A Survey of Trace Exploration Tools and Techniques

Abdelwahab Hamou-Lhadj
University of Ottawa
SITE, 800 King Edward Avenue
Ottawa, Ontario, Canada K1N 6N5
ahamou@site.uottawa.ca

Timothy C. Lethbridge
University of Ottawa
SITE, 800 King Edward Avenue
Ottawa, Ontario, Canada K1N 6N5
tcl@site.uottawa.ca

Abstract

The analysis of large execution traces is almost impossible without efficient tool support. Lately, there has been an increase in the number of tools for analyzing traces generated from object-oriented systems. This interest has been driven by the fact that polymorphism and dynamic binding pose serious limitations to static analysis. However, most of the techniques supported by existing tools are found in the context of very specific visualization schemes, which makes them hard to reuse. It is also very common to have two different tools implement the same techniques using different terminology. This appears to result from the absence of a common framework for trace analysis approaches. This paper presents the state of the art in the area of trace analysis. We do this by analyzing the techniques that are supported by eight trace exploration tools. We also discuss their advantages and limitations and how they can be improved.

1 Introduction

Maintaining a poorly documented system is a difficult task. Reverse engineering tools can help with several maintenance activities such as source code exploration, data flow analysis, and design and architecture recovery. The common objective is to extract high-level views from low-level components to facilitate the comprehension of the system. Most of these tools rely on static analysis of the source code, since the documentation is often deemed to be out of date.

Static analysis has been shown to be successful for procedural software systems; however, polymorphism and dynamic binding make it difficult to fully understand object-oriented systems by merely performing static analysis [16]. For such systems, there is a need to apply dynamic analysis techniques.

Dynamic analysis consists of analyzing the *behavior* of a software system to extract its proprieties. Ball explains that run-time information has the advantages of being precise and being sensitive to the input data [1]. Indeed, dynamic analysis typically involves instrumenting the program under investigation. Unlike static analysis, where the analyst needs to go through the many different relationships of all the system artifacts, dynamic instrumentation can be tuned to collect only the information needed to perform the maintenance activity at hand. Also, system execution can be driven by specific input data which provides a powerful mechanism for relating program inputs, program outputs and program behavior. This can help software engineers understand the program by looking at how it changes according to different inputs and outputs.

The behavioral aspect of object oriented systems consists mainly of interactions between objects. However, important interactions are usually hard to extract because they are often mixed with low-level implementation details generating very large traces.

Research in this area has led to many tools to tackle this problem. In this paper, we present an analysis of eight tools. The goal is to uncover the underlying concepts behind these tools in an

attempt to build a common core of techniques that can be useful for understanding the behavior of object-oriented systems. To achieve this, we present the advantages and limitations of these tools. We also discuss how we think they can be improved, and outline key research questions that need to be addressed.

Pacione et al. [19] conducted a study in which they evaluated five tools based on how they enabled a number of program comprehension and reverse engineering tasks such as identifying the system architecture, extracting design patterns etc. Their conclusion is that none of the tools performs well for all tasks and some of the tasks are even beyond all of the tools. A key difference between Pacione's work and this paper is that the maintenance activities targeted are totally different.

There are several aspects of an execution trace that can be studied. In our study, we focus on the following points:

- How do the studied tools model large execution traces?
- What levels of abstraction can be achieved by the tools?
- What techniques are used to reduce the size of the traces?

This paper is organized as follows: first, we describe traces of object interactions in a more precise way and discuss the techniques that allow generating them. Next, we describe the tools chosen for this study. Finally, we present a detailed analysis of these tools by exhibiting their advantages, limitations and ways of improving them.

2 Traces of Object Interactions

Objects interact by sending messages, which can be depicted using a UML sequence diagram. Figure 1 shows an example of interactions among three objects of the classes C1, C2 and C3 respectively. Traces of object interactions are also referred to as traces of method invocations or simply traces of method calls.

To reproduce the execution of an object-oriented system, one needs to collect at least the events related to object construction and destruction and method entry and exit [4]. Additional information can be collected as well. For example, if the system is multi-threaded then events related to thread execution will need to be collected. A more practical way of representing traces of method calls is using a tree structure [6]. For example, the tree in Figure 2 corresponds exactly to the sequence diagram presented in Figure 1. Time flows from top to bottom and from left to right.

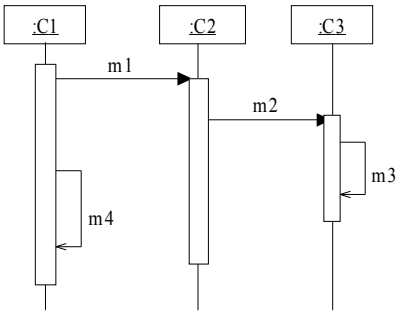

Figure 1: Object interaction depicted by a sequence diagram

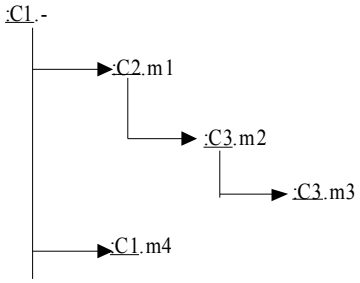

Figure 2: Tree representation of an object interaction trace

It is very common that traces, once generated, are saved in text files. A trace file usually contains a sequence of lines in which each line represents an event. An example of this representation is given by Richner and Ducasse in [20]. Each line records: The class of the sender, the identity of the sender, the class of the receiver, the identity of the receiver and the method invoked. The order of calls can be maintained in two ways, either each entry and exit of the method is recorded, which results is a very large trace file, or an integer is

added to represent the nesting level of the calls. In this case, we do not need to record the exit event of a method as shown by the following illustration.

Assuming that obj1 is a unique identifier of the object :C1 and that obj2 represents :C2 and obj3 represents :C3, the trace file that corresponds to the trace of Figure 1 should have the following events:

Sender Class	Sender ID	Receiver Class	Receiver ID	Called method	Nesting Level
C1	obj1	C2	obj2	m1	1
C2	obj2	C3	obj3	m2	2
C3	obj3	C3	obj3	m3	3
C1	obj1	C1	obj1	m4	1

There are different techniques for generating traces of method calls. The first technique is based on instrumenting the source code, which consists of inserting probes (e.g. a print statement) at different locations in the source code. In the context of object-oriented systems, probes are usually inserted at each entry and optionally each exit of every method. Instrumentation is usually done automatically.

Another technique for collecting run-time information consists of instrumenting the execution environment in which the system runs. For example, the Java Virtual Machine can be instrumented to generate events of interest. The advantage of this technique is that it does not require the modification of the source code.

Finally, it is also possible to run the system under the control of a debugger. In this case, breakpoints are set at locations of interest (e.g. entry and exit of a method). This technique has the advantage of not modifying the source code and the environment; however, it can slow down considerably the execution of the system.

3 Tools and Techniques

In this section, we present the tools that are chosen for this study. We selected this tools based on the numerous concepts they implement. This does not represent all the tools that exist in the literature. It is also important to mention that some of these tools are not openly available and their analysis is based on the scientific publications that describe them. We exclude from our analysis tools that deal with distributed systems for simplicity reasons.

3.1 Shimba

Systä presents a reverse engineering environment, called Shimba, which combines static and dynamic analysis to understand the behavior of Java software systems [21, 22, 23, 24]. Static analysis is used to select a set of components that need to be examined later during dynamic analysis. Systä's approach is based on the fact that a software engineer does not need to trace the whole system if only a specific part needs to be analyzed.

For this purpose, the system artifacts and their inter-dependencies are extracted from the Java class files and viewed using a reverse engineering tool called Rigi [17]. In Rigi, all the artifacts are shown as nodes and the dependencies are shown as directed edges between the nodes. Shimba considers the following system artifacts: classes, interfaces, methods, constructors, variables, and static initialization blocks. The dependencies among these artifacts include inheritance relationships, containment relationships (e.g. a class contains a method), call relationships and so on. Using Rigi, a software engineer can run a few scripts to exclude the nodes that are not of interest and keep only those she or he wants to investigate. Breakpoints are then set at events of interest (e.g. the entry of a method or a constructor) of the selected classes. The target system is executed under a customized debugger and the trace is collected.

The next step is to analyze the trace. For this purpose, a software engineering tool called SCED is used [15]. SCED permits representing execution traces in the form of scenario diagrams -scenario diagrams are similar to UML sequence diagrams. SCED has the ability to extract state machines given several scenario diagrams.

Although the execution trace represents only the classes that were selected using static analysis, Systä recognizes the fact that these traces may still be large. To overcome this problem, she applies the Boyer-Moore string matching algorithm to SCED scenario diagrams in order to detect repeated sequences of identical events that she refers to as *behavioral patterns.* She distinguishes between two kinds of behavioral patterns. The

first one is related to contiguous repetitions of sequences of events due to loops. These patterns are shown using a repetition construct that exists in SCED. The second type consists of behavioral patterns that occur in a non-contiguous way in the trace. They are represented using subscenarios constructs, which consists of boxes that are added to SCED scenario diagrams. A subscenario box encapsulates the events of an instance of the behavioral pattern. A user can double click on a subscenario box to display the detailed pattern information.

3.2 ISVis

ISVis is a visualization tool that supports analysis of execution traces of object-oriented systems [11, 12]. ISVis is based on the idea that large execution traces are made of recurring patterns, referred to as *interaction patterns*, and that visualizing these patterns is useful for reverse engineering. Interaction patterns are in fact the behavioral patterns as described in Shimba.

The execution trace is visualized using two kinds of diagrams: the information mural and a temporal message-flow diagram (a variant of UML sequence diagrams). The two diagrams are connected and presented on one view called the *scenario view* as shown in Figure 3. The information mural uses visualization techniques to create a miniature representation of the entire trace that can easily show repeated sequences of events. The temporal message-flow diagram is used to display the detailed content of the trace. The software engineer can spot a pattern on the information mural view, select it and investigate its content using the temporal message-flow diagram**.**

To deal with the size explosion problem, ISVis uses an algorithm that detects patterns of identical sequences of calls. Given a pattern, the user can search in the trace for an exact match, an interleaved match, a contained exact match (components in the trace that contain components in the pattern) and a contained interleaved match. Additionally, the user can use wildcards to formulate more general search queries.

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 interactions between this subsystem and the other components of the trace.

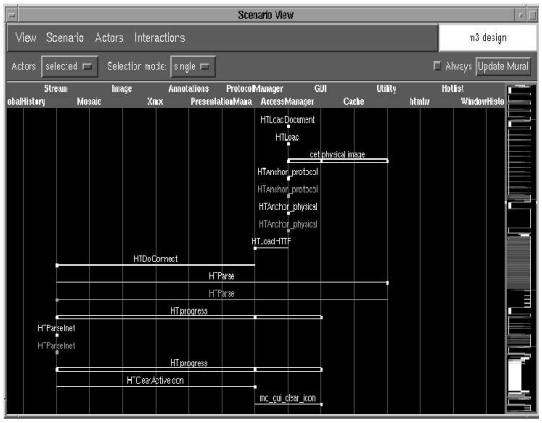

Figure 3: ISVis scenario view which consists of the information mural view (on the right) and the temporal message-flow diagram (center).

In [12], Jerding et al. describe a data structure for the internal representation of traces. This is based on the idea that a trace of method calls, which is a tree structure, can be transformed into its compact form, which results in a directed acyclic graph where the same subtrees are represented only once. This representation allows ISVis to scale up to very large traces.

3.3 Ovation

De Pauw et al. introduce a tool called Ovation [6]. Unlike Shimba and ISVis, Ovation is based on visualizing traces using the *execution pattern view* [6], which is not a form of UML sequence diagrams. According to the authors, the execution pattern view, which is a tree-like structure view as shown in Figure 4, is less cumbersome than UML sequence diagrams. This view lets the user browse the program execution at various levels of detail. For example the user can collapse and expand subtrees, show only messages sent to a particular object, remove contiguous repetitions of sequences of calls, zoom in and out the trace panel and many other useful visualization features.

To overcome the size explosion problem, similar sequences of events are shown as instances of the same pattern. The patterns are then color

coded to allow software engineers to notice them easily.

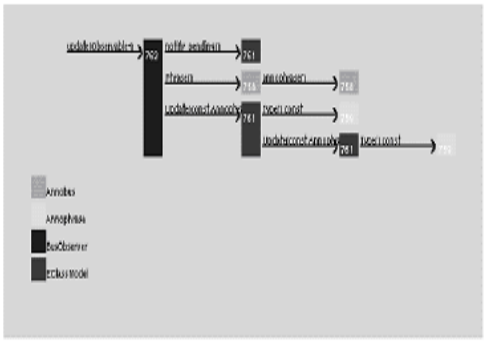

Figure 4: The Execution Pattern view of Ovation

However, the authors notice that exact match results in several patterns that do not reduce much of the size problem. For this purpose, they present a set of matching criteria that can be used to decide when two sequences of events can be considered equivalent [6]. We summarize the main criteria in what follow:

- Identity: Two sequences of calls are considered instances of the same pattern if they have the same topology: same order, objects, methods, and so on.
- Class Identity: If two sequences of calls involve the same classes but different objects then they can be considered similar according to this criterion.
- Depth-limiting: This criterion consists of comparing two sequences up to a certain depth only.
- Repetition: It is very common to have two different sequences of calls that differ only by the number of repetitions due to loops and recursion. If this number is ignored then these two sequences can be considered equivalent.
- Polymorphism: This criterion suggests considering two subclasses of the same base class as the same. However, this applies only if they invoke the same polymorphic operations.

Although these criteria seem to be interesting, the authors do not discuss whether they can be combined or not and if yes, how? The paper also lacks statistical data to support the use of these criteria. For example, it would be interesting to know the gain in terms of the number of patterns that results after applying a given matching criterion.

3.4 Jinsight

Jinsight is a Java visualization tool that shows the execution behavior of Java programs [3, 4]. Jinsight provides several views that can be very helpful for detecting performance problems. These views can be summarized in what follows:

- The Histogram View helps the analyst detect performance bottlenecks. It also shows object references, instantiation and garbage collection.
- The Execution View: This view displays the program execution sequence (Figure 5). It helps the analyst understand concurrent behavior, thread interactions and detect deadlocks.
- The Reference Pattern View: It is used to show the interconnections among objects. For this purpose, Jinsight implements pattern recognition algorithms to reduce the information overhead. In fact, this view is equivalent to the pattern execution view of Ovation introduced by the same authors and that was described in the previous subsection.
- The Call Tree View: shows the sequence of method calls, including the number of calls and their contribution to the total execution time as shown in Figure 6.

Figure 5: Jinsight Execution View

Jinsight is heavily tuned towards performance analysis rather than program comprehension. However, according to the authors, the reference pattern and the call tree views can be used for general understanding of the system execution.

Jinsight uses a model introduced by De Pauw et al. in [5] for representing the information about the execution of an object-oriented program. Interesting events are object construction/destruction and method invocation and return. They organized these artifacts in a four-dimensional event space having axes for classes, instances, methods and time as shown in Figure 7. Each point corresponds to an event during program execution. Information is extracted by traversing or projecting one or more dimensions of the space in different combinations to produce subspaces.

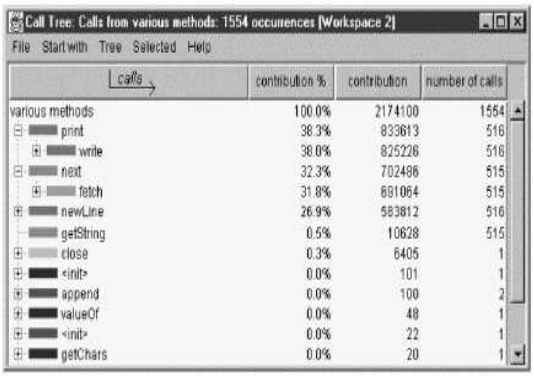

Figure 6: Jinsight Call Tree View

However, the event space of even a small system might be very large. To overcome this problem, the authors introduce the concept of *call frames*. A call frame is a combination of events that depicts a communication pattern between a set of objects. For example, consider a method m1 of class c1 that calls a method m2 of class c2. This sequence typically involves an object o1 of c1 and an object o2 of c2 (this does not apply if static methods are used). The whole sequence is saved as one call frame instead of saving every single event of this sequence.

Statistical information can also be computed at the same time the system executes. For example, we can associate to the previous call frame the number of times the method m1 calls m2 or the number of times the class c1 calls c2. For this purpose, the authors use several data structures to represent the call frames.

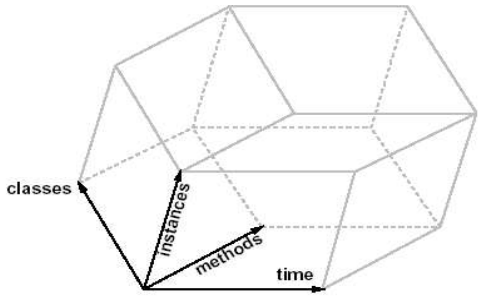

Figure 7: Four-dimensional event space used by Jinsight to represent trace events

Although this technique might result in a significant reduction of the number of events, it is more tuned to performance analysis than program comprehension. Indeed. Most of the visualization views presented in the authors of Jinsight, which are built on the top of this model, exhibit statistical information only and are similar in principle to the way profilers work.

3.5 Program Explorer

Program Explorer is a C++ exploration tool that focuses on analyzing interactions between objects and classes [16].The authors start by introducing a common model and notation for OO program execution. Interactions between objects are modeled using a directed graph called *the iteraction graph* (Figure 8). The nodes of the graph represent objects and the arcs represent method invocations. Arcs are labeled with the name of the method, the time at which the invocation of the method takes place and the time at which the execution returns to the caller.

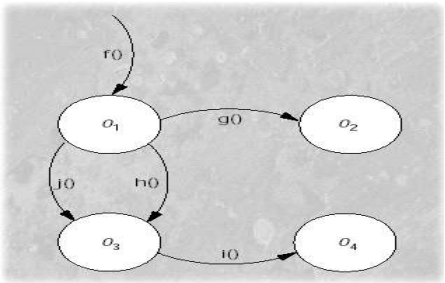

Figure 8: The interaction graph is used by Program Explorer to represent object interactions

To overcome the size explosion problem, Program Explorer uses several filtering techniques, which are:

Merging: Using Program Explorer, the analyst can merge arcs that represent identical methods between pairs of objects. The resulting graph is called the Object Graph and emphasizes how object interacts but hides the order and the multiplicity of invocations. Furthermore, the analyst can merge objects of the same classes into one node to reduce the number of nodes. The resulting graph is called the Class Graph and focuses on class interaction rather than object interaction. An example of the class graph is shown in Figure 9. This is similar in principle to the pattern matching criteria that are supported by Ovation and Jinsight.

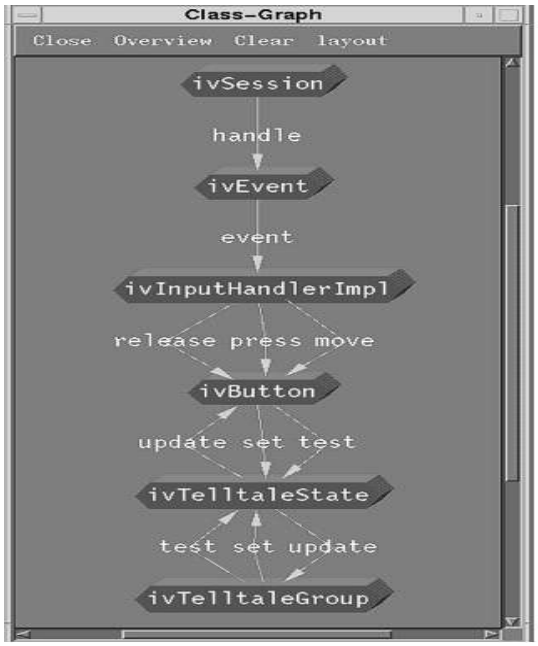

Figure 9: An example of the class graph. A class graph focuses on class interactions rather than object interactions

Pruning: Pruning is the process of removing information from the interaction graph in order to reduce its size. Program Explorer implements three kinds of pruning: object pruning, method pruning and class pruning. Pruning an object consists of removing its corresponding node from the interaction graph. The incoming and outgoing arcs of this node are also removed. Method pruning consists of performing the same task on specific methods. The subsequent invocations that derive from them are also removed. Pruning can also apply to inheritance hierarchies and is called class pruning. Class pruning consists of the fact that pruning a superclass method will result in pruning this method at the subclasses level. The several pruning techniques are exactly similar to the many different browsing capabilities that exist in ISVis, Ovation and Jinsight.

Slicing: Object slicing is similar to dynamic slicing and aims at keeping all the activation paths in which the object participates. That is, all the other paths are removed from the graph. Method slicing accomplishes the same task as object slicing except that it focuses on keeping specific methods of an object.

3.6 AVID

Walker et al. describe a tool, called AVID (Architecture Visualization of Dynamics in Java Systems), for visualizing dynamic behavior at the architectural level [26]. AVID uses run-time information and a user-defined architecture of the system to create a dynamic view of the system components and the way they interact.

First, the analyst creates a trace about the calls between methods and about the instantiation and destruction of objects. Next, she or he needs to cluster classes into components called entities. In AVID, clusters are represented as boxes and the dynamic relationships extracted from the trace as directed arcs as shown in Figure 10. An arc between two entities A and B is labeled with the number of calls the methods of the classes in A make to the methods of the classes in B. Instantiation and destruction of objects are shown as bar-chart style of histograms associated with each box.

In AVID, the analyst can control the sequence of events she or he wants to visualize. This is done by breaking the execution trace into a sequence of views called *cels.* Animation techniques allow the analyst to show the whole execution cel by cel, which is also called the play mode, stop the animation, as well as go forward and backward. These techniques aim at reducing the information overhead when dealing with large execution

traces. Furthermore, AVID contains a summary view in which all the interactions are shown.

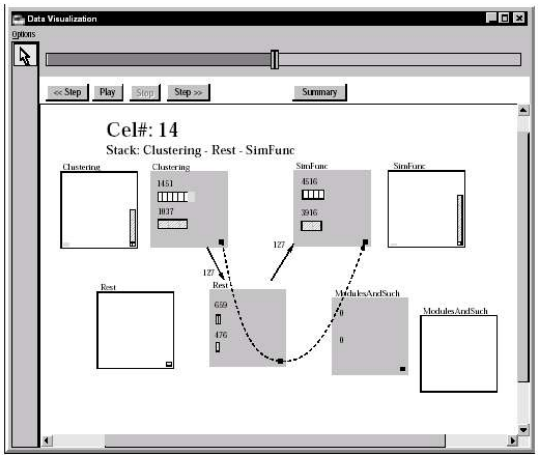

Figure 10: Interactions among the system clusters as represented by AVID. Here, the analyst has replayed the execution and stopped at Cel#14

Although animation techniques can help reduce the information overhead, traces are very large and there is a need to investigate more techniques to reduce their size. In [2], AVID was improved to consider compression techniques based on sampling. The authors describe a set of sampling parameters that can be used by the analyst to consider only a sample of the execution trace. For example, the analyst can choose the events that appear after a certain timestamp only or a snapshot of the call stack every x^{th} event and so on.

However, there is a lack of scientific evidence regarding which parameters are best to use. If they work for a given scenario they may not work for another one as shown by the results of the experiments conducted by the authors of AVID in which some parameters worked for one case study but did not work for the other case study.

3.7 Scene

Koskimies and Mössenböck present a tool called Scene (Scenario Environment) that is used to produce scenario diagrams from a dynamic event trace [13, 14]. The authors notice that horizontal scrolling makes the diagrams cumbersome and there is a need for techniques that center the information conveyed by scenario diagrams on the screen. They name this problem *the focusing problem* and suggest several visualization-oriented techniques to solve it. Among these techniques, we have:

Call Compression: This technique consists of collapsing the internal calls that derive from a given call. A click on this call will result in making its internal calls visible. Figure 11 shows an example of three calls that have been collapsed. The user can click on these calls to view the internal calls they generate.

Partitioning: It is very common that the internal calls of a given call may not fit on the screen, therefore, Scene divides these calls into parts. A click on a part will show only the calls that it encapsulates.

Projection and removal: This operation allows the user to select an object and show only the interactions that involve its methods. The other interactions are then hidden.

Single-step mode: The single-step mode allows the user to display the internal calls of a given call one step at a time by clicking on the last visualized call.

Scene provides also a call summary view which consists of a call matrix that shows how the classes interact between each other.

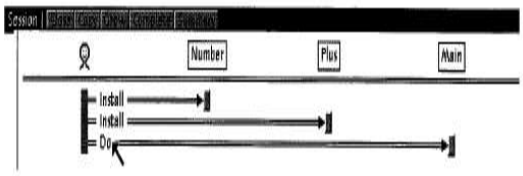

Figure 11: The calls Install and Do have been collapsed. The user can click on these calls to see the methods they invoke

3.8 The Collaboration Browser

Richner and Ducasse describe a tool called Collaboration Browser that is used to recover object collaborations from execution traces [20].

The authors define a collaboration instance as a sequence of method invocations that starts from a given method and terminates at its return point. This includes a single method call that does not generate other calls. Similar collaboration instances define a *collaboration pattern* (which is similar to Shimba behavioral patterns and ISVis

interaction patterns). Similarity is measured according to three kinds of matching criteria:

Criteria based on information about the event: An event in the trace contains information about the sender, receiver and the invoked method. The analyst can choose to include or omit any of these attributes in the matching process. For example, the analyst may decide to ignore the invoked method and match two sequences of calls using the sender and receiver classes (or objects) only. These represent an extension to the criteria that are supported by Ovation.

Excluding events: This category allows the analyst to exclude specific events in the matching scheme. For example, the analyst may decide to ignore events in which an object sends a message to itself or events that appear after a certain depth in the trace and so on.

Structure of the collaboration instance: A collaboration instance is a tree of events. The authors notice that similar collaboration instances may differ in their structure and still represent the same behavior. Therefore, one can consider two collaboration instances as instances of the same collaboration pattern if they contain the same set of events no matter in which order they occur or their nesting relationships.

Once the classes that constitute a given collaboration are determined, the user can query the trace or the collaboration pattern to extract the role of each of its classes, which corresponds to their public methods. In addition to that, the tool enables the developer to filter out dynamic information by removing classes or methods that are not of interest. The tool can also display an instance of a collaboration pattern as a UML sequence diagram.

The authors conducted an experiment with a framework for the creation of graphical editors called HotDraw. They were interested in understanding the implementation of one aspect of this framework, which is concerned with the tools that are responsible for creating and manipulating figures. First, they instrumented all the methods of the system. Next, they run a short scenario that involves the feature under analysis. The resulting trace contains 53735 method invocations.

To extract collaboration patterns, the authors, arbitrarily, picked several matching criteria. For example, they decided to ignore self-invocations, limit the depth of invocation to 20 and not consider the tree structure of collaboration instances during the matching process. 183 patterns were generated.

The next step is to query the patterns to extract only the collaboration patterns that describe the implementation of the feature under analysis. This process is iterative and assumes that the analyst has knowledge of the system so to know what to look for.

4 Discussion

The following discussion summarizes the main concepts that are implemented by these tools. We also discuss the advantages and limitations of these concepts and suggest possible improvements. This discussion is based on three main criteria, which aim at answering the three questions mentioned in the introduction section. We list these criteria in what follows and elaborate on them in more details in the next subsections:

- Criteria with respect to modeling the execution traces
- Criteria with respect to the level of abstraction achieved by these techniques
- Criteria with respect to reducing the size of the traces

4.1 Modeling Execution Traces

In order to visualize and analyze large program executions, an efficient representation of the event space is needed. Unfortunately, most of the tools mentioned above do not even discuss this aspect, which makes us have doubts regarding their scalability. It is also important to note that most of the experiments that are conducted by the authors of these tools are based on very small execution traces.

Among the tools that do discuss modeling issues, ISVis seems to implement the most interesting approach. ISVis uses a graph-theory concept that consists of transforming a rooted, labeled tree into a directed acyclic graph by representing iden-

tical subtrees only once. This concept is also known as the common subexpression problem and was first introduced by Downey et al. in [7].

In our recent work [9], we experimented with the idea of transforming the tree into an acyclic graph and found that this technique can reach a very high compression ratio. Although, we need to experiment with several other traces, we believe that this technique can help built very scalable tools.

Another interesting approach for modeling large execution traces is implemented in Jinsight. As we showed earlier, Jinsight uses the call frame principle to represent cumulative information about the traces such as the number of calls a method A makes to B and so on. However, the approach supported by Jinsight is more useful for performance analysis than program comprehension.

Finally, Trace Explorer uses a graph to represent the execution traces, where the nodes represent the objects and the arcs represents the method calls. However, this technique requires extra data structures to keep track of the order of calls, which makes traversing the graph time consuming.

In addition to this, the lack of a common modeling technique hinders interoperability between these tools. There is a need to work towards a common format for exchanging traces of object interactions. A common exchange format can also help researchers to use different tools on the same input in order to compare the techniques.

In our recent study, we presented the Compact Trace Format (CTF), which is a schema for exchanging traces of object interactions [10]. CTF is built with the idea of scalability in mind. Although, this work is still in progress, we believe that it is a step forward towards building a common exchange format for object-oriented dynamic information.

4.2 Levels of Abstraction

A key aspect of reverse engineering is to extract different levels of abstraction of a software system. There seems to be an agreement about the levels of abstraction that are needed for understanding the system functionality based on analyzing its execution traces. Using these tools, one can view the content of an execution trace at some of the following levels of abstraction:

Statement level: this level includes the execution of every single statement of the code. Most of the tools do not offer this view except debuggers. This level of abstraction suits best specific maintenance activities such as fixing bugs.

Object level: this level is concerned with visualizing method interactions among objects. This level can be useful for detecting memory leaks and other performance bottlenecks. Most of the tools support this level.

Class level: in this level, objects of the same class are substituted with the name of their classes. This level, that is supported by most of the tools, suits best activities that require high-level understanding of the system behavior such as recovering the documentation, understanding which classes implement a particular feature etc.

Architectural level: this level consists of grouping classes into clusters and showing how the system components interact with each other. AVID seems to be the tool that focuses most on analyzing this kind of interactions. For example, with AVID, one can display how many calls a given subsystem make to another subsystem etc.

Inter-thread communication level: although this paper does not focus on multi-threaded systems, it is important to notice that none of these tools except Jinsight implement the capability of viewing the interactions between the system threads. On the other hand, the tools that do usually do not allow viewing the system at the other levels of detail.

4.3 The Size Explosion Problem

A key element for a successful dynamic analysis tool consists of implementing efficient techniques for reducing the amount of information that exist in the traces. We classify the techniques used by these tools into two categories. The first category is concerned with the ability to browse the content of the trace easily, search for specific elements and so on. We call this category: *Trace Exploration*. The second category is concerned with the ability to reduce the size of the trace by removing

(or hiding) some of its components. We call this category of techniques: *Trace Compression.*

Trace exploration techniques are tightly coupled with the visualization tools that implement them. Generally speaking, using these techniques an analyst can browse, animate, slice or search the traces. It seems that there is an agreement about the importance of such techniques in reducing the information overhead and most of the tools seem to support these features. Trace Exploration is also concerned with techniques that allow searching the trace content for specific components. ISVis, for example, allows the user to perform the search using wildcards to formulate more general queries.

Trace compression techniques operate on the execution traces independently from any visualization scheme. The goal is to hide some elements from the trace in order to compress it – by compression; we mean reducing its size. For this purpose, we found that the tools implement different techniques that we present in what follows. We also discuss their advantages and limitations.

Data Collection Techniques: The collection of trace data can be done either at the system level or at the level of selected components. These two approaches have their advantages and disadvantages. The advantage of the system-level data collection approach is that the analyst does not need to know which components implement the feature under study. However, the resulting execution traces are usually very large and need to be preprocessed. The component-level data collection technique has the obvious advantage of resulting in smaller execution traces but requires from the analyst to know which components need to be instrumented. Shimba, for example, involves the analyst for detecting the components that implement the desired feature. We do not think that this is very practical in many situations. For complex features, this may take a long time. There is a need for feature localization techniques such as the ones described by Wilde et al. [27] and Eisenbarth et al. [8].

Pattern Matching: Most of the tools use pattern detection abilities to group similar sequences of events in the form of execution patterns. Execution patterns have been named differently including behavioral patterns in Shimba, interaction patterns in ISVis, collaboration patterns in The Collaboration Browser. Patterns are efficient at reducing the size of traces if they are generalized [6]. For this purpose, different matching criteria are used such as the ones implemented in Ovation and The Collaboration Browser. Some matching criteria require the setting of some parameters. For example, the depth-limiting criterion presented in Section 3.3. involves setting the depth at which two sequences of events need to be compared. The challenge is to find the appropriate settings for understanding the feature under study. Furthermore, the different combinations of the matching criteria will result in different compressions of the trace. There is a need to analyze which combinations best suit the comprehension process. Another serious limitation of most of the studies presented above is that they do not give any statistics regarding the compression gain attained. They also did not experiment with many software systems.

Sampling: Sampling is an interesting way of reducing the size of the trace and was used in AVID. It is concerned with choosing only a sample of the trace for analysis instead of the whole trace. However, finding the right sampling parameters is not an easy task and even if some parameters work for understanding one feature, it is not evident that they work for another feature.

Hiding Components: Another way for reducing the trace size is to hide some of its components. For example, the analyst may decide to hide all the invocations of a specific method. Most of the tools implement capabilities for removing information from the trace. The Collaboration Explorer and Program Explorer, for example, allow the analyst to remove methods, specific objects or even classes from the trace. Pruning and slicing are two concepts used in Program Explorer that achieve this. However, it is totally up to the maintainer to decide which components to hide.

In a recent study conducted by Zayour and Lethbridge [28] involving a large real world procedural telecommunication system, the authors 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 affect much the comprehension process. We think that this could be applied to object-oriented systems as well. In object oriented systems, utility methods (by analogy with

procedures) are very frequent due to encapsulation. For example accessing methods are a very good example of such methods. In addition to this, methods can be removed not only because they are utilities. For example, an abstract method usually corresponds to an abstract operation that is implemented in different ways. Since we seek abstraction, it may improve comprehensibility to hide the calls made by the various polymorphic implementations of an abstract operation. Future research should focus on finding heuristics that can be used to hide automatically elements from a trace without affecting its content.

Architectural-level filtering: Another approach for reducing the size of the traces is to show the dynamic behavior between the architectural components of the system rather than between single objects. For this purpose, the analyst first determines the system architecture (if it is not available) and then the execution trace is abstracted out to show the interaction between these components. However, this approach can help software engineers understand the system at the architectural level only. In addition to this, it requires the system architecture to be present. AVID, for example, assumes that the analyst is familiar enough with the system to cluster classes into components. However, this is not always true in practice. There is a need for automatic clustering techniques such as the ones described by Müller et al. in [18] and Tzerpos et al. in [25].

Conclusions and Future Directions

In this paper, we studied the concepts implemented in eight reverse engineering tools for analyzing traces of object interactions. The goal is to work towards building a common core of techniques for the efficient analysis of the behavior of OO systems. We found that these tools use a very rich set of techniques that are worth integrating into one common framework. Using these techniques, a maintainer is provided with features that enable them to:

1. Explore the trace and search for specific components
2. View the trace content at different levels of abstraction (e.g. object interactions, class interaction, etc…)
3. Filter the trace content by using several techniques (e.g. pattern matching, sampling etc.)

In all but the first of these, any implementing tool would need to allow the maintainer considerable control over how the technique is 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 3, there will be individual parameters to adjust, e.g. so the maintainer can 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 2 and 3. Settings could be determined based on the nature of the trace, and the current goals and experience of the maintainer. Machine learning could be employed to help tune the settings by learning over time from the adjustments maintainers make.

There is also a need for fundamental research in several areas: For example, we need to investigate what can be removed from the trace without affecting its content. We also need to experiment with a much larger variety of software systems to understand how to combine the compression techniques in order to extract the most important interactions. For this purpose, we might find it useful to start by assessing the gain in terms of the amount information that remains after removing unnecessary data.

There is also a need for a common metamodel for representing the execution traces of object-oriented systems in order to permit interoperability among tools. The reader might find the Compact Trace Format (CTF) [10] useful for this purpose. We are actually in the process of testing this format with large execution traces.

Finally, the suite of techniques described in this paper needs to be integrated with appropriate visualization techniques.

About the Authors

Abdelwahab Hamou-Lhadj, also known as Wahab, is a Ph.D. candidate at the School of Information Technology and Engineering at the University of Ottawa. His PhD thesis focuses on techniques for the understanding of large object-oriented systems.

Timothy C. Lethbridge is an Associate Professor of software engineering at the University of Ottawa. He studies techniques to help people better understand complex information. He is also author of the textbook Object-Oriented Software Engineering: Practical Software Development Using UML and Java.

References

[1] T. Ball. The Concept of Dynamic Analysis. *In Proc. of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 216-234, Toulouse, France, 1999

[2] A. Chan, R. Holmes, G. C. Murphy, A. T. Ying. Scaling an Object-Oriented System Execution Visualizer through Sampling. *In Proc. of the 11th International Workshop on Program Comprehension,* pages 237-244, Portland, Oregon, USA, 2003

[3] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, J. Yang,. Visualizing the Execution of Java Programs. *In Proc. International Seminar on Software Visualization*, pages 151-162, Dagstuhl Castle, Wadern, 2002

[4] W. De Pauw, R. Helm, D. Kimelman, J. Vlissides. Visualizing the Behaviour of Object-Oriented Systems. *In Proc. of the 8th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 326-337, Washington, DC, 1993

[5] W. De Pauw, D. Kimelman, J. Vlissides. Modelling Object-Oriented Program Execution. *In Proc. of the 8th European Conference on Object-Oriented Programming (ECOOP)*, pages 163-182, Bologna, Lecture Notes in Computer Science 821, Berlin, 1994

[6] 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)*, pages 219-234, Santa Fe, NM, 1998

[7] J. P. Downey, R. Sethi, R. E. Tarjan. Variations on the Common Subexpression Problem. Journal of the ACM. 27(4), pages 758-771, 1980

[8] T. Eisenbarth, R. Koschke, D. Simon. Feature-Driven Program Understanding Using Concept Analysis of Execution Traces. *In Proc. of the 9th International Workshop on Program Comprehension*, pages 300-309, Toronto, Ontario, Canada, 2001

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

[10] A. Hamou-Lhadj, and T. Lethbridge. A Metamodel for Dynamic Information Generated from Object-Oriented Systems. *1st International Workshop on Meta-models and Schemas for Reverse Engineering (ATEM)*, ENTCS, pages 59-69, Victoria, Canada, 2003

[11] D. Jerding., S. Rugaber. Using Visualisation for Architecture Localization and Extraction. *In Proc. Of the 4th Working Conference on Reverse Engineering*, pages 56-65, Amsterdam, Netherlands, 1997

[12] D. Jerding, J. Stasko, T. Ball. Visualising Interactions in Program Executions. *In Proc. of 19th the International Conference on Software Engineering*, pages 360-370, Boston, USA, 1997

[13] K. Koskimies, H. Mössenböck. Scenario-based browsing of object-oriented systems with Scene. *Report 4, Department of System Software, University of Linz*, August 1995.

[14] K. Koskimies, H. Mössenböck. Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs. *In Proc. of the 18th International Conference on Software Engineering (ICSE)*, pages 366-375, Berlin, Germany, 1996

[15] K. Koskimies, T. Männistö, T. Systä, J. Tuomi. SCED: A Tool for Dynamic Modeling of Object Systems. *University of Tampere, Dept. of Computer Science, Report A-1996-4*, 199

[16] D. B. Lange, Y. Nakamura. Object-Oriented Program Tracing and Visualization. *IEEE Computer, 30(5)*, pages 63-70, 1997

[17] H. A. Müller, K. Klashinsky. Rigi – A System for Programming In-the-Large. *In Proc. of the 10th International Conference on Software Engineering (ICSE)*, pages 80-86, Singapore, 1988

[18] H. A. Müller, M. A. Orgun, S. R. Tilley, J. S. Uhl. A Reverse Engineering Approach to Subsystem Structure Identification. Journal of Software Maintenance: Research and Practice, 5(4): pages 181-204, 1993

[19] M. J. Pacione, M. Roper, M. Wood. A Comparative Evaluation of Dynamic Visulation Tools. *In Proc. Of the 10th Working Conference on Reverse Engineering (WCRE)*, pages 80-89, Victoria, BC, Canada,, 2003

[20] T. Richner, S. Ducasse. Using Dynamic Information for the Iterative Recovery of Collaborations and Roles. *In Proc. of the 18th International Conference on Software Maintenance (ICSM)*, pages 34-43, Montréal, QC, 2002

[21] T. Systä. Understanding the Behaviour of Java Programs. *In Proc. of the 7th Working Conference on Reverse Engineering (WCRE)*, pages 214-223, Brisbane, QL, 2000

[22] T. Systä. Incremental Construction of Dynamic Models for Object-Oriented Software Systems. *Journal of Object-Oriented Programming, 13 (5)*, pages 18-27, 2000

[23] T. Systä, K. Koskimies, H. A. Müller. Shimba – An Environment for Reverse Engineering Java Software Systems. *Software–Practice and Experience, 31(4)*, pages 371-394, 2001

[24] T. Systä. Dynamic Reverse Engineering of Java Software. *In Proc. of 13th European Conference on Object-Oriented Programming (ECOOP), 3rd Workshop on Experiences in Object-Oriented Reengineering*, Lisbon, 1999

[25] V. Tzerpos, R. C. Holt. ACDC: An Algorithm for Comprehension-Driven Clustering. *In Proc. Of 7th the Working Conference on Reverse Engineering*, pages 258-267, Brisbane, Australia, 2000

[26] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Swanson, J. Isaak. Visualizing Dynamic Software System Information through High-level Models. *In Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 271-283, British Columbia, Canada, 1998

[27] N. Wilde, M. Scully. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance: Research and Practice, Vol. 7, No. 1*, 1995

[28] I. Zayour, T. C. Lethbridge. A Cognitive and User Centric Based Approach For Reverse Engineering Tool Design. *CASCON*, pages 16-30, Toronto, Canada, 2000