# Techniques for Reducing the Complexity of Object-Oriented Execution Traces*

Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge
University of Ottawa
SITE, 800 King Edward Avenue
Ottawa, Ontario, K1N 6N5 Canada
{ahamou, tcl}@site.uottawa.ca

## Abstract

*Understanding the behavior of object-oriented systems is almost impossible by merely performing static analysis of the source code. Dynamic analysis approaches are better suited for this purpose. Run time information is typically represented in the form of execution traces that contain object interactions. However, traces can be very large and hard to comprehend. Visualization tools need to implement efficient filtering techniques to remove unnecessary data and present only information that adds value to the comprehension process. This paper addresses this issue by presenting different filtering techniques. These techniques are based on removing utility methods and the use of object-oriented concepts such as polymorphism and inheritance to hide low-level implementation details. We also experiment with 12 execution traces of an object-oriented system called WEKA and study the gain attained by these filtering techniques*

**Keywords:**

Reverse engineering, program comprehension, dynamic analysis, object-oriented systems, and software visualization.

## 1. Introduction

Understanding object-oriented systems is a challenging task. Such systems are designed with the idea of interactions between objects in mind and in order to fully understand them we need to analyze these interactions rather than merely performing static analysis of the source code.

Information about the execution of an object-oriented system is typically represented in the form of traces of object interactions. Figure 1 shows an example of a very simple trace of method calls where specific objects are substituted by their class type – the term trace of class interactions would be more appropriate in this case. An alternative representation consists of labeling the edges with the messages and nodes with object identifiers or class names.

However, traces can be very large and hard to understand. This is due to the fact that important interactions are mixed with low-level implementation details. To overcome the size explosion problem, many visualization tools and techniques [1, 2, 4, 5] proceed by detecting repeated sequences of object interactions as distinct patterns of execution, which are then rendered in a way that helps a software analyst notice them easily and explore their content.

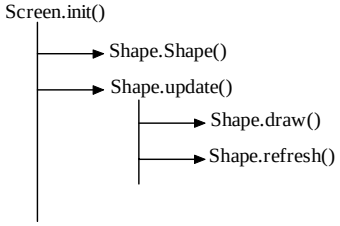


**Figure 1. Trace of method calls. Objects are substituted with their class type**

In this paper we present a set of techniques that aim at filtering the trace by removing unnecessary data with respect to program comprehension. We call this process: *Trace Compression.* For example, utility methods can be removed safely if the goal of the maintenance activity is to understand the overall design of the system, which in turn, can be very useful for design recovery.

Our approach consists of three main steps. First, we preprocess the trace by removing repeated interactions due to loops. Then we detect different types of utilities and remove them. Finally, we use object-oriented concepts, namely, polymorphism and inheritance to hide low-level implementation details.

We also present an experiment that we conducted on 12 execution traces of an object-oriented system called WEKA to estimate the compression gain attained by these techniques.

* This research is sponsored by NSERC

The rest of this paper is organized as follows; the next section discusses the size problem of the traces. In section 3, we present the compression techniques. In Section 4, we describe the experiment and discuss the results.

## 2. The Size Problem

Although traces can be very large, a closer analysis of their content shows that they contain many redundancies. From the comprehension perspective, a software engineer needs to understand a repeated sequence of calls only once and reuse this knowledge whenever it occurs. Therefore, a more accurate way of reasoning about the size problem of a trace should be based on analyzing distinct subtrees of calls instead of the number of lines. We refer to each distinct subtree as *a comprehension unit*.

Figure 2a. shows a trace *T* (the class and method names are represented with one letter to avoid cluttering) that contains 9 calls but only 6 comprehension units as shown in Figure 2b.

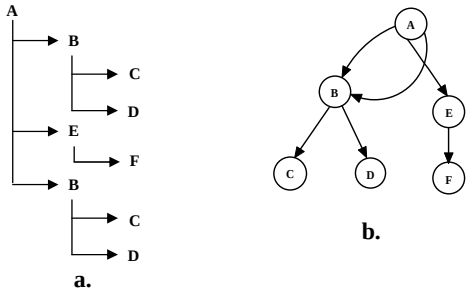


**Figure 2. a. The trace T has 9 calls. b. an acyclic graph that represents the compact form of T and shows 6 comprehension units. Note that the crossing line represents the order of calls**

In order to reduce the trace overhead problem, we need to find ways to group different subtrees as instances of the same comprehension units. The compression techniques[1] presented in this paper aim at accomplishing this.

There are different ways for measuring the compression gain. In this paper, we use a compression ratio and we define it as follows:

- Let $T_1$ be the original trace such as T has $CU_1$ comprehension units.
- Let $T_2$ be the resulting trace after compressing $T_1$ and $CU_2$ is the number of comprehension units of $T_2$
- The compression ratio R is:

$$R = 1 - CU_2/CU_1$$

This means that the higher the ratio the better the compression we get.

## 3. Trace Compression Techniques

### 3.1 Trace preprocessing

The first step consists of preprocessing the trace by removing contiguous repetitions of method calls or sequences of method calls that are due to loops. However, consider the trace of Figure 3, the two sequences rooted at *B* are not identical but can be considered similar from the comprehension point of view if the number of repetitions of *C* of the first subtree is ignored.

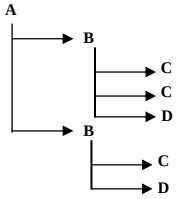


**Figure 3. The subtrees rooted at B can be considered the same if number of repetitions is ignored**

Therefore, we expand the preprocessing stage to consider two contiguous subtrees as the same even though the number of contiguous repetitions of their nodes is not exactly identical. This will result in a better compression ratio without a considerable loss of the trace content. Other matching criteria such as the ones presented by DePauw et al. [3] can also be used

### 3.2 Removing utilities:

The following are some criteria that can be used to rate the extent to which a method is a utility.

**Constructors and destructors:**

Constructors and destructors are used simply to create and delete objects, rather than to implement the core system operations. Therefore, it may be best to ignore them while trying to understand the behavior of a specific scenario. However, if the maintenance task involves such things as performance analysis or detecting memory leaks, then preserving constructors and destructors would be important.

**Accessing methods:**

Accessing methods are methods that return or modify directly the values of member variables. Accessing methods are used as a means to reinforce information hiding. Although, software engineers tend to follow the same naming convention for accessing methods, which consists of prefixing them with “get/set” followed with the name of the variable, it might be necessary to perform data flow analysis of the class that defines them to automatically detect them.

[1] We are not talking about data compression in the conventional sense (which results in unintelligible output), but rather, compression of the visible output so that it can be more easily understood.

**Utility classes:**

It is a common practice for software developers to create utility classes that can be used by other classes of the system. If those classes are already known by the software maintainer then she or he can remove them from the trace. There may also be a need to automatically detect such classes. For this purpose, the class dependency graph can be of assistance. Utility classes correspond usually to the graph nodes with a very large number of incoming edges and a very small number of outgoing edges [8].

### 3.3 Techniques based on OO concepts:

**Polymorphic methods:**

T. Lethbridge and R. Laganière define polymorphism as "a property of object-oriented software by which an abstract operation may be performed in different ways, typically in different classes" [7]. The methods that implement the operation need to have the same name although they might have different signatures. Polymorphism is typically implemented using method overloading and inheritance.

Although the semantics of these methods should be the same, the execution trees that derive from them can be significantly different due to the way they are implemented. However, it is unlikely that the software maintainer will need to look inside the encapsulation to see the implementation details if only an abstract view of the design is needed, which leads to an opportunity to remove these details (since they merely implement an abstract operation). Removing such details can result in a significant compression. This concept applies to interfaces as well since an interface is considered as a pure abstract class.

## 4. Experiment

### 4.1 Description and settings

We experimented with an object-oriented system called WEKA version 3.0.6 [9]. WEKA is a tool that implements several data mining and machine learning algorithms including classification algorithms, association rules generators and clustering techniques. In addition to that, WEKA implements several filters that transform the input datasets in different ways such as adding or removing attributes, removing instances from the dataset and so on. WEKA is implemented in Java and contains around 160 classes and over 1680 methods. For more information about WEKA, please refer to [9].

We used our own instrumentation tool that is based on BIT [6] to add probes at each entry and exit point of the system public methods. Constructors are considered as regular methods. However, private methods are not instrumented to reduce the amount of processing time.

Traces are generated as the system runs and saved in a text file. Although WEKA comes with a GUI version, every WEKA algorithm and feature can be executed from the command line. We favored the command line approach over the GUI to avoid encumbering the traces with GUI components. A trace file contains the following information: Thread name; full class name (e.g. weak.core.Instance); method name and a nesting level that maintains the order of calls

We noticed that all WEKA algorithms use only one thread. Therefore, the thread name information is useless for this experiment. However, in case of a multi-threaded system, one needs to break the trace into different threads and apply the compression techniques to each of them.

**Table 1. The traces used in this experiment**

| Trace | Algorithm or Filter | Description |
|---|---|---|
| 1 | Cobweb | Clustering algorithm |
| 2 | IBk | Classification algorithm |
| 3 | OneR | Classification algorithm |
| 4 | Decision Table | Classification algorithm |
| 5 | J48 (C4.5) | Classification algorithm |
| 6 | Apriori | Association algorithm |
| 7 | Attribute Filter | Filter |
| 8 | Add Attribute | Filter |
| 9 | Merge Two Values | Filter |
| 10 | Instance | Filter |
| 11 | Swap Attribute Values | Filter |
| 12 | Split Dataset | Filter |

The main objective of this experiment is to estimate the gain attained by the compression techniques. We chose to analyze 12 execution traces of WEKA. Table 1. describes the algorithms and filters that correspond to each trace.

### 4.2 Experiment Design

The compression techniques presented in this paper can be combined in different ways. Each combination will eventually result in a different compression ratio. We narrow down all the possible results to the following:

- Initial information about the trace such as the number of lines, the number of comprehension units, etc.
- The gain attained after preprocessing the trace.
- The gain attained after removing constructors from the preprocessed trace
- The gain attained after removing accessing methods from the preprocessed trace. For this purpose, we noticed by inspecting the source code that WEKA follows the "set/get" naming style.
- The gain attained after removing utility classes from the preprocessed trace. For this purpose, we analyzed WEKA documentation to discover eventual utility classes. We found that WEKA contains a class called

*Utils* where many utility methods such as *doubleToString, eq, etc* are defined

- The gain attained after removing the details of polymorphic methods from the preprocessed trace. In this paper, we focus on overriding only. Overloaded methods that are defined in the same class are considered identical since we do not take into account the arguments list. However, methods that are overloaded in different classes are not considered in this paper for simplicity reasons.
- Finally, we also combine these techniques together to show the gain attained after removing all utilities (constructors, accessing methods…) and removing details of polymorphic methods from the resulting trace. However, this is not the only approach to combining. Future research can focus on other possibilities.

Table 2. summarizes the variables used to describe the results in a more precise way:

### 4.3 Results and discussion

Table 3 shows general information about the traces. Although some traces contain over 100 000 lines (e.g. Trace 6), we notice that they do not contain a lot of distinct methods (e.g. only 65 methods in trace 6). The number of comprehension units is also low. This means that there are many repetitions in the trace that are either due to loops or the presence of the same sequences of calls all over the trace. The preprocessing stage reduces considerably the size of most of the traces although Traces 4, 5 and 6 are still considerably large. We also notice that Trace 5 has a very large number of comprehension units, which might imply that it is the most complex trace. It is also interesting to see that ignoring repetitions when removing contiguous repetitions of sequences of calls results in a higher reduction of the number of comprehension units for large traces compared to small traces. For example, Trace 10 and 12 still keep the same number of comprehension units although the number of lines is considerably smaller after the preprocessing stage.

Table 4. shows the results of removing utilities and the call hierarchies that are derived from polymorphic calls. We notice that removing the constructors for large traces (Traces 1 to 6) results in a higher reduction compared to removing accessing methods. This is due to the fact that most of these methods were already removed during the preprocessing stage. Another reason is that, these traces use a large number of objects. On the other hand, small traces do not use a lot of objects and removing constructors might not be that important, which explains why removing accessing methods still gives a slightly better compression ratio. Removing the methods of the class *Utils* seems to give almost the same result for all the traces.

**Table 2. Variables used to represent the results**

| Variable | Description |
|---|---|
| $N_{init}$ | The number of calls of the initial trace |
| $CU_{init}$ | The number of comprehension units of the initial trace |
| Classes | The number of distinct classes of the system that the initial trace contains |
| Methods | The number of distinct methods of the system that the initial trace contains |
| $N_{prep}$ | The number of calls of after preprocessing the initial trace. Let us call the resulting trace $T_{prep}$ |
| $CU_{prep}$ | The number of comprehension units of $T_{prep}$ |
| $R_{prep}$ | Compression ratio = $1 - CU_{prep} / CU_{init}$ |
| $N_{const}$ | The size of the resulting trace after removing constructors from $T_{prep}$ (preprocessed trace) |
| $CU_{const}$ | The number of its comprehension units |
| $R_{const}$ | $= 1 – CU_{const} / CU_{prep}$ |
| $N_{access}$ | The size of the resulting trace after removing accessing methods from $T_{prep}$ |
| $CU_{access}$ | The number of its comprehension units |
| $R_{acces}$ | $= 1 – CU_{access} / CU_{prep}$ |
| $N_{util}$ | The size of the resulting trace after removing the methods of the class Utils from $T_{prep}$ |
| $CU_{util}$ | The number of its comprehension units |
| $R_{util}$ | $= 1 – CU_{util} / CU_{prep}$ |
| $N_{poly}$ | The size of the resulting trace after removing the details of polymorphic methods |
| $CU_{poly}$ | The number of its comprehension units |
| $R_{poly}$ | $= 1 – CU_{poly} / CU_{prep}$ |
| Poly_Meth | Number of polymorphic methods |
| $N_{cum\text{-}utils}$ | The size of the resulting trace (let us call it $T_{util}$) after removing all utilities (constructors, accessing methods…). |
| $CU_{cum\text{-}utils}$ | The number of its comprehension units |
| $R_{cum\text{-}utils}$ | $= 1 – CU_{cum\text{-}utils} / CU_{prep}$ |
| $N_{cum\text{-}poly}$ | The size of the resulting trace after removing polymorphic calls details from $T_{util}$ |
| $CU_{cum\text{-}poly}$ | The number of its comprehension units |
| $R_{cum\text{-}poly}$ | $= 1 – CU_{cum\text{-}poly} / CU_{prep}$ |

On the other hand, removing the details of polymorphic methods reduces considerably the size of Trace 1 but reduces its comprehension units by only 45.77% as shown in Table 4. The analysis of Trace 1 showed that the method buildClusterer() is the main cause behind this high reduction. WEKA implements two clustering algorithms and both of them consist of classes that override the method buildClusterer(). Building clusters might involve going through the dataset several times to find relationships between them. This usually generates very large hierarchies of calls, which explains the significant reduction when these details are hidden.

**Table3: General information about the trace and the results of preprocessing them**

| Trace | Classes | Methods | $N_{init}$ | $CU_{init}$ | $N_{prep}$ | $CU_{prep}$ | $R_{prep}$ |
|---|---|---|---|---|---|---|---|
| 1 | 10 | 63 | 193121 | 108 | 6015 | 79 | 27% |
| 2 | 12 | 92 | 37882 | 185 | 3719 | 113 | 39% |
| 3 | 10 | 89 | 27554 | 223 | 4557 | 124 | 44% |
| 4 | 19 | 150 | 154185 | 305 | 29576 | 224 | 27% |
| 5 | 23 | 152 | 95118 | 469 | 25933 | 306 | 35% |
| 6 | 9 | 65 | 156792 | 317 | 19810 | 127 | 60% |
| 7 | 11 | 76 | 1902 | 83 | 281 | 83 | 0% |
| 8 | 10 | 71 | 2534 | 80 | 351 | 80 | 0% |
| 9 | 10 | 73 | 2245 | 84 | 752 | 83 | 1% |
| 10 | 10 | 68 | 1248 | 73 | 247 | 73 | 0% |
| 11 | 10 | 73 | 2256 | 83 | 374 | 82 | 1% |
| 12 | 10 | 71 | 1398 | 79 | 289 | 79 | 0% |

Similarly, Traces 3 and 5 represent two classification algorithms represented by two classes that override the buildClassifier() method. This method also generates large hierarchies of method calls. The results presented here go along with the idea of abstracting out the trace to extract high-level interactions. For example, a software engineer might only want to know that, at this point of time, a classifier or a clusterer is being built without having to go into the details.

**Table 4: Removing utilities and details of polymorphic methods from the preprocessed traces**

| T | $N_{const}$ | $CU_{const}$ | $R_{const}$ | $N_{access}$ | $CU_{access}$ | $R_{access}$ | $N_{util}$ | $CU_{util}$ | $R_{util}$ | $N_{poly}$ | $CU_{poly}$ | $R_{poly}$ | Poly. Meth |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5305 | 67 | 15.19% | 6009 | 75 | 5.06% | 6008 | 73 | 7.59% | 289 | 46 | 41.77% | 4 |
| 2 | 3329 | 91 | 19.47% | 3409 | 99 | 12.39% | 3599 | 104 | 7.96% | 1973 | 95 | 15.93% | 2 |
| 3 | 3898 | 106 | 14.52% | 4260 | 115 | 7.26% | 4449 | 116 | 6.45% | 1253 | 80 | 35.48% | 3 |
| 4 | 27039 | 183 | 18.30% | 28068 | 186 | 16.96% | 27759 | 213 | 4.91% | 20916 | 158 | 29.46% | 5 |
| 5 | 21408 | 275 | 10.13% | 25124 | 290 | 5.23% | 24297 | 286 | 6.54% | 1633 | 99 | 67.65% | 5 |
| 6 | 18880 | 113 | 11.02% | 19610 | 116 | 8.66% | 19771 | 119 | 6.30% | 19810 | 127 | 0.00% | 0 |
| 7 | 228 | 70 | 15.66% | 252 | 65 | 21.69% | 267 | 79 | 4.82% | 227 | 68 | 18.07% | 3 |
| 8 | 299 | 68 | 15.00% | 329 | 68 | 15.00% | 332 | 75 | 6.25% | 285 | 65 | 18.75% | 3 |
| 9 | 674 | 70 | 15.66% | 718 | 68 | 18.07% | 721 | 78 | 6.02% | 626 | 66 | 20.48% | 3 |
| 10 | 208 | 61 | 16.44% | 219 | 52 | 28.77% | 232 | 69 | 5.48% | 192 | 61 | 16.44% | 3 |
| 11 | 316 | 69 | 15.85% | 350 | 68 | 17.07% | 355 | 77 | 6.10% | 274 | 66 | 19.51% | 3 |
| 12 | 242 | 67 | 15.19% | 262 | 63 | 20.25% | 271 | 74 | 6.33% | 245 | 72 | 8.86% | 3 |

Trace 4 represents an algorithm that creates a decision table and generates classifiers out of it. Although some polymorphic methods were found, the number of lines is still very large; this might be due to the complexity of this algorithm.

It is interesting to notice that the number of overridden methods that appear in the traces as indicated by the variable *Poly_Meth* is low. Trace 6, for example, does not contain any polymorphic method. In fact, Trace 6 corresponds to the only association algorithm that is implemented in WEKA, called Apriori. The class Apriori is created to build association rules which are an important part of this algorithm. However, this class does not have a superclass, which explains why polymorphism was not used here. This result is very interesting because it shows the limitations of using polymorphism (based on overriding) to hide details and requires investigating other means for hiding these details. Perhaps, a more general definition of utility methods can lead the way.

Finally, Table 5 shows the cumulative results of removing utility methods and removing polymorphic methods from the resulting trace. The table shows higher

compression ratios in terms of the number of comprehension units and very high reduction of the number of lines. The compression ratio of all the traces has increased expect for Trace 6 because it does not have polymorphic methods. Different combination of these techniques will lead to different compression ratios. However, future work needs to involve the users in order to discover which combinations suit them the best.

**Table 5: Cumulative results**

| T | $N_{cum\text{-}utils}$ | $CU_{cum\text{-}utils}$ | $R_{cum\text{-}utils}$ | $N_{cum\text{-}poly}$ | $CU_{cum\text{-}poly}$ | $R_{cum\text{-}poly}$ |
|---|---|---|---|---|---|---|
| 1 | 5296 | 59 | 25.32% | 202 | 30 | 62.03% |
| 2 | 2925 | 71 | 37.17% | 1320 | 61 | 46.02% |
| 3 | 3514 | 92 | 25.81% | 734 | 52 | 58.06% |
| 4 | 23770 | 138 | 38.39% | 17048 | 96 | 57.14% |
| 5 | 19047 | 247 | 19.28% | 1006 | 68 | 77.78% |
| 6 | 18645 | 96 | 24.41% | 18645 | 96 | 24.41% |
| 7 | 190 | 48 | 42.17% | 173 | 44 | 46.99% |
| 8 | 264 | 53 | 33.75% | 227 | 45 | 43.75% |
| 9 | 615 | 52 | 37.35% | 540 | 44 | 46.99% |
| 10 | 173 | 38 | 47.95% | 132 | 32 | 56.16% |
| 11 | 279 | 52 | 36.59% | 216 | 44 | 46.34% |
| 12 | 205 | 48 | 39.24% | 180 | 44 | 44.30% |

## Conclusions and future directions

Dynamic analysis is very useful for understanding the behavior of object-oriented systems. The analysis of traces of object interactions can bridge the gap between low level implementation details and high level domain concepts, if effective filtering techniques exist. Interactions between objects are typically depicted in traces of method calls. In this paper, we presented many techniques that can help hide implementation details and reveal only important interactions. The experiment showed several results that can be used by tool builders to improve their tools. First, we showed that the number of calls in the trace is not the major factor of complexity. Traces can be very large but have few comprehension units. Another interesting result is that traces always need to be preprocessed. Although, using these compression techniques separately might result in a good compression ratio, they work best when combined.

Future work should focus on several areas such as considering a broader definition of utility methods. We also need to experiment with many other software systems to understand how to combine the compression techniques in order to extract the most important interactions that many software designers agree about.

## References


[1] W. De Pauw, D. Kimelman, and J. Vlissides, "Modeling Object-Oriented Program Execution", *In Proc. 8th European Conference on Object-Oriented Programming*, Bologna, Italy, 1994, pp. 163-182.,

[2] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides, "Visualizing the Behavior of Object-Oriented Systems", *In Proc. 9th Conference on Object-Oriented Programming Systems, Languages, an Applications*, Portland, Oregon, USA, Oct. 1994, pp. 326-337

[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, COOTS, 1998*, pp. 219-234

[4] D. Jerding, S. Rugaber, "Using Visualization for Architecture Localization and Extraction." *In Proc. 4'th Working Conference on Reverse Engineering*, Amsterdam, Netherlands, Oct. 1997

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

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

[7] T. C. Lethbridge, R. Laganière, *Object-Oriented Software Engineering: Practical Software Development using UML and Java*, McGraw Hill, 2001

[8] H. A. Müller, M. A. Orgun, S. Tilley, J. Uhl, "A Reverse Engineering Approach To Subsystem Structure Identification", *Journal of Software Maintenance: Research and Practice, Vol 5, No 4,* December 1993, pp. 181-204

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