SEAT: A Usable Trace Analysis Tool[1]

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

Understanding the dynamics of a program can be made easier if dynamic analysis techniques are used. However, the extraordinary size of typical execution traces makes exploring the content of traces a tedious task. In this paper, we present a tool called SEAT (Software Exploration and Analysis Tool) that implements several operations that can help software engineers understand the content of a large execution trace. Perhaps, the most powerful aspect of SEAT is the various filtering techniques it incorporates. In our precious work, we showed that these techniques can reduce significantly the size of traces in order to reveal the main content they convey.

1. Introduction

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

In this paper, we introduce a trace analysis tool called SEAT (Software Analysis and Exploration Tool) that can be used by software engineers to understand the content of traces and hence the system behaviour.

There are several aspects that distinguish SEAT from the existing trace analysis tools such as the ones presented in [4]. First, SEAT is based on the Compact Trace Format (CTF) [3], which is a scalable exchange format for representing traces. Also, SEAT incorporates several filtering techniques that we showed in our previous work to be effective in reducing the size of traces and yet keep their main content. Many other features of SEAT do not simply exist in other systems.

The rest of this paper is organized as follows: The next section is a description of the overall architecture of SEAT. In Section 3, we show the main features of SEAT. In Section 4, we show how we will present the tool during the demo session.

[1] This research is sponsored by NSERC, NCIT and QNX Software Systems

2. Introduction to SEAT

SEAT is a trace analysis tool developed at the School of Information Technology and Engineering (SITE) of the University of Ottawa for exploring large execution traces of routine (or methods) calls. Figure 1 shows the overall flow of information using SEAT. The tool takes traces of routine calls as input and displays them using visualization techniques based on 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.

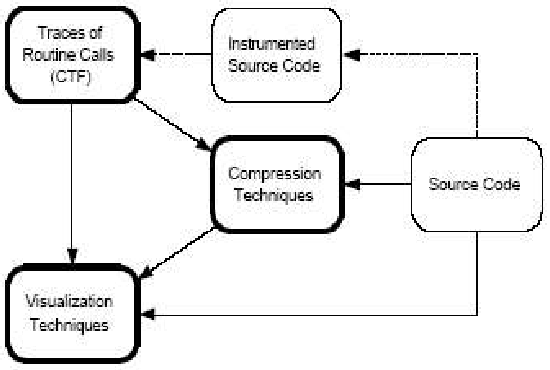


Figure 1. SEAT data flow

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.

SEAT user interface is based on the Eclipse platform and consists mainly of a multiple-page trace editor and a set of auxiliary views. The trace is displayed in the trace editor in the form of a tree structure. The auxiliary views are used to display different kinds of information such as patterns of execution, statistical data and so on. A snapshot of SEAT is shown in Figure 3.

3. SEAT Features

In this section, we present the main operations that are supported by SEAT:

3.1 Trace Exploration

With SEAT, a software maintainer can explore several traces at the same time. The traces can be put in the same Eclipse Perspective as the one containing the system that was used to generate the traces. This allows the maintainer to manipulate traces the same way as he or she would manipulate any other artifacts of the system.

3.2 Filtering the traces

SEAT supports numerous features for reducing the size of traces. The first set of techniques consists of allowing the users to hide specific components or group of components. The tool adds these components to a specific list called the 'utility list' that can be later used to shrink other traces generated from the same system. At any time, the user can have access to this list and restore any desired components.

In addition to this, SEAT supports several filtering algorithms that aim at reducing the size of the traces by removing unnecessary data. For example, imagine that the user wants to explore the content of the subtree of Figure 2a. One possible way to reduce the size of this subtree without loosing the main content it conveys is to apply three transformations. The first transformation consists of ignoring the number of contiguous repetitions of the routines. This will cause the two first subtrees of the subtree rooted at 'A' to collapse since they differ only due to the fact that 'D' is repeated contiguously. The second transformation is to ignore the order of calls so as to group the third subtree of 'A' with the two previous ones. Finally, a maintainer might decide to limit the depth of the subtree to two assuming that all routines that are executed after this level are utilities. SEAT will then ignore the routine 'E' and collapse the last subtree of 'A' with the others. The resulting subtree after these transformations is shown in Figure 2b. This powerful capability can also be combined with the "user-defined utilities" feature discussed previously to reach even better compaction.

Moreover, SEAT supports predefined filtering techniques that can be applied to object-oriented systems to hide some types of implementation details. These techniques include the automatic filtering of accessing methods, constructors, details of polymorphic operations, etc. The motivation behind considered these elements as implementation details is explained in [5].

In addition to this, SEAT is designed in a way that separates the filtering algorithms from the other artifacts of the tool. That is, adding a new filtering algorithm can be done very easily.

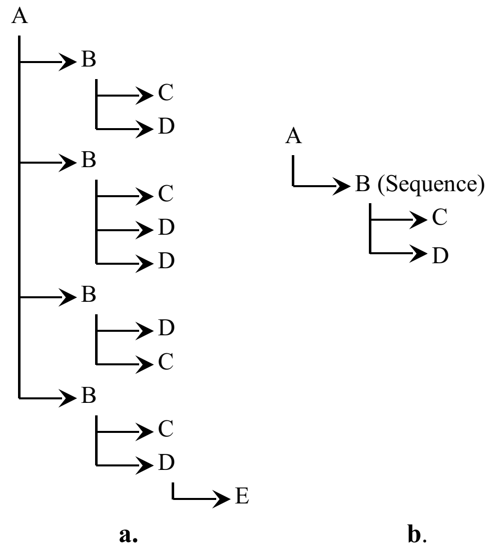


Figure 2. The tree in a) can be transformed into the one in b) using SEAT filtering algorithms

3.3 Pattern Detection

SEAT supports the automatic detection of patterns of execution –These are the similar subtrees that are repeated non-contiguously in the trace. The user is given the possibility to browse the list of detected patterns (that appears in the 'pattern view') and select the ones that need to be analyzed. Color-coded techniques are used to distinguish among the occurrences of a particular pattern in the tree structure. SEAT facilitates the navigation from one occurrence to another using special navigation buttons.

In addition to that, the filtering algorithms discussed in Section 3.2 can be used as matching criteria to detect patterns composed of similar occurrences that are not necessarily identical. In fact, SEAT supports many other matching criteria that have shown to be useful for understanding traces. Some of these criteria are described in [1]. This capability offers endless possibilities to the maintainers that will help them uncover the most useful patterns of execution.

3.4 Scalability

SEAT uses the Compact Trace Format (CTF) [3] to represent traces. CTF has been designed with the idea of scalability in mind. In our previous work [2], we showed that CTF uses around 5% of the total number of nodes of the initial trace to represent the whole trace, which makes SEAT scalable to extremely large traces.

Also, SEAT implements a sophisticated tree loading algorithm that loads into memory only the nodes that are being used, which facilitate greatly the exploration process.

3.5 Other Features

The tool supports various other features that are important for exploring traces such as searching for specific components using wildcards, mapping the content of the trace to the source code, etc.

In addition, SEAT provides a view that can be used by the maintainer to display various statistics about the content of the trace during the exploration process. Statistics can be obtained for the entire trace, specific subtrees, a specific routine, etc. The metrics used include the number of calls, number of distinct routines, the number of distinct subtrees, number of patterns, and many others. The complete list of metrics supported by SEAT is presented in [6].

The usability of the tool has been evaluated and various usability issues have been addressed. The tool is under Eclipse so it takes advantage of the same look and feel as any other application developed for the Eclipse platform.

4. Demo Session

During the demo session we will show how SEAT can be used to analyze sample execution traces. The focus will be on applying the various filtering techniques to vary the amount of information displayed in order to abstract out the most important content.

We will also discuss the other features of SEAT and its overall effectiveness to understand the dynamics of a system and therefore enhance program comprehension. A special attention will be paid to the usability of the tool as well.

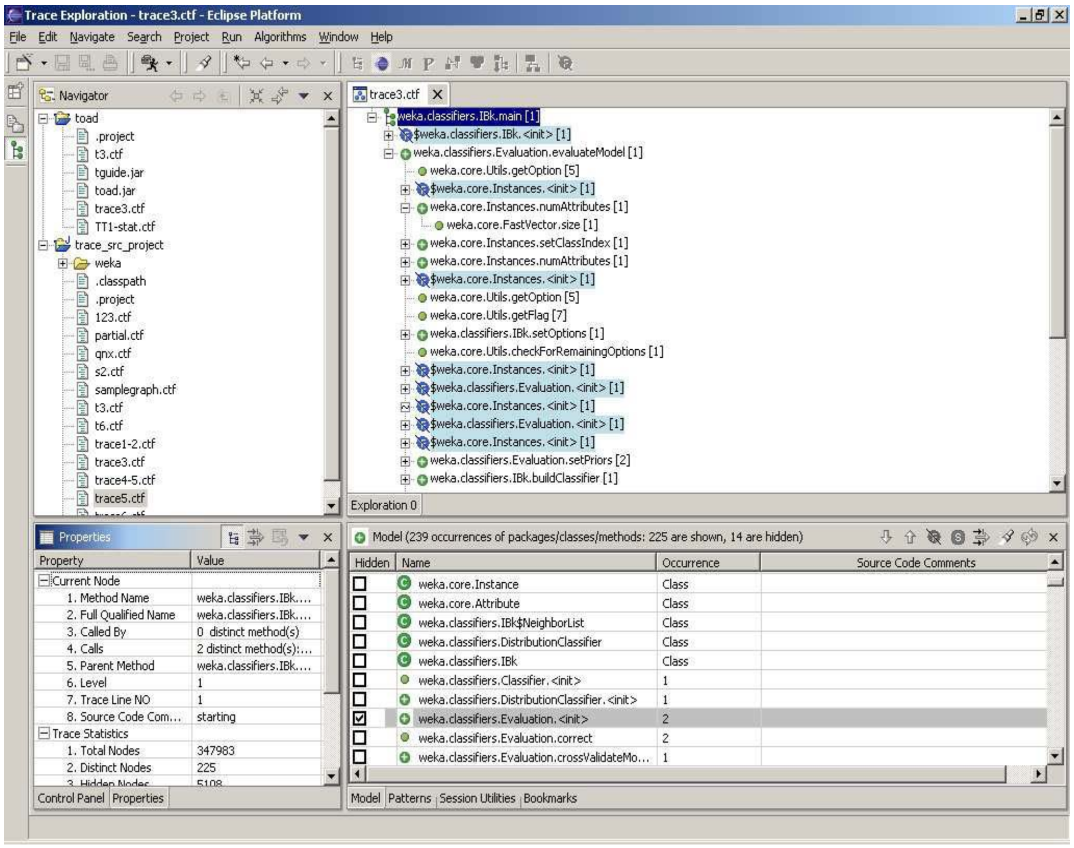
5. Conclusions and Future Directions

In this paper, we presented a new trace analysis tool called SEAT (Software Exploration and analysis Tool) and the various features it supports. Using SEAT, maintainers are provided with features that can help them:

1. Explore the trace and search for specific components
2. Browse the trace content using a tree-like widget
3. Map the trace content to the other system artifacts using Eclipse facilities
4. Filter the trace content by using several filtering techniques (e.g. pattern matching)
5. Detect patterns of execution using various matching criteria
6. Remove specific trace components and add them to a list of utilities that can be used during the processing of other traces generated from the same system
7. Exchange traces using the Compact Trace format (CTF)
8. Manipulate very large traces due to the design for scalability on which SEAT is based (e.g. representing traces in CTF, efficient tree loading algorithms, etc.)

The maintainer has also a considerable control over how the above techniques are applied. This is because the needs for trace abstraction and compression will vary from task to task and person to person. For example, for feature 4, there will be individual parameters to adjust to help the maintainer control what she or he wants to hide.

One direction for future work would be to investigate how the system could automatically or semi-automatically suggest appropriate settings for the parameters of features 4 and 5. Settings could be determined based on the nature of the trace, and the current goals and experience of the maintainer. Finally, there is also a need to investigate how utilities can be detected automatically (or semi-automatically) to prevent the maintainers from detecting them manually.


Trace Exploration - trace3.ctf - Eclipse Platform
File Edit Navigate Search Project Run Algorithms Window Help
Navigator
trace3.ctf
Exploration 0
Properties
Control Panel Properties
Model Patterns Session Utilities Bookmarks

```
toad
    .project
    t3.ctf
    tguide.jar
    toad.jar
    trace3.ctf
    TT1-stat.ctf
trace_src_project
    weka
    .classpath
    .project
    123.ctf
    partial.ctf
    qnx.ctf
    s2.ctf
    samplegraph.ctf
    t3.ctf
    t6.ctf
    trace1-2.ctf
    trace3.ctf
    trace4-5.ctf
    trace5.ctf
```

```
weka.classifiers.IBk.main [1]
    $weka.classifiers.IBk.<init> [1]
    weka.classifiers.Evaluation.evaluateModel [1]
        weka.core.Utils.getOption [5]
        $weka.core.Instances.<init> [1]
        weka.core.Instances.numAttributes [1]
            weka.core.FastVector.size [1]
        weka.core.Instances.setClassIndex [1]
        weka.core.Instances.numAttributes [1]
        $weka.core.Instances.<init> [1]
        weka.core.Utils.getOption [5]
        weka.core.Utils.getFlag [7]
        weka.classifiers.IBk.setOptions [1]
        weka.core.Utils.checkForRemainingOptions [1]
        $weka.core.Instances.<init> [1]
        $weka.classifiers.Evaluation.<init> [1]
        $weka.core.Instances.<init> [1]
        $weka.classifiers.Evaluation.<init> [1]
        $weka.core.Instances.<init> [1]
        weka.classifiers.Evaluation.setPriors [2]
        weka.classifiers.IBk.buildClassifier [1]
```

Property	Value
Current Node	
1. Method Name	weka.classifiers.IBk.…
2. Full Qualified Name	weka.classifiers.IBk.…
3. Called By	0 distinct method(s)
4. Calls	2 distinct method(s):…
5. Parent Method	weka.classifiers.IBk.…
6. Level	1
7. Trace Line NO	1
8. Source Code Com…	starting
Trace Statistics	
1. Total Nodes	347983
2. Distinct Nodes	225
3. Hidden Nodes	5108

Model (239 occurrences of packages/classes/methods: 225 are shown, 14 are hidden)

Hidden	Name	Occurrence	Source Code Comments
☐	weka.core.Instance	Class	
☐	weka.core.Attribute	Class	
☐	weka.classifiers.IBk$NeighborList	Class	
☐	weka.classifiers.DistributionClassifier	Class	
☐	weka.classifiers.IBk	Class	
☐	weka.classifiers.Classifier.<init>	1	
☐	weka.classifiers.DistributionClassifier.<init>	1	
☑	weka.classifiers.Evaluation.<init>	2	
☐	weka.classifiers.Evaluation.correct	2	
☐	weka.classifiers.Evaluation.crossValidateMo…	1	

Figure 3. A snapshot of SEAT GUI

References

[1]. 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, Berkeley, CA, 1998, pp. 219-234

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

[3]. A. Hamou-Lhadj, T. C. Lethbridge**,** "A Metamodel for Dynamic Information Generated from Object-Oriented Systems", *In Proc of the 1st International Workshop on Meta-models and Schemas for Reverse Engineering (ATEM), Victoria, Canada, Electronic Notes in Theoretical Computer Science (ENTCS),* 94: 2004, pp. 59-69

[4]. A. Hamou-Lhadj, T. C. Lethbridge, "A Survey of Trace Exploration Tools and Techniques", *In the Proc. of the 14th Annual IBM Centers for Advanced Studies Conferences (CASCON)*, Toronto, Canada, October 2004

[5]. A. Hamou-Lhadj and T. Lethbridge, "Techniques for Reducing the Complexity of Object-Oriented Execution Traces", *In Proc. of VISSOFT 2003, 2nd IEEE International Workshop on Visualizing Software for Understanding and Analysis* Amsterdam, The Netherlands, October 2003, pp. 35-40

[6]. A. Hamou-Lhadj, T. C. Lethbridge, "Measuring the Content of Traces to Better Characterize the Work Required for Program Comprehension", *In the Proc. of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), Shanghai, China,* June 2005