Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System

Abdelwahab Hamou-Lhadj and Timothy Lethbridge

SITE, University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, Canada
{ahamou, tcl}@site.uottawa.ca

Abstract

In this paper, we present a semi-automatic approach for summarizing the content of large execution traces. Similar to text summarization, where abstracts can be extracted from large documents, the aim of trace summarization is to take an execution trace as input and return a summary of its main content as output. The resulting summary can then be converted into a UML sequence diagram and used by software engineers to understand the main behavioural aspects of the system. Our approach to trace summarization is based on the removal of implementation details such as utilities from execution traces. To achieve our goal, we have developed a metric based on fan-in and fan-out to rank the system components according to whether they implement key system concepts or they are mere implementation details. We applied our approach to a trace generated from an object-oriented system called Weka that initially contains 97413 method calls. We succeeded to extract a summary from this trace that contains 453 calls. According to the developers of the Weka system, the resulting summary is an adequate high-level representation of the main interactions of the traced scenario.

Keywords: Reverse engineering; Dynamic analysis; Design recovery; Program comprehension

1. Introduction

Since the outset of our research, our goal has been to find ways to make it easier for software engineers to understand a program's behaviour by exploring traces. This necessitates simplifying views of traces in various ways – in other words, reducing their size and complexity while keeping as much of their *essence* as possible.

Most existing trace analysis tools such as the ones presented in [3, 4, 6, 8, 15, 19, 20, 23] rely a set of fine-grained operations that software engineers can use to go from a raw sequence of events to a more understandable trace content. But due to the size and complexity of typical traces, this bottom-up approach can be difficult to perform and requires a considerable intervention of the users.

However, when asked to describe their experience with using traces, many software engineers of QNX Software Systems, the company that supports part of this research, argued that they almost always want to be able to perform top-down analysis of a trace by having the ability to look at the *big picture* first and then dig into the *details* [10]. Many research studies in program comprehension have shown that an adequate understanding of the system artefacts necessitates both strategies (i.e. bottom-up and top-down) [16].

In this paper, we present the concept of trace summarization and define it as the process of taking a trace as input and returning a summary of its main content as output. Most of the steps involved in this process are performed automatically. However, we anticipate that software engineers will need to further manipulate the resulting summary by adjusting its content to their specific needs. The result of this step will depend on the knowledge they have of the functionality under study, the nature of the function being traced, and the type of problem the trace is being used to solve (debugging, understanding, etc.).

This paper is based on previous work (see [13]), where we presented a technique for recovering high-level behavioural design models from traces. The new contributions of this paper consist of:

- The concept of trace summaries.
- An improved metric for measuring the extent to which a component can be considered a utility.
- A trace summarization algorithm
- A case study to validate the summarization technique.

The traces we focus on in this paper are the ones based on routine calls. We use the term 'routine' to refer to any routine, function, or procedure whether it is a method of a class or not.

The remainder of this paper is organized as follows: In the next section, we define what we mean by a trace summary. In Section 3, we discuss the concepts of utilities and implementation details. A trace summarization

algorithm is presented in Section 4. In Section 5, we present a case study where we evaluate the effectiveness of our approach. In Section 6, we discuss related work.

2. What is a Trace Summary?

In general, a summary represents the main points of a document while removing the details. The term ‘summary’, as used here is effectively synonymous with the term ‘abstract’, discussed below in the context of text summarization.

Jones defines a summary of a text as “a derivative of a source text condensed by selection and/or generalization on important content” [17]. Similarly, we define a summary of a trace as an abstract representation of the trace that results from removing unnecessary details by both selection and generalization.

The study conducted in QNX, which is described in more detail in [10], shows that when trying to understand the content of large traces, most software engineers will likely want to hide low-level implementation details such as utilities. Our approach for summarizing the main content of traces is therefore based on the successive filtering of traces by removing such low-level implementation details. In other words, the trace content selected to be part of the summary consist of the routines that implement high-level domain concepts.

In the next section, we present a metric based on *fan-in* and *fan-out* that will help us detect the utility components of a poorly documented system. However, we draw a distinction by considering utilities to be a subset of the concept of implementation details. A definition of what we mean by an implementation detail is presented in the end of the next section. In Section 4, we will present a trace summarization algorithm that uses the detection of utilities as its main mechanism.

Content generalization, an alternative approach to content selection, consists of generalization of specific content; i.e. replacing it with more general abstract information [17]. When applied to execution traces, generalization can be performed in two ways:

The first approach to generalization involves assigning a high-level description to selected sequences of events. For example, many trace analysis tools provide the users with the ability to manually select a sequence of calls and replace it with a description expressed in a natural language. However, this approach relies on user input so would not be practical to automate.

A second approach to generalization relies on treating similar sequences of events as if they were the same. This approach can be automated by varying a *similarity function* used to compare sequences of calls. Other possibilities include treating all subtrees that differ by only a certain *edit distance* as the same.

In the remainder of this paper, we will focus on content selection, leaving content generalization for consideration as another line of research.

3. The Concept of Utilities and Implementation Details

We define a utility as: *Any element of a program designed for the convenience of the designer and implementer and intended to be accessed from multiple places within a certain scope of the program.* The rationale behind this definition is as follows: the more calls a method has from different places (i.e. the more incoming edges in the static call graph), then the more purposes it likely has, and hence the more likely it is to be a utility. Conversely, we would expect a utility routine to be relatively self-contained (i.e. to have low coupling and high cohesion); if a routine has many calls (outgoing edges in the static call graph), this is evidence that it is less likely to be considered a utility. Also, routines that make many calls may be more needed in a trace summary to understand the system.

In [13], we presented a metric for measuring the extent to which a particular routine can be considered a utility that is based on the fan-in metric. In this paper, we improve this metric by considering fan-out as well as fan-in. We therefore suggest the following utilityhood metric:

Given a routine *r* and the following:

- N = The number of routines in the routine call graph
- Fanin(r) = The number of routines in the graph, other than r, that call r.
- Fanout(r) = The number of routines in the graph, other than r, that r calls.

We define the utilityhood metric of the routine *r*, U(r), as:

$$U(r) = \frac{Fanin(r)}{N} \times \frac{Log\left(\frac{N}{Fanout(r)+1}\right)}{Log(N)}$$

U(r) has 0 (not a utility) as its minimum and approaches 1 (most likely to be a utility) as its maximum.

Explanation:

First, we want to note that Fanin(r) and Fanout(r) both vary from 0 to |N|-1 (i.e. self dependencies are ignored).

This formula can be split into two parts. The first part $\frac{Fanin(r)}{N}$ simply reflects the fact that the routines with large fan-in are the ones that are most likely to be utilities.

For example, if the routine is called from all other routines of the system then its $\frac{Fanin(r)}{N}$ will be close to 1 (it will never reach 1 since self dependencies are ignored, i.e., Fanin(r) < N).

However, as discussed earlier, it is also important to consider the number of routines that are called by a particular routine. Therefore, we multiply the first part (i.e. $\frac{Fanin(r)}{N}$) by a coefficient that takes into account fan-out, although with lower emphasis. We achieve this using $Log(\frac{N}{Fanout(r)+1})$. We use $Fanout(r)+1$ for convenience to avoid situations where Fanout = 0.

If a routine *r* does not call any other routine of the system then Fanout (r) = 0, hence $Log(\frac{N}{Fanout(r)+1}) = Log(N)$, which would be dependent on the size of the system under study. We divide this result by Log(N) to ensure that both this expression and the entire formula vary from 0 to 1. In the case of *r,* its utilityhood metric will be equal to $\frac{Fanin(r)}{N}$.

On the other hand, a routine that has very large fan-out will result in $Log(\frac{N}{Fanout(r)+1})$ that tends to zero cancelling the effect of fan-in. This is a routine that should not be considered as a utility.

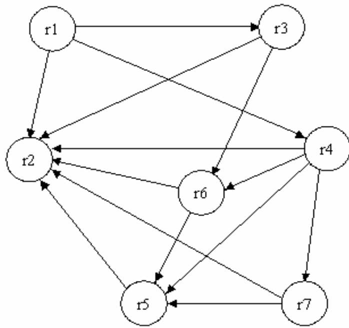

Figure 1. Example of a routine call graph

The result of applying the utilityhood metric to the static call graph of Figure 1 is shown in Table 1 (in this table, the base of the logarithm is 2). This table is sorted according to the descending order of U. In this example, we can see that the routine r2 is a candidate utility routine since it is called by all other routines and does not call any other routine (its U value is the highest).

Table 1. Utilityhood metric for the routine of Figure 1.

Routines	Fanin	Fanout	U
r2	6	0	0.86
r5	3	1	0.27
r6	2	2	0.12
r3	1	2	0.06
r7	1	2	0.06
r4	1	4	0.02
r1	0	3	0.00

However, it is important to use a static call graph rather than a dynamic call graph generated from the trace itself (or a set of traces). The rationale is that the if we were to use the dynamic call graph, we may find many cases where a routine by chance has only one call to it (or just a few) and hence would not be considered a utility merely because the scenario that generated the graph did not result in calls from any other places.

A tool using the utilityhood metric would need to select a threshold above which to consider routines as utilities, and therefore to suppress them from the trace. The exact threshold will vary depending on the context; for example, if there is a strong need to compact the trace further, then a higher threshold can be picked. Alternatively, if the trace has already been compacted too far, and the software engineer finds that he or she would like to see more detail in order to understand it, then the threshold can be reduced.

We want to note that according to the above definition of utilities, not all implementation details will be considered utilities. Many routines that implement details in algorithms might be designed to be called from one specific place. Later on, when we examine the compacting of traces by the removal of utilities, it may be necessary to consider the removal of other implementation details as well.

We define an implementation detail as: *Any element of a program whose presence could be suppressed without reducing the overall comprehensibility of the design of a particular feature, component or algorithm*.

This above definition is, of course, dependent on the design component or algorithm being considered, which is one of the reasons why the software engineer needs to be given some control over the parameters of our algorithm, such as the threshold mentioned above.

Utilities are clearly one kind of implementation details. Other examples of implementation details include constructors/destructors, accessing methods, etc. These elements are used simply to *support* the implementation of the system by performing functions that need to be performed in any system, rather than to implement the operations distinct to this particular system. A more detailed

discussion about the types of implementation details found in an object-oriented system can be found in [14].

4. Trace Summarization Algorithm

In this section, we present the steps of the trace summarization algorithm. The algorithm is deliberately underspecified, since further research is needed to determine the best settings of certain parameters (see Step 1 below) and how to categorize and detect various other kinds of implementation details in addition to utilities (Step 2). The following are the steps of the algorithm:

Step 1: Set the parameters for the summarization process. A key parameter is the Exit Condition (EC) that will be used to determine when to stop summarizing. More details about setting parameters are presented below.

Step 2: Remove various categories of known implementation details such as accessing methods, constructors and any methods the user manually wishes to exclude from the result.

Step 3: Compute the utilityhood metric (U) for the routines remaining after Step 2:

- While EC is False, remove the routines invoked in the trace that have the highest remaining value of U.

Step 4: Evaluate the result of Step 3, adjust the parameters and run the algorithm again if necessary, or manually manipulate the output (e.g. using a trace analysis tool).

Step 5: Output the final result

A. Some details of Step 1: Setting the parameters

Step 1 of the algorithm sets certain parameters that will guide the summarization process. The first of these is to determine an exit condition (EC) that will be used to stop the filtering process. There are several criteria that could be considered for this purpose. Perhaps the simplest one might be to compute the ratio of the size of the summary to the size of the initial trace. However, due to the various types of repetitions that exist in a trace, using a simple size ratio will often not be useful. For example, simple elimination of the large number of repeated calls in one loop may cut the trace to 10% of its size, without improving our ability to comprehend it very much.

In [11], we introduced a set of practical metrics for measuring various properties of traces. One of these metrics is the number of comprehension units (Scu), which we defined as the number of distinct subtrees of the trace (i.e. a comprehension unit is a distinct subtree of the call tree representing a trace of routine calls). We have therefore found it useful to base the exit condition, EC, on comprehension units. In particular we can define a ratio R that compares the number of comprehension units contained in the summary to the number of comprehension units of the initial trace.

More formally, let:

- Scu(T) = Number of comprehension units of a trace T
- Scu(S) = Number of comprehension units of the summary S extracted from the trace T

Then we define this ratio as R = Scu(S)/Scu(T)

Another parameter to set in Step 1 of the algorithm is the matching criteria used to compute Scu; choosing appropriate criteria allows one to vary the degree of generalization of the trace content. In other words, two sequences of calls that are not necessarily identical can be grouped together as instances of the same comprehension unit if they exhibit certain similarities. In this paper, the grouping we use is of the simplest kind: we ignore the *number of contiguous repetitions* when computing the number of comprehension units. A more discussion about other possible criteria can be found in [3].

B. Explanation of the remaining steps

Step 2 of the algorithm proceeds by removing any known implementation details from the source trace. Examples of implementation details were presented in the previous subsection and include accessing methods, methods that override the methods contained in the language library (e.g. Java.util), constructors, etc.

The software engineers can, as one of the parameters set in step 1, manually specify a list of components to be considered implementation details; these will be removed at this step.

We considered omitting Step 2, and letting Step 3 so all the summarization work is done automatically. However, we found in early trials that better results are obtained by incorporating Step 2 as described here.

Step 3 takes the traces resulting from Step 2 and proceeds by iteratively removing the routines with the highest value of U (utilityhood) until the exit condition is satisfied.

The application of the above steps is mainly automatic. However, there is a need to account for the following situations:

- Situation 1: The resulting summary still contains too much information for the users.
- Situation 2: The resulting summary is too abstract for the users to develop a sufficient understanding of the system behaviour. For example, the removal of a certain widely-used utility might cause the number of comprehension units to drop considerably below the designated threshold.

If either of these situations occurs, the maintainer will find the summary to be uninformative, and will have to adjust the exit condition and re-run the algorithm (Step 4 of the algorithm). The maintainer might alternatively further process the result using a trace analysis tool.

Step 5 is simply a presentation step where the final summary is turned into a visual representation such as a UML sequence diagram and given to the users as output.

5. Case Study

In this section, we present a case study in order to evaluate the effectiveness of the trace summarization approach by applying it to a trace generated from the execution of a Java-based system called Weka (ver. 3.0) [21]. Weka has 10 packages, 147 classes, 1642 public methods, and 95 KLOC.

We do not only apply the algorithm, but also the manual steps involved in its use such as manipulating the results (i.e. Step 4). We then evaluate the overall approach by asking the developers of the system to provide feedback on the final results.

5.1 Usage Scenario

The software feature we selected to analyze is the Weka implementation of the C4.5 classification algorithm, which is used for inducing classification models, also called decision trees, from datasets [22]. Weka proceeds by building a decision tree from a set of training data that will be used to classify future samples. It uses the concept of information gain to determine the best possible way of building the tree. The information gain can be described as the effective decrease in entropy resulting from making a choice as to which attribute to use and at what level of the tree.

Another important step Weka performs is pruning the decision tree. This is done by replacing a whole subtree by a leaf node to reduce the classification error rate. Weka supports various techniques that can be used to evaluate the learning results using the same dataset. In our usage scenario, we chose to apply the *cross-validation technique*, which is a procedure that involves splitting the training data into equally sized mutually exclusive subsets (called folds). Each one of the subsets is then used in turn as a testing set after all the other sets combined have been the training set on which a tree has been built.

5.2 Process Description

To perform trace summarization on runs of the Weka system, we performed the following activities:

a) We instrumented the Weka source code, using our own instrumentation tool based on the BIT framework [9] to insert probes at the entry and exit points of each system's non-private methods. Constructors are treated in the same way as regular methods.

b) We generated a trace of method calls by exercising the target system according to the functionality under study (i.e. C4.5 algorithm): The trace was generated as the system was running, and was saved in a text file containing raw lines of events, where each line represents the full class name, method name, and an integer indicating the nesting level. For simplicity reasons, in the rest of this paper, we refer to the generated trace as *The C45 Trace*.

c) We built the static call graph of the Weka system that is needed for our trace summarization algorithm: One of the most difficult aspects of this step is resolving polymorphic calls. There are several techniques to accomplish this task including Class Hierarchy Analysis (CHA) [2, 5], Rapid Type Analysis (RTA) [2], and Reaching-Type Analysis [18], which differ mainly in the way they estimate the run-time types of the receiver objects. In this case study, we used RTA for its simplicity, efficiency, and tool support [2].

d) Finally, we applied the trace summarization algorithm described earlier to the C45 trace. The results are discussed in the next subsections.

5.3 Quantitative Results

In this section, we present the gain in terms of size achieved by filtering the C45 trace using the trace summarization algorithm.

Step 1: Setting the parameters:

The most important parameter we set is the exit condition, EC. We randomly chose a threshold R = 10% (future research needs to investigate adequate thresholds). That is, we stop the algorithm when the ratio of the number of comprehension units of the resulting trace to the number of comprehension units of the initial trace drops just below 10%.

We also specified the methods to be manually removed in Step 2. We specified that the following methods were to be removed prior to the automatic detection of utilities:

- Methods found in the java.lang.Object class (usually overridden by user-defined classes)
- Methods in the classes of the java.util package (e.g. methods of the Enumeration interface, etc)
- Methods in a class called weka.core.Utils that contains general purpose methods such as grOrEq, etc. We were able to easily see that these are utilities from a quick scan of the Weka documentation.

Step 2: Removing implementation details

Step 2 of the algorithm deals with removing various categories of implementation details. The implementation details removed in this case study include methods of inner classes, accessing methods, constructors/finalizers, etc. A more complete list of implementation details and the rationale behind considered these components as low-level details are discussed in [14].

As shown in Table 2, The C45 trace, referred to it as *T* in the table, contains initially 97413 calls (S = 97413), 275 comprehension units (Scu = 275), and invokes 181 distinct methods (Nm = 181).

The removal of the above implementation details results in a trace $T_{impldetails}$ whose size, S, is 31102 calls (i.e. 32% of the size of the initial trace). Its number of comprehension unit is 120 (44% of Scu of the initial trace) and the number of distinct methods is 95.

Table 2. Quantitative results

	T	$T_{impldetails}$		$T_{utilities}$	
S	97413	31102	32%	3219	3%
Scu	275	120	44%	67	24%
N_m	181	95	52%	51	28%

Step 3: Detecting utilities

Step 3 aims to improve the results obtained in the previous step by detecting and removing utilities. For this purpose, the utilityhood metric was computed for the methods that are invoked in the C45 trace using the Weka static call graph. We proceeded by removing the routines that have high utilityhood value (the ones that are ranked first in the ranking table). After each iteration, we checked whether the exit condition, R = 10%, holds or not. This process continued until the algorithm hit a method called *weka.j48.J48.buildClassifier*.

The removal of this method resulted in a trace that contains 156 calls (0.2% of the size of the initial trace), 20 comprehension units (7% of the number of comprehension units of the initial trace), and 20 routines (11% of the number of routines of the initial trace). Note that this trace contains considerably fewer comprehension units than the threshold (7% compared to 10%). Based on that, we decided to reverse the removal of this method and stop this step at a higher EC threshold.

The resulting trace is called $T_{utilities}$ and it contains S = 3219 calls (3%), Scu = 67 comprehension units (24%), and 51 methods (28%) as shown in Table 2.

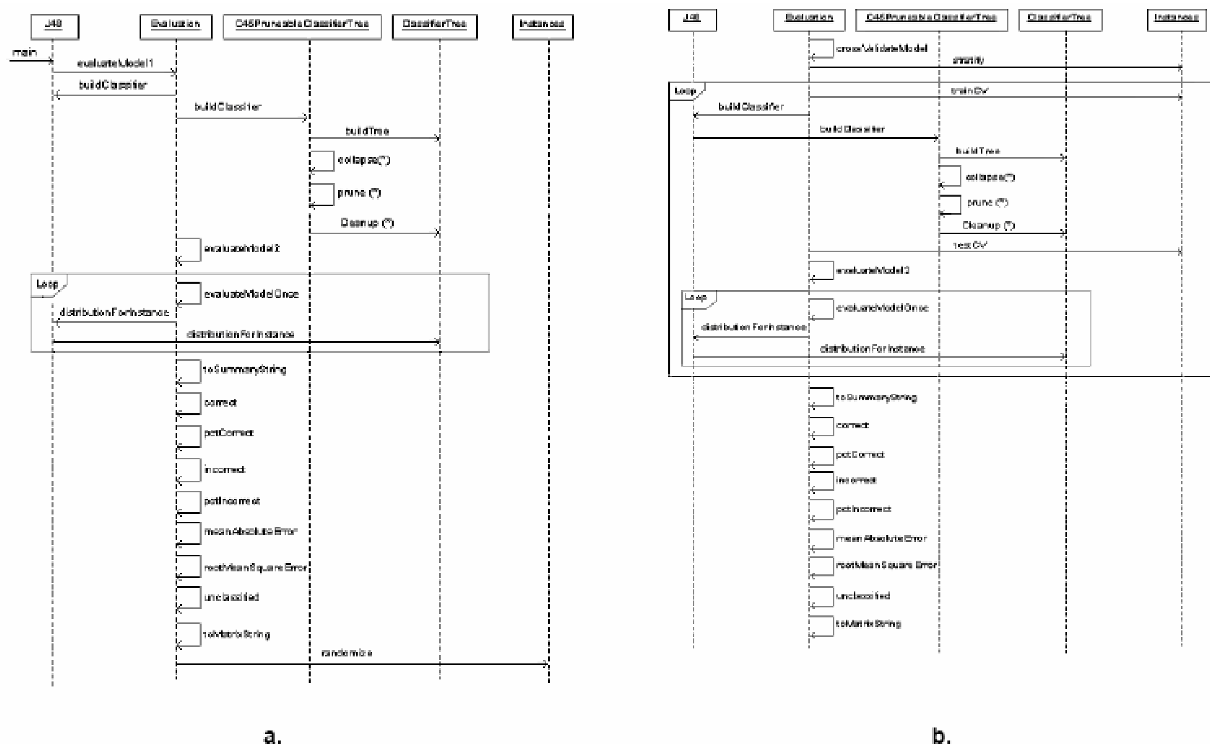

Figure 2. The summary extracted from the C45 Trace and represented as a UML sequence diagram. a) The diagram shows the interactions involved in building the decision tree; b) The remaining part of the diagram. This part shows how the cross-validation process is performed

Step 4: Further Manipulation of the Results

Our initial objective was to have a summary that contains just below 10% of the total comprehension units of the initial trace. However, the algorithm in Step 3 overshot this, so as mentioned we backed up and stopped at 24% (i.e. we are in Situation 1 as described in Section 4). Therefore, we decided to further explore the content of the final trace, $T_{utilities}$, in order to make some further adjustments. This process was done using a trace exploration tool called SEAT (Software Exploration and Analysis Tool) [12] that we have developed to support fast analysis of large traces.

Exploration using the tool showed that the method called *buildTree* generates three additional levels of the tree representation of the trace and most of the methods that appear in these levels have small fan-in (1 or 2) and small fan-out (1 or 2). The role of the *buildTree* method is to build the decision tree that is used by the C4.5 algorithm. At this point, we thought that the details of how the tree is built might be something that can be hidden and that it is sufficient for a summary to have an indication that a tree is being built. Therefore, we decided to remove the methods generated from the *buildTree* method from the summary. The whole process took no more than fifteen minutes and involved expanding and collapsing the tree along with displaying statistics about the content of the trace – these operations are efficiently supported by SEAT. Whether the content of the *buildTree* method should be kept in the summary or not is something that we will discuss in the next section in the context of evaluating the content of the summary. The resulting trace is called T_{adjust} and contains 453 calls (0.5% of the initial size), 26 comprehension units (10% of the initial number of comprehension units), and 26 methods (14% of the initial total of methods).

Finally, the trace was converted into a UML sequence diagram (Figure 2) where the contiguous repetitions have been collapsed (some additional notations have been used to show repeated sequences such as the Loop and (*) constructs). The sequence diagram and the tree representation of the final trace were presented to the Weka software developers for evaluation.

5.4 Questionnaire Based Evaluation

We designed a questionnaire that aims to evaluate various aspects of the extracted summary (in this paper, we only report on the main findings). The questionnaire was given to nine software engineers who have experience with using the Weka system: Either they were part of the Weka development team or they added new features to the system.

Background of the Participants:

We designed three questions to enable us classify our participants according to their expertise in the domain represented by Weka (i.e. machine learning algorithms) as well as their knowledge of the system structure. For each question, the participants selected from fixed values ranging between *'Very poor'* (score of 1) and *'Excellent'* (score of 5). The questions are:

Q1. My knowledge of the Weka system (i.e. classes, methods, packages, etc.) is:

Q2. My knowledge of the domain represented by Weka (i.e. machine learning algorithms) is:

Q3. My experience in software development is:

Table 3 shows the answers of the participants (P1 to P9), which can be divided into three groups according to the knowledge they have of the Weka structure (Q1) as well as the knowledge they have of the domain (Q2). The first group consists of participants P1 and P2 and can be qualified as intermediate users since they have an average knowledge of the Weka internal structure (score of 3) although they have good knowledge of the domain (score of 4). The second group consists of participants P3, P4, and P5 and we refer to them as experienced users (they all scored 4 out of 5 in both questions Q1 and Q2). Finally, the last group includes participants P6, P7, P8, and P9 and we call them experts since their knowledge of the internal structure of Weka as well as the domain is excellent (score of 5 for Q1 and Q2). These are also the users who contributed to the original development of Weka.

In addition, all participants except P1 have good to excellent experience in software development. Presuming that they were also involved in maintaining software, their feedback will certainly help us evaluate the overall effectiveness of a trace summary in performing software maintenance tasks.

Quality of the Summary:

The objective of this category of questions is to assess whether the extracted summary captures the main interactions that implement the traced scenario.

Question Q4 asked:

Q4. How would you rank the quality of the summary with respect to whether it captures the main interactions of the traced scenario?

The participants were asked to select from fixed values ranging between *'Very poor'* (score of 1) and *'Excellent'* (score of 5).

Table 4 shows that intermediate and experienced participants all agree that the summary captures the most important interactions of the trace. Two experts added that it is actually an excellent representation of the main interactions.

Table 3. Background information about the participants

	Intermediate		Experienced			Experts				
	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**Average**
Q1 (System)	3	3	4	4	4	5	5	5	5	4.2
Q2 (Domain)	4	4	4	4	4	5	5	5	5	4.4
Q3 (Experience)	3	4	4	4	4	4	4	5	5	4.1

Participant P9 (an expert) commented that the overall summary is good but he would have preferred to see more details about way the decision tree is built and therefore ranked it as an average (score of 3) representation of the main events. These are the routines we removed manually using SEAT in order to reach a threshold of 10%. This confirms the fact that any tool that would support trace summarization will need to allow enough flexibility so as the users vary the amount of information displayed.

Question Q5 asked:

Q5. If you designed or had to design a sequence diagram (or any other behavioural model) for the traced feature while you were designing the Weka system, how similar do you think that your sequence diagram would be to the extracted summary?

The participants were asked to select from fixed values ranging between *'Completely different'* (score of 1) and *'Very similar'* (score of 5)

Most participants including three experts answered that the sequence diagram they would have designed would most likely be similar (sometimes even very similar) to the summary extracted semi-automatically from the trace. However, participants P3 (experienced) and P9 (expert) commented that their design would have been slightly more concise than the summary. They mostly refereed to the fact that the summary lacks details about building the decision tree.

Question Q6 asked:

Q6. In your opinion, how effective can a summary of a trace be in software maintenance?

The participants were asked to select from fixed values ranging between *'Very ineffective'* (score of 1) and *'Very effective'* (score of 5).

All participants agreed that a trace summary can be effective in software maintenance. Many of them added that this is a very good way to understand what the system is doing when the documentation is out of date or simply inexistent. They also said that recovering the system behavioural design models can be made easier if trace summarization is applied. Indeed, design recovery has always been a challenging task, and when it is done it usually focuses on the system architecture. The techniques for recovering dynamic models are also needed just like in forward engineering where engineers focus on developing both static and dynamic views of the system.

6. Related Work

Although, there are many tools that manipulate execution traces of object oriented systems, they are either tuned to analyze performance problems [4] or they rely heavily on specific visualization techniques [3, 6, 8, 15, 19, 20, 22].

ISVis is a visualization tools that supports analysis of execution traces [6]. ISVis is based on the idea 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.

Given a trace 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.

Richner and Ducasse present a tool, called Collaboration Browser that is used to extract collaboration patterns from traces of method calls [15]. A collaboration pattern consists of a repeated sequence of method calls. Additionally, Collaboration Browser provides a query mechanism that allows the user to search for interesting collaborations. In order to understand the main content of a trace, a user needs

Table 4. Evaluating the quality of the summary

	Intermediate		Experienced			Experts				
Questions	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**Average**
Q4 (Quality)	4	4	4	4	4	4	5	5	3	4.1
Q5 (Diagram)	4	5	3	4	4	4	4	5	3	4
Q6 (Effectiveness)	4	4	5	5	5	4	4	5	4	4.4

to perform several queries. Our approach does not heavily rely on the user's intervention.

Systä presents a reverse engineering environment based on dynamic analysis to extract state machines from traces of object-oriented systems [19]. Her approach is based on the use of SCED [7], 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, which consist of detecting patterns of repeated sequences of events. However, Systä's approach considers exact matches only which limit her approach to small execution traces only.

DynaSee is another reverse engineering tool, developed to support 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. However, he did not develop this concept and his analysis tool does not focus on utility removal in order to help software engineers identify important content.

Amyot et al. suggest tagging the source code at particular places in order to generate a trace that can later be represented using a use case map [1]. This approach has the obvious drawback that it requires from the software engineers to know, in advance, where to insert the tags. It also necessitates the usage of static analysis tools, which is not the case in our approach.

7. Conclusion and Future Work

In this paper, we presented a technique for summarizing the content of large traces. One direct application of this concept is to enable top-down analysis of traces. Another application would be to recover the behavioural design models of the system under study. Our approach consists of suppressing implementation details from traces. We presented a metric that can measure the extent to which a routine can be considered as a utility.

In addition, we presented a trace summarization algorithm that uses the utilityhood metric as its main mechanism. This reduces the size of the trace, using the number of comprehension units metric, to below some threshold. Our approach also assumes that the users will adjust the algorithm's parameters and re-run the algorithm if they wish to try to improve the summary. Users are also expected to be able to use tools that would allow further manipulation of the results.

One direction for future work would be to have the system automatically or semi-automatically suggest appropriate settings for the trace summarization algorithm based on the nature of the trace, as well as the current goals and experience of the maintainer. A key setting to investigate is the exit condition (i.e. when to stop the summarization process).

There is also a need for fundamental research in several areas: For example, we need to fine tune the concept of utilities and algorithms for detecting them. We also need to investigate ways of generalizing the content of traces and therefore lead to more compact summaries. Finally, the technique presented in this paper needs to be integrated with existing trace analysis techniques.

Acknowledgements

We would like to thank the software developers of the Weka system for creating and maintaining this great tool and making it open, and for evaluating the results presented in this paper.

References

[1] D. Amyot, G. Mussbacher, and N. Mansurov, "Understanding Existing Software with Use Case Map Scenarios", In *Proc. of the 3rd SDL and MSC Workshop*, LNCS 2599, pp. 124-140, 2002

[2] D. F. Bacon and P. F. Sweeney, "Fast static analysis of C++ Virtual function calls", In *Proc. of the 10th Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM Press, pp. 324-341, 1996

[3] 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*, pp. 219-234, 1998

[4] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, J. Yang, “Visualizing the Execution of Java programs”, In *Proc. of the International Seminar on Software Visualization,* LNCS 2269, Springer-Verlag, pp. 151-162, 2002

[5] J. Dean, D. Grove, and Chambers, “Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis”, In *Proc. of the 9th European Conference on Object-Oriented Programming,* LNCS 952, Springer-Verlag, pp. 77-101, 1995

[6] D. Jerding, S. Rugaber, "Using Visualisation for Architecture Localization and Extraction", In *Proc. of the 4th Working Conference on Reverse Engineering*, IEEE Computer Society, pp. 56-65, 1997

[7] 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

[8] D. B. Lange., Y. Nakamura, “Object-Oriented Program Tracing and Visualization”, *IEEE Computer*, Volume 30, Issue 5, pages 63-70, 1997

[9] H. B. Lee, B. G. Zorn, “BIT: A tool for Instrumenting Java Bytecodes”. *USENIX Symposium on Internet Technologies and Systems*, 1997, pp. 73-82.

[10] A. Hamou-Lhadj, T. C. Lethbridge, “Reasoning about the Concept of Utilities”, In *Proc. of the 1st ECOOP Workshop on Practical Problems of Programming in the Large*, Oslo, Norway, June 2004

[11] A. Hamou-Lhadj and T. Lethbridge, “Measuring Various Properties of Execution Traces to Help Build Better Trace Analysis Tools”, In *Proc. of the 10th International Conference on Engineering of Complex Computer Systems*, IEEE Computer Society, pages 559–568, 2005

[12] A. Hamou-Lhadj, T. Lethbridge, and L. Fu, “Challenges and Requirements for an Effective Trace Exploration Tool”, *In Proc. of IWPC,* pp. 70-78, 2004

[13] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge, “Recovering Behavioral Design Models from Execution Traces”, In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering,* IEEE Computer Society, pp. 112-121, 2005

[14] A. Hamou-Lhadj and T. Lethbridge, “Techniques for Reducing the Complexity of Object-Oriented Execution Traces”, In *Proc. of VISSOFT, 2003,* pp. 35-40

[15] Richner T. and Ducasse S., “Using Dynamic Information for the Iterative Recovery of Collaborations and Roles”, In *Proc. of the 18th International Conference on Software Maintenance*, IEEE Computer Society, pages 34-43, 2002

[16] M. A. Storey, K. Wong, and H. A. Muller, “How do Program Understanding Tools Affect how Programmers Understand Programs?”, In *Proc. of the 4th Working Conference on Reverse Engineering*, IEEE Computer Society, pages 183-207, 1997

[17] K. Sparck Jones, “Automatic summarising: factors and directions”, In *Advances in Automatic Text Summarization, MIT Press*, pp. 1-14, 1998

[18] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, “Practical virtual method call resolution for Java”, *In Proc. of OOPSLA*, pp. 264-280, 2000

[19] T. Systä, “Understanding the Behaviour of Java Programs”, In *Proc. of the 7th Working Conference on Reverse Engineering*, IEEE Computer Society, pages 214-223, 2000

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

[21] WEKA: http://www.cs.waikato.ac.nz/ml/weka/

[22] I. H. Witten, E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann, 1999

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