters of the compression algorithms.

4. Related Work

There are several other tools that developers can use to explore traces. These tools barely support trace compression as described in this paper. In this section we describe a few of these.

ISVis is a visualization tool that supports analysis of execution traces of object-oriented systems [13, 14]. ISVis is based on the idea, also incorporated into SEAT, that large execution traces are made of recurring patterns and that visualizing these patterns is useful for reverse engineering. The execution trace is visualized using two kinds of diagrams: the information mural and message sequence charts. The two diagrams are connected and presented on one view called the *scenario view*. The information mural uses visualization techniques to create a miniature representation of the entire trace that can easily show repeated sequences of events. Message sequence charts are used to display the detailed content of the trace.

To deal with the size explosion problem, ISVis uses an algorithm that detects patterns of identical sequences of calls [14]. Given a pattern, the user can search in the trace for an exact match, an interleaved match, a contained exact match (components in the scenario that contain the components in the pattern) and a contained interleaved match. The authors do not really motivate why these criteria are useful to understanding the trace. Additionally, the user can use wildcards to formulate more general search queries. The tool does not provide any guidance regarding which kinds of queries might lead to important

interactions. Another important feature of ISVis is that trace events can be abstracted out using the containment relationship. For example, a user can decide to hide the classes that belong to the same subsystem and only show the interaction between this subsystem and the other components of the trace.

De Pauw et al. introduce a variation of UML sequence diagrams called the execution pattern view [5]. This view lets the user browse the program execution, which consists mainly of traces of method calls, at various levels of detail. To overcome the size problem, similar sequences of events are shown as instances of the same pattern. However, the authors introduce many useful pattern matching criteria that can be used to decide when two sequences of events can be considered equivalent. Most of these criteria are supported by SEAT. These capabilities give the users more support in browsing traces as opposed to ISVis.

Richner and Ducasse present a tool, called Collaboration Browser that is used to extract collaboration patterns from traces of method calls [17]. A collaboration pattern consists of a repeated sequence of method calls. The matching criteria used in their approach are almost the same as those used by De Pauw et al. [5]. Additionally, Collaboration Browser provides a query mechanism that allows the user to search for interesting collaborations.

Systä presents a reverse engineering environment based on dynamic analysis to extract state machines from object-oriented systems [19, 20, 21]. Her approach is based on the use of SCED [15], a software engineering tool that permits representing execution traces in the form of scenario diagrams – Scenario diagrams are similar in semantics to UML sequence diagrams. SCED has also the ability to extract state machines from scenario diagrams. Systä deals with the size explosion problem the same way as the other tools presented so far do, i.e. detecting patterns of repeated sequences of events. However, Systä's approach considers exact matches only, which limits her approach to small execution traces only.

DynaSee is another reverse engineering tool that supports the analysis of traces of procedure calls of procedural software systems [23]. Besides the ability to detect patterns of procedure calls, the author noticed that not all procedures are equally important to the software engineer. Procedures at high level of the call tree are closer to application concepts, and those at bottom are implementation concepts. He used heuristics such as the number of occurrences, fan-in and fan-out to design a weighting function to rank procedure calls. Procedures with low weight are considered utilities and can be removed from the trace.

Some other tools such as Jinsight [4] provide many views to simplify the analysis of execution traces of object oriented systems. However, these tools are tuned to detect performance problems such as memory leaks and resource allocation. We do not think that they are useful to program comprehension.

Conclusions and Future Directions

In this paper, we presented several challenges that we have encountered while building a tool named SEAT (Software Exploration and Analysis Tool). The objective of SEAT is to help software engineers efficiently explore large execution traces.

Unlike other tools that are based on trace exploration techniques such as browsing the trace and searching for particular components, SEAT is based on the idea that in order to exhibit the main interactions that exist in the trace, we need to investigate what can be removed from it without affecting its overall content. We refer to such a process as trace compression. This paper discuses the main trace compression techniques and the challenges they represent in integrating them into one tool.

To overcome these challenges, the following key research questions need to be addressed:

1. The support of different similarity metrics to reduce the trace size by grouping similar subtrees together
2. Automatic detection of utilities by combining static and dynamic information such as static call graphs and execution traces.
3. Detecting abstract operations using the system naming conventions
4. Defining a metamodel to represent the data and the transformation rules that it undergoes to promote interoperability between tools
5. Supporting a variety of sources and destinations of data such as TCP/IP based input/output.
6. Supporting different data formats
7. Improving input/output performance to reduce the user response time while manipulating large traces.
8. Developing efficient rendering techniques for visualizing the result of various transformations of the trace components
9. Integrating SEAT with existing platforms such as Eclipse to have a uniform look and feel of the user interface

Some of these challenges require further research in the area of dynamic analysis. Taken together, the above

items are both research challenges, as well as requirements for an effective trace exploration tool.

We are currently working with QNX software engineers to validate our techniques using SEAT as a testbed to explore various solutions to the above challenges. The main objective is to find answers to the questions that were asked in the Introduction Section of this paper.

References

[1] N. Anquetil, and T. C. Lethbridge, "Recovering Software Architecture from the Names of Source Files", *Journal of Software Maintenance: Research and Practice, 11*, pp. 201-221.

[2] N. Anquetil, and T. C. Lethbridge, "Assessing the Relevance of Identifier Names in a Legacy Software System", *CASCON*, Toronto Canada, 1998, pp 213-222

[3] N. Anquetil, and T. Lethbridge, "Extracting Concepts from File Names; a New File Clustering Criterion", *International Conference on Software Engineering,* Japan, 1998, pp 84-93

[4] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, J. Yang, "Visualizing the Execution of Java Programs", *Lecture Notes In Computer Science, Revised Lectures on Software Visualization,* 2001, pp. 151-162

[5] W. De Pauw, D. Lorenz, J. Vlissides, M. Wegman, "Execution Patterns in Object-Oriented Visualization", *In Proc. of the 4th USENIX Conference on Object-Oriented Technologies and Systems, COOTS, Santa Fe, NM, April 1998,* pp. 219-234

[6] J.P. Downey, R. Sethi and R.E. Tarjan, "Variations on the Common Subexpression Problem", *Journal of the ACM.* 27(4), October 1980, pp. 758-771

[7] P. Flajolet, P. Sipala, J.–M. Steyaert, "Analytic Variations on the Common Subexpression Problem", *In Proc. Automata, Languages, and Programming*, *volume 443 of Lecture Notes in Computer Science*, Springer-Verlag, 1990, pp. 220-234

[8] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call Graph Construction in Object-Oriented Languages", *In Proc. of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Atlanta, Georgia, United States, 1997, pp. 108-124

[9] A. Hamou-Lhadj, T. C. Lethbridge, "Compression Techniques to Simplify the Analysis of Large Execution Traces", *In Proc. of the 10th International Workshop on Program Comprehension (IWPC), Paris*, 2002, pp. 159-168

[10] A. Hamou-Lhadj, and T. C. Lethbridge, "Techniques for Reducing the Complexity of Object-Oriented Execution Traces", *In Proc. of the 2nd Annual "DESIGNFEST" on Visualizing Software for Understanding and Analysis, co-located with the 19th International Conference on Software Maintenance (ISCM)*, Amsterdam, The Netherlands, 2003, pp. 35-40

[11] A. Hamou-Lhadj, and T. Lethbridge, "A Metamodel for Dynamic Information Generated from Object-Oriented Systems", *Electronic Notes in Theoretical Computer Science, from a paper presented at 1st International Workshop on Meta-models and Schemas for Reverse Engineering (ATEM), co-located with The 10th Working Conference on Reverse Engineering (WCRE)*, Victoria, Canada, 2003

[12] R. C. Holt, A. Winter, and A. Schürr, "GXL: Toward a Standard Exchange Format", *In Proc. of the 7th Working Conference on Reverse Engineering (WCRE)*, Brisbane, Australia, November 2000, pp. 162-171

[13] D. Jerding, and S. Rugaber, "Using Visualization for Architecture Localization and Extraction", *In Proc. of the 4th Working Conference on Reverse Engineering*, Amsterdam, Netherlands, October 1997, pp. 219-234

[14] D. Jerding, J. Stasko, and T. Ball, "Visualizing Interactions in Program Executions", *In Proc. of the 19th International Conference on Software Engineering (ICSE)*, Boston, Massachusetts, 1997, pp. 360-370

[15] K. Koskimies, T. Männistö, T. Systä, and J. Tuomi, "SCED: A Tool for Dynamic Modeling of Object Systems". *University of Tampere, Dept. of Computer Science, Report A-1996-4*, 1996

[16] M. Nicola, and J. John, "XML parsing: a threat to database performance", *In Proc. of the 12th International Conference on Information and Knowledge Management*, New Orleans, November 2003, pp. 175 - 178

[17] T. Richner, and S. Ducasse, "Using Dynamic Information for the Iterative Recovery of

Collaborations and Roles", *In Proc. of the 18th International Conference on Software Maintenance (ICSM)*, Montréal, Canada, 2002, pp. 34-43

[18] H. M. Sneed. "Object-oriented Cobol Re-cycling", *In Proc. of the 3rd Working Conference on Reverse Engineering*, Monterey, CA, November 1996, pp. 169-178.

[19] T. Systä. "Understanding the Behaviour of Java Programs", *In Proc. of the 7th Working Conference on Reverse Engineering (WCRE)*, Brisbane, Australia, November 2000, pp. 214-223

[20] T. Systä, K. Koskimies, and H. Müller, "Shimba – An Environment for Reverse Engineering Java Software Systems.", *Software Practice & Experience, Volume 31 Issue 4*, 2001, pp. 371-394

[21] T. Systä, "Dynamic Reverse Engineering of Java Software", *In Proc. of the ECOOP Workshop on Experiences in Object-Oriented Reengineering*, Lisbon, 1999, pp. 174-175

[22] K. C. Tai, "The Tree-To-Tree Correction Problem", *Journal of the ACM*, 26(3), 1979, pp. 422-433

[23] I. Zayour, "Reverse Engineering: A Cognitive Approach, a Case Study and a Tool", Ph.D. dissertation, University of Ottawa, 2002, www.site.uottawa.ca/~tcl/gradtheses/izayour

[24] V. Tzerpos, and R. Holt, "MoJo: A Distance Metric for Software Clustering", *In Proc. of the 6th Working Conference on Reverse Engineering*, Atlanta, October 1999, pp. 187-193

[25] Eclipse: http://www.eclipse.com

[26] TkSee: http://www.site.uottawa.ca/~tcl/kbre/options/intro.html