Preprint of the paper to appear in the Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR'04)

Managing trace data volume through a heuristical clustering process based on event execution frequency

Andy Zaidman Serge Demeyer

University of Antwerp
Department of Mathematics and Computer Science
Lab On Re-Engineering
Middelheimlaan 1
2020 Antwerpen
Belgium
{Andy.Zaidman, Serge.Demeyer}@ua.ac.be

Abstract

To regain architectural insight into a program using dynamic analysis, one of the major stumbling blocks remains the large amount of trace data collected. Therefore, this paper proposes a heuristic which divides the trace data into recurring event clusters. To compose such clusters the Euclidian distance is used as a dissimilarity measure on the frequencies of the events. Manual inspection of these event sequences revealed that the heuristic provides interesting starting points for further examination.

1. Introduction

When programming a piece of software, the programmer has to build a mental bridge that connects the code he's writing and the program behavior he's trying to accomplish [13]. Conversely, when a programmer is trying to get an understanding of a system, he's actually trying to get the reverse mapping: from the functionality he's trying to understand to the code that's making it all happen. It is estimated that 30 - 40% of a programmer's time is spent in studying old code and documents in order to get an adequate understanding of a software system before making changes [16, 20]. The manner in which a programmer gets understanding of a software system varies greatly and depends on the individual, the magnitude of the program, the level of understanding needed, ... [8] While in principle it is necessary to understand the entire system before making changes, in practice it is essential to use an *as-needed strategy*: you want to get an understanding of the part of the system that you're concerned with as quickly and as thoroughly as possible [10].

In the reverse engineering literature, several program understanding techniques have been explored. Most of them can be classified as *static* since they start from a static description of the program under study: either the program code or the documentation, or a combination of both. A few program understanding techniques however, exploit *dynamic* information, because they analyse the program's behavior while its running, or study a post-mortem dump of its event trace. Dynamic analysis is especially relevant for program understanding, because of the "use as needed" strategy mentioned before: you only trace those parts of the program that you're really concerned with, as such getting a very quick and thorough overview of what's actually happening.

Unfortunately, this thoroughness comes at a cost. Even a short program run quickly generates thousands of events, so the question of *scaleability* inevitably arises [9, 15]. Therefore it's necessary to study techniques which group trace data into interesting event clusters. In traditional dynamic analysis, where most research concentrates around program optimization, *soundness* plays a crucial role in developing a technique to guarantee behavior preservation [11]. Dynamic analysis for program understanding relaxes the problem considerably, because in this context we can afford a few misses.

Therefore, this paper proposes a *heuristic* [6] which provides a reverse engineer with interesting starting points for code and program-structure discovery. The heuristic itself is based around the following idea: in object-oriented systems, objects work together to reach a certain goal, i.e. perform a certain functionality. This collaboration is

expressed through the exchange of messages according to a certain interaction protocol. This interaction protocol, although not static in the sense that the sequence of exchanged messages is always identical, is based on a common theme. It is this that makes it interesting to go and have a look at the frequencies of the messages used. More specifically: the assumption is that messages that work together to reach a common goal, will always be executed more or less the same number of times. It is on this assumption that we build a heuristic.
As case studies we use Fujaba and Jakarta Tomcat. The former being a good example of an application with a heavy GUI, the latter being a good representative for a server-like application.

2. Formal background

In a more formal way, we can say that we are actually looking for evidence of the concepts of **dominance** and **post-dominance** [18]:

> We say that an instruction x dominates an instruction y if the trace prefix which ends with y also contains an instruction x. In other words an instruction x dominates an instruction y if and only if the only way to make sure that y gets executed means that x has already been executed. x post-dominates y if every trace postfix which begins with y also contains x. Or one can say that x post-dominates y if every execution of y indicates that x will also be executed in a relatively short period of time.

Frequency of execution can help us determine whether two events will always be executed at – more or less – the same time.

3. The Heuristic

Applying the heuristic and analyzing its results is defined as a six-step process. In this section, we'll describe each of these six steps and explain the inner workings.

Step 1: Define a filter A first reduction step is the elimination of irrelevant events from the trace, i.e. low-level method calls or method calls to parts of the system we're not interested in. Table 1 shows the results of a normal tracing operation and of a tracing operation which filters out all method calls belonging to classes from the Java API[1] (Java 2 Standard Edition, release 1.4.1).

[1]The Java API is a standard library that contains functionality for dealing with strings, inter process communication, containers, ...

	Jakarta Tomcat 4.1.18	Fujaba 4
Execution time (without tracing)	48s	70s
Classes (total)	13 258	15 630
Events	6 582 356	12 522 380
Unique events	4 925	858 505
Classes (filtered)	3 482	4 253
Events	1 076 173	772 872
Unique events	2 359	95 073

Table 1. Comparison of total tracing versus filtered tracing.

As we can see in Table 1, the reduction is significant as the resulting trace is between 7 and 15% of the original trace.

Step 2: Trace using the filter This step consists of running the program according to a specific scenario which activates the functionality the reverse engineer is interested in. The result of this step is a file which contains a chronological list of all method calls which were executed during the scenario. The filter defined under step 1 is crucially important here, because it allows to reduce the *probe effect*[2] [1].

Step 3: Frequency Analysis This step is inspired on the *Frequency Spectrum Analysis* (FSA) work of Thomas Ball [2]. The idea is that using the calling frequency of the executed methods, we can distinguish (1) high-level functionality from more low-level functionality and (2) can relate certain method-calls through their frequency.
In our case, we count the events, i.e. we create a map which contains for each unique method found in the trace the number of times it has been called. We decided to perform this step post-mortem, i.e. after the tracing operation itself (Step 2), in order to make sure that the tracing operation remained lightweight and thus didn't influence the program under observation (*see also the probe effect*[2]).

[2]This effect can be compared to the Heisenberg principle (1927), well-known in the field of quantumphysics. This principle states that the more is known about the place of a particle, the less can be known about it's speed, and vice versa. The same is true in our context: if the level of detail of the trace capturing mechanism is high, i.e. the more you know about the program, the behavior of the program gets more affected. The converse is also true.
Some examples of how the program under review can be influenced are: different scheduling of threads, slow responsiveness from the application which gets users agitated so that the users keeps repeating his/her commands to the program, ... All these factors can influence the quality of the trace data.

Step 4: Frequency Annotation We walk over the original trace and annotate each event with the frequency we find in the map we've created in Step 3. The result is a chronological list of executed methods, with an added first column which respresents the frequency of execution of the method listed in column two. Remark that the values found in the first column represent the total number of times a method is executed during the scenario. It is important to use the total frequency of execution because we want to distinguish methods working together based on their relative frequencies. An example:

...	
543	$XMLParser.init()$
978	$XMLParser.parseString(String)$
1243	$XMLParser.closingTagFound()$
1243	$XMLParser.validXMLElement()$
543	$XMLParser.close()$
...	
543	$XMLParser.init()$
978	$XMLParser.parseString(String)$
1243	$XMLParser.closingTagFound()$
...	

As we wish to make an abstraction, we explictely omit object identifiers (OID's) as we aren't looking for specific instances of interaction protocols.

Step 5: Dissimilarity Measure Using the annotated trace we sample the frequencies of a sequence of method calls, resulting in a characteristic *dissimilarity measure* for that sequence of events. Conceptually this characteristic dissimilarity measure can be compared with a *fingerprint*. When a regular pattern emerges in the values of the characteristic measure, we talk about a *frequency pattern*.
The sampling mechanism uses a sliding window mechanism to walk over the annotated trace. When going over the trace, we let the window fill up; once the window size is reached, we apply the dissimilarity measure on the frequencies of the events in the window and then discard the contents of the window. We repeat the process until the end of the trace is reached.

...	...	
f_{i-1}	$event_{i-1}$	
$\mathbf{f}_i$	$event_i$	apply dissimilarity measure
$\mathbf{f}_{i+1}$	$event_{i+1}$	
$\mathbf{f}_{i+2}$	$event_{i+2}$	
$\mathbf{f}_{i+3}$	$event_{i+3}$	
$\mathbf{f}_{i+4}$	$event_{i+4}$	
f_{i+5}	$event_{i+5}$	
...	...	

We illustrate the process with a window size 5. The dissimilarity index is applied on the frequencies of the events that lie in the interval [f_i, f_{i+4}], after that 'i' is incremented by 5, the window size, and the process is repeated. The current implementation thus uses simple consecutive blocks for the windows. In the future we will also be looking at a sliding window mechanism, which would allow for overlaps to be taken into account.

In our experiment, we've taken the most commonly used distance metric, namely the Euclidian distance [5, 7].

$$d = \sum_{j=1}^{w-1} \sqrt{(f_{j-1} - f_j)^2}$$

Euclidian distance: with 'w' the window size and f_j as the frequency of the j-th event in the current window on the trace

Step 6: Analysis When we've covered the previous steps, we are now in a position to analyze the dissimilarity measure and the trace looking for clues that point to interesting event clusters.

- On the one hand we are looking for regions in the trace where the frequency of execution is (almost) identical. Looking at the events in these regions, we observe that there is a very small number of methods, i.e. less than ten, that is frequently executed for some time. From our two case studies we found that a good example of this situation is the traversal of a linked list where the same operations are performed on the listmembers.
  Other cases, where the dissimilarity value is near-zero, are where there is a frequently executed interaction protocol.
- On the other hand we look for recurring patterns in the dissimilarity index. As we've mentioned before, we call this category *frequency clusters*. A single occurrence of such a recurring event cluster is then called a *frequency signature*. Finding clusters of events with a similar pattern in the dissimilarity index points to the recurrent activation of the same (end-user) functionality. The fact that the dissimilarity index for these regions is not the same, but similar, can be attributed to the polymorphic behaviour of the application. The same polymorphic behaviour is also one of the reasons why the dissimilarity index isn't near-zero.

Hypothesis Having explained the inner workings of the heuristic, we are now ready to formulate our four-part hypothesis.

1. The majority of the found clusters will in fact be *frequency patterns*. Frequency patterns are mostly the result of using polymorphism and because polymorphism is abundantly present in object-oriented software, we expect this type of clusters to be numerous.

2. Enlarging the window size introduces noise in the frequency signatures because sequences of methods which logically form a whole are perhaps smaller than the window size. This can lead to false negatives.

3. Shrinking the window size introduces noise on the results because when frequency signatures become so small, everything becomes a frequency pattern. This can lead to false positives.

4. Regions in which a certain action is repeated become easily discernible: if at a point in time x a certain functionality is activated and at another point in time y the same functionality is activated, this will be visible in the dissimilarity values.

4. The experiment

This section will provide empirical data on and anecdotal evidence about the clusters found in the event traces in the two case-studies we used. We consider these results to be preliminary, because (1) the validation of the results has only been done for two cases, (2) the results have only been compared manually with the traces. We want to verify the results more thoroughly in another experiment which would allow us to visualize the clusters in parallel to browsing the traces in order to do a more thorough validation.

We use two well-known open-source Java programs in our experiments:

1. For the representative of a non-graphical, server-like program we chose Jakarta Tomcat of the Apache Software Foundation[3]. This application is well-known and widely used for its excellent Java Servlet and Java Server Pages (JSP) implementations.

2. On the other hand we have chosen Fujaba[4], an open-source UML tool with Java reverse engineering capabilities. Due to its heavy use of the Java Swing API it's an excellent representative for applications with a heavy GUI.

We performed three experiments. We will present them in a brief overview to give a clearer view on why we performed each of them.

1. The first experiment, performed on Jakarta Tomcat, was executed in order to validate our hypothesis about window sizes. Starting from the same event trace we used different window sizes when applying our algorithm.

2. The second experiment recreates the first one, but this time for our other case-study, namely Fujaba.

3. The third experiment on the other hand, focusses on a slightly different aspect. We wanted to know how a very specific usage scenario would be projected onto the dissimilarity graph. Therefore, we defined a usage scenario with a small number of repetitive actions in it and looked at the results of our heuristic.

As a final note we wish to add that for all three experiments we made use of the filteringtechnique that eliminates method calls to classes from the Java API, see also Table 1.

4.1. Jakarta Tomcat 4.1.18

4.1.1 Experiment 1

As we pointed out in the previous subsection, this experiment was set-up to show the results of differing the window size in our heuristic. We discuss the results of the experiment by looking at Figures 1 through 4. These figures represent the dissimilarity value of a group of methods, the current window, at a certain point in time during the execution of the program. As such, the X-axis can be interpreted as being *time*. The Y-axis then is the *dissimilarity value*.

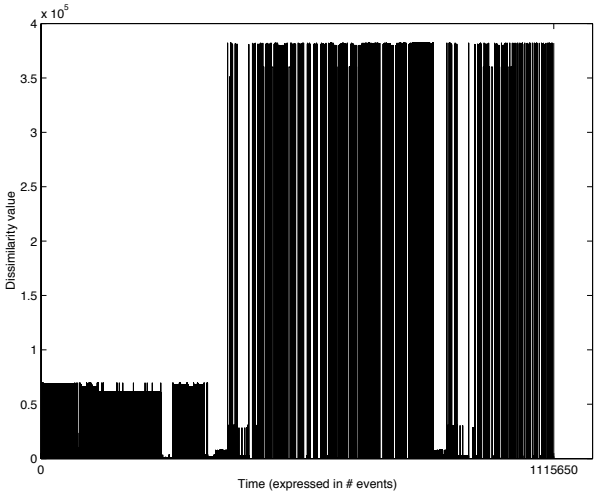
We start by looking at the smaller window sizes and then gradually increase the size of the window. The reason behind this is that we want to make sure that by increasing the window size, we don't get any false negatives due to the noise we expect when using larger window sizes.
For the purpose of detecting the frequency patterns we talked about earlier, we zoomed in on an interval of the chart in Figure 4. The result of this is shown in Figure 5. When comparing the results of our first experiment with the hypotheses we introduced in the previous section, where does this leave us?

1. From figures 1 through 4 it is clear that regions where the dissimilarity is near-zero are rather limited. In this trace we can only detect a handful of them. Frequency patterns however are much more frequent, just look at Figure 5: between index 86000 and 99000 on the X-axis there is a clear repetition in the dissimilarity measure.

2. Increasing the window size doesn't seem to have an influence on the regions with near-zero dissimilarity. This is mainly due to the fact that the execution sequences in these regions remain constant for

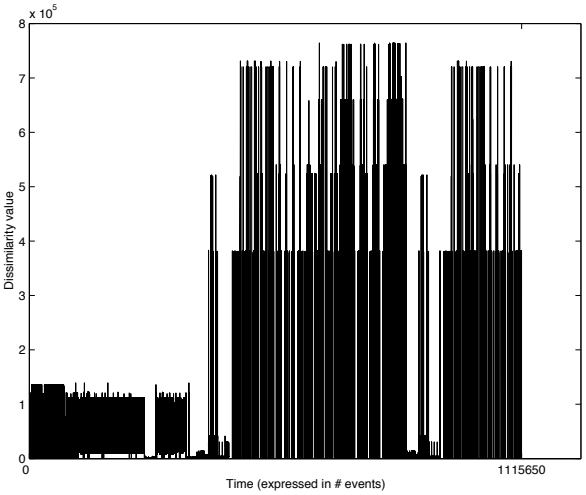
[3] More information can be found at: http://www.apache.org/

[4] Fujaba stands for "**F**rom **U**ML to **J**ava **a**nd **B**ack **A**gain", more information on this project can be found at: http://www.uni-paderborn.de/cs/fujaba/



As the chart shows, until the 150.000th x-value the dissimilarity measure (Y-axis) remains low. After that there is a small period where the dissimilarity is near-zero. The interval where the dissimilarity is low, points to a high repetition of method invocations (either identical method invocations or method invocations related through their frequency of invocation). The most common instances of this kind of repetition are for example the traversal of a linked list.

Figure 1. Visualization of Tomcat with dissimilarity measure using window size 2



Low or almost zero values for the dissimilarity measure are still clearly visible when using a window size of 5 events. No extra places where there is a low dissimilarity value have been added, so there's no report on false positives. The false negatives didn't come through either: no regions where the dissimilarity value is near-zero have disappeared with regard to Figure 1

Figure 2. Visualization of Tomcat with dissimilarity measure using window size 5

some time, i.e., the execution pattern is longer than the (large) window size. Experimenting with window sizes in the neighborhood of 100, however, does show that noise is introduced. This is true for both the regions with near-zero dissimilarity and the frequency patterns. On the other hand, frequency patterns are more easily discernible with slightly larger window sizes: in figures 3 and 4 for example, they are much easier to spot than in figures 1 and 2.

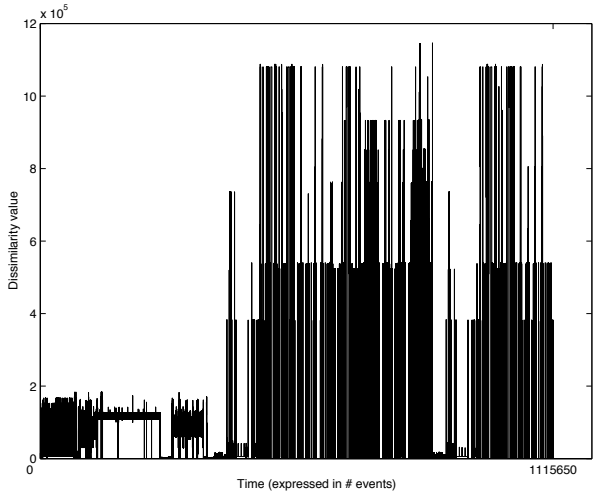
3. Taking a small window size on the other hand makes no real difference for distinguishing regions with near-zero dissimilarity. Frequency patterns however do become more difficult to see in the trace with small window size.

Before going over to our second experiment, we first turn our attention to the specifics of the already mentioned frequency patterns. Some intervals show a recurring pattern in the dissimilarity measure. We took Figure 4 and blew up the interval [80000, 100000] for the X-axis. The result is shown in Figure 5.

Frequency patterns are even more interesting than the regions that have a near-zero dissimilarity value. Why? Because (1) these frequency patterns are much more common and (2) because of the polymorphic nature of object-oriented software, it is much more realistic to find clusters in which not every event is executed the same number of times over and over again. This can be explained by the late binding mechanism in which the exact method invocation depends on the type of data to be processed. We illustrate this with an example:

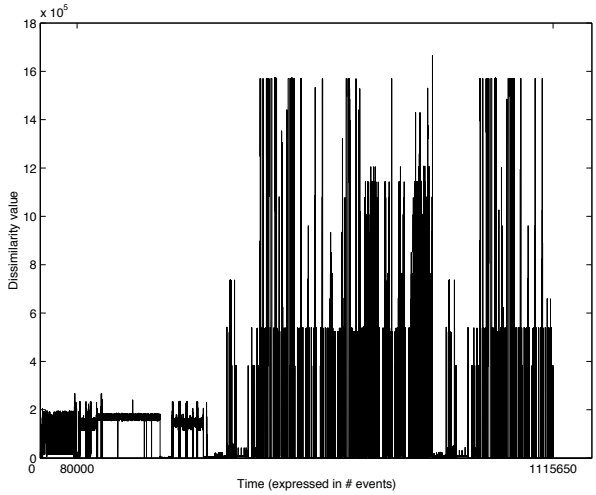
execution sequence 1	*execution sequence 2*
event$_a$	event$_a$
event$_b$	event$_b$
event$_c$	**event**$_x$
event$_d$	**event**$_y$
event$_e$	event$_e$

In this example, after event$_a$ and event$_b$ have been executed, due to polymorphism there is a choice between for example events c, d or events x, y.
Let's suppose $f_a = f_b = f_e$ and that $f_a \neq f_c$, $f_a \neq f_d$. Neither for execution sequence 1 nor execution sequence 2 would this yield a zero dissimilarity value. The chance



When doubling the window size to 10, there is still no indication of false negatives. Intervals with low dissimilarity are still easily discernible.

Figure 3. Visualization of Tomcat with dissimilarity measure using window size 10



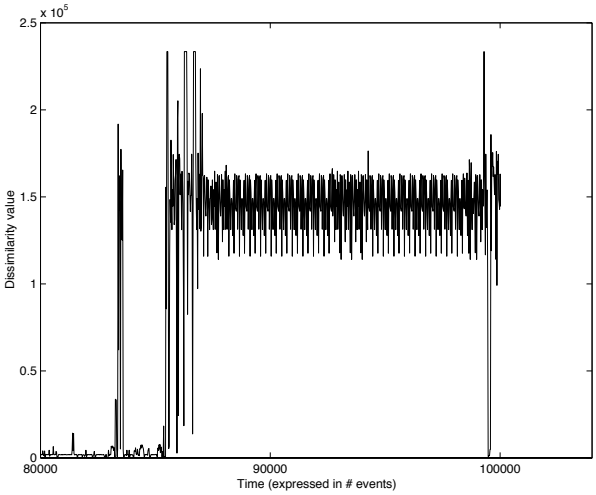
We again doubled the window size and have no indication of false negatives.

Figure 4. Visualization of Tomcat with dissimilarity measure using window size 20

that $f_c = f_x$ and $f_d = f_y$ is pretty slim. That's why both execution sequences give rise to a unique frequency signature. Unique, because when $f_c \neq f_x$ or $f_d \neq f_y$ they will certainly generate different values for the dissimilarity measure.

Recording these frequency patterns as clusters when they tend to be present multiple times in the event trace is a good idea, because they have some interesting properties:

- They often tend to repeat themselves in the same locality.
- From manual inspection we've learned that the repetition of method invocations is more realistic because of the greater number of unique events involved. More realistic because in the case of clusters formed through the near-zero dissimilarity measure criterium, the repetition seems concentrated around only a few methods. The clusters based on the frequency pattern criterium however are constituted out of a variety of method invocations. That makes these clusters much more realistic in large-scale object-oriented systems.



We've made an annotation with two horizontal stripes to the figure where there is a clear pattern of repetition. In fact, if you look closely, you see a one-time repetition pattern in the annotated region. Considering the fact that this pattern ranges over around 10000 events, we have to say that this is important enough to investigate further.

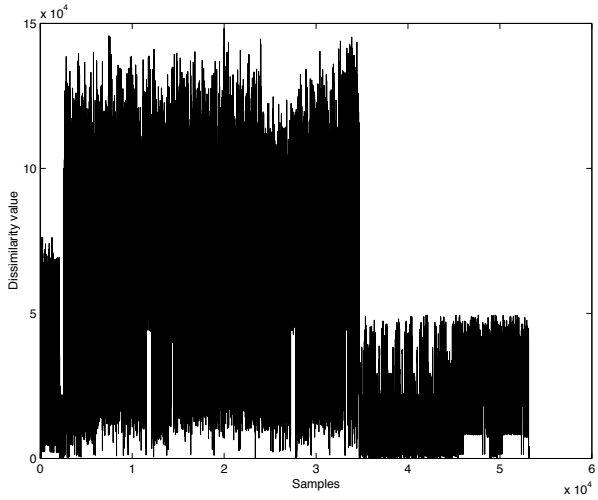
Figure 5. Blowup of the interval [80000, 100000] of Figure 4 to show frequency patterns

4.2. Fujaba 4.0

For this case-study we've opted to perform two separate experiments. One experiment is a repeat of the Tomcat experiment, but this time on Fujaba. The second is an experiment whereby a scenario with some repetitive actions is observed.

4.2.1 Fujaba experiment 1

We won't show the results for all window sizes as we did for the Tomcat case-study, we'll just go straight to the largest window size, namely window size 20.
When looking at Figure 6, what immediately stands out is the oscillation of the dissimilarity measure in the interval [1, 35000]. From manual inspection, we learn that this behavior stems from the animated "splash screen"[5] from Fujaba. From index 35 000 onwards, we begin executing the scenario. This scenario consists of the drawing of a simple class hierarchy. Intuitively it's logical to assume that drawing a number of classes also invokes a sequence of methods the same number of times. This is exactly what Figure 6 shows when you look at the interval [35000, 45000].



The most interesting interval is [35000, 45000]. Here we clearly see a four-time repetition pattern.

Figure 6. Visualization of Fujaba with dissimilarity measure using window size 20

[5] A splash screen is an introduction screen for a program that's starting up. In the case of Fujaba it is animated and has text scrolling over it. Graphically it is quite heavy, so this can explain the heavy oscillating behavior of the dissimilarity measure.

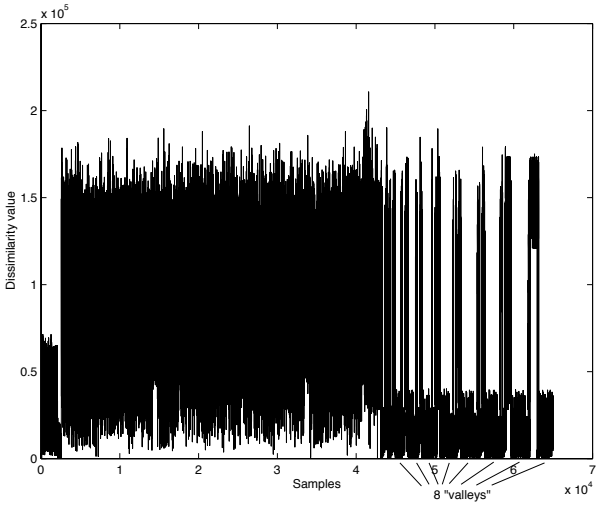
Although this second experiment isn't a good example for the near-zero dissimilarity measure, it clearly does support the *frequency patterns* theory. The regular pattern that is visible after X-index 35000 is a good example of this. With respect to our hypotheses, we can say that the conclusions from the Tomcat case remain valid here: medium to large window sizes remain the most interesting to distinguish the frequency patterns.

4.2.2 Fujaba experiment 2

Remaining with Fujaba, we conducted a second experiment. We defined a specific usage-scenario with a highly repetitive nature. This scenario can be described as follows: after starting the program, we defined a class-hierarchy. The hierarchy consisted of one abstract base class, several child-classes, who themselves also had a number of child-classes. The total hierarchy consisted of 8 classes with a maximum nesting depth of 3.
Intuitively we expect that the visualization of the dissimilarity metric would show an 8-time repetition. Figure 7 shows that this is indeed the case. The graph clearly shows 9 peaks in the dissimilarity value. Although these are interesting, we are more interested in the 8 interlying "valleys" (or depressions). The reason that these 8 regions are valleys and not peaks can be explained by the fact that the methods who are working together to draw such a class are closely related through their frequencies. These 8 valleys point to the functonality that's activated for drawing the class that's added to the hierarchy. Note however, how the valleys become more stretched as we add more classes to the hierarchy. Inspection of the trace showed that this is due to the *layout algorithm* which needs more actions to perform the (re)layout operation due to the higher number of objects that have to be placed.

Instead of showing listings from the actual trace to show you the repetitive nature of the actions that can be seen around the X-axis interval [44 000, 54 000], we decided to use techniques for the detection of duplicated code. This allows us to show you that the valleys in Figure 7 contain a lot of repetition in the executed methods. This evidences only the repetitive nature of method invocations when performing a specific functionality. The second aspect, namely that methods working together to achieve a common goal have the same (or related) method invocation frequency became clear after manual inspection of the annotated trace (see also section 3, step 4).
The result of applying duplicate code detection techniques is shown in the mural view of Figure 8. Two interesting properties of this figure are:

1. (short) lines that run parallel to the main diagonal. This points to (quite lengthy) duplication.



This graph shows the dissimilarity evolution of Fujaba scenario with a high degree of repetition. The executed scenario consisted of drawing a class hierarchy consisting of 8 classes. The 8 corresponding "valleys" are annotated on the graph. Note that the valleys become somewhat larger towards the end, this can be attributed to the fact that the layout algorithm has to be called more times as more objects are placed on the drawingcanvas.

Figure 7. Visualization of a Fujaba scenario with a high degree of repetition

2. recurring patterns in the lower right quadrant of the figure. The very similar shape of the white spots in the lower right quadrant also points to a lot of repetition in the execution trace.

Moreover, when we compare this with the findings from Figure 7 we find that the regions which are *white* in Figure 8 are the regions which come out as "peaks" in Figure 7. This evidences the fact that the methods which are performed during the peaks can in fact be seen as glue code. This is in accordance with our earlier findings from the dissimilarity value: regions with a high degree of repetition (and/or methods who work together) show a relatively low dissimilarity value.

4.3. Results

What can all these charts learn us?

1. Regions with near-zero dissimilarity value are easy to spot, even with a window size that's quite large. This means that we can easily use a big window size, thus reducing the amount of data and still find sequences of events that logically form a whole.

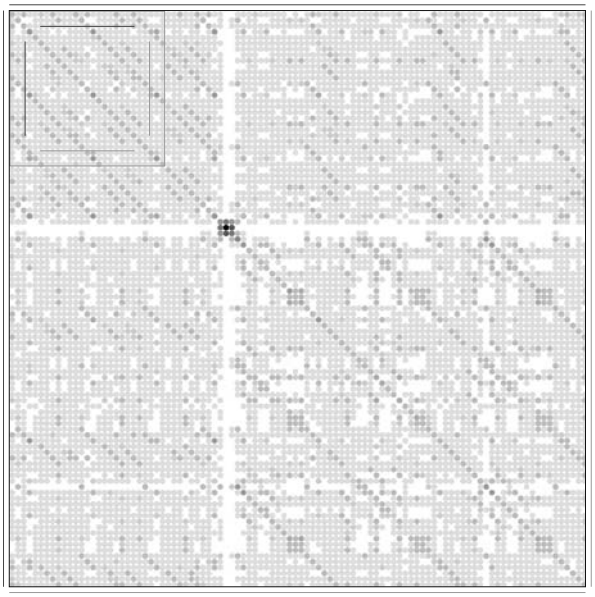


Figure 8 shows a mural view of the trace in the interval 44 000 till 54 000. This mural view is produced by Duploc [4], a tool for detecting duplicated code.
In short, this technique plots a point every time a duplicate line in the event trace is found. Logically, the diagonal (from top left to bottom right) always contains such a dot. However, it becomes more interesting when you can see other lines and/or patterns in it: this points to real duplication. As such, this plot evidences that there is indeed a lot of repetition in the trace.

Figure 8. Duploc output of part of the trace (event interval 44 000 to 54 000).

2. Frequency patterns are much more common than the first type of clusters. How common they are exactly is difficult to state at the moment. We presume that the size of the program, i.e. the number of classes and methods, plays a crucial role. Programs in which certain actions are performed frequently also form better candidates for detecting frequency patterns. Both Tomcat and Fujaba fall into this category. From our

experiences with the two case studies presented here, our predictions are that of the full event trace, some 70% of the events can be catalogued as belonging to a detected cluster. This number sounds reasonable, but is nevertheless perhaps not optimal. A full 100%, however, can in our opinion never be reached because of the necessary "*glue code*" between components of a large software system.

3. The experiment in which we used a scenario with a highly repetitive nature learned us that it is quite easy to spot functionality when using our heuristic. A groups of methods working together for reaching a common goal leave behind a very characteristic frequency pattern.

4.4. Open questions

After performing our case studies, some open questions remain. For example, we didn't establish an optimal window size for our heuristic. Indeed, a large window size makes the analysis-step more efficient because there is less data, i.e. less dissimilarity measures, to work with; frequency patterns are also easier to distinguish. Taking the window size *too* large, however, means a loss of precision. The *ideal* window size doesn't exist however, because it's related to the size and structure of the program. More research however can be spent in determining a window size that's *acceptable* for a wide range of programs.
A second open question is the dissimilarity measure used. Although the Euclidian distance is the most commonly used distance metric, it isn't perhaps the best one for our type of experiment [5, 7]. Future experiments with different distance metrics should bring clarity here.

5. Related work

A number of techniques have been proposed to present the end-user with an analyzed trace. These techniques try to formulate a synthesis for the user, so that the huge raw data is reduced to a more scalable, more readable, easier to interpret format. We will give a brief overview of some of these techniques.

Remark that the techniques listed below are – like ours – based solely on dynamic analysis. Other techniques that combine static and dynamic analysis are the works of Tamar Richner [14] and Tarja Systä [17].

Software reconnaissance "Software reconnaissance" [20] is a technique whereby several execution scenarios are defined. The set of scenarios can be divided in two: one subset of scenarios activate a certain functionality the user is interested in, while the other subset doesn't have this functionality activated. The resulting traces from these 2 subsets are than analyzed for that piece of trace that is present in the *positive* subset, i.e. the traces that result from executing the scenarios that contain the specific functionality, and not present in the *negative* subset.

Mapping traces to architectural views This technique [19] aims to cluster events together into architectural views based on basic - a priori - knowledge the user has of the system, i.e. the main components have to be known. The fact that there has to be some knowledge of the system can be seen as a disadvantage, but in practice due to the iterative nature of regaining architectural insight, this will almost surely be the case with many techniques.

Statistical clustering The technique of real-time statistical clustering [12] was designed for use in parallel processing environments. The basic idea behind it however remains interesting when considering ways to reduce event trace data. Using performance measures from the processors in this multi-processor environment, similar temporal trajectories in the performance measures are detected. Regions of the trace which coincide with these similar temporal trajectories are considered clusters. It is easy to see that this technique can also be adapted to single-processor systems.

6. Future work

Up until now we've concentrated on discovering clusters in the event trace. These clusters can help in visually detecting patterns in the trace, but can also help in analyzing the trace further. One of the steps we hope to make in the near future is finding clusters that are (1) identical or (2) very similar and abstracting these clusters into a pattern of execution. When we can make this abstraction, we will be able to reduce the size of the trace considerably [3].

7. Conclusion

We have shown that clustering event traces based on the frequency of events is a real possibility. We employed the Euclidian distance as the dissimilarity measure and have also shown that increasing the window size, doesn't impair with distinguishing regions with near-zero dissimilarity values. Moreover, we have found that regions that contain so-called *frequency patterns* are even more interesting, because the conditions causing them are (1) more frequent and (2) more realistic in object-oriented software. We conclude by saying that from the manual inspections we did of the case-

studies our predictions are that using our heuristic helps in dividing around 70% of the trace into clusters.

References

[1] J. Andrews. Testing using log file analysis: tools, methods, and issues, 1998. Proc. 13 th IEEE International Conference on Automated Software Engineering, Oct. 1998, pp. 157-166.

[2] T. Ball. The concept of dynamic analysis. In *ESEC / SIGSOFT FSE*, pages 216–234, 1999.

[3] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, 1998.

[4] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In H. Yang and L. White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 109–118. IEEE, 1999.

[5] C. Fraley and A. E. Raftery. How many clusters? which clustering method? answers via model-based cluster analysis. *The Computer Journal*, 41(8):578–588, 1998.

[6] J. H. Jahnke and A. Walenstein. Reverse engineering tools as media for imperfect knowledge. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 22–31. IEEE, 2000.

[7] L. Kaufman and P. Rousseeuw. *Finding groups in data*. Wiley-Interscience, 1990.

[8] A. Lakhotia. Understanding someone else's code: Analysis of experiences. *Journal of Systems and Software*, pages 269–275, Dec. 1993.

[9] J. R. Larus. Efficient program tracing. *Computer*, 26:52–61, May 1993.

[10] K. Lukoit, N. Wilde, S. Stoweel, and T. Hennessey. Tracegraph: Immediate visual location of software features. In *ICSM 2000 Proceedings*, pages 33–39, 2000.

[11] M. Mock. Dynamic analysis from the bottom up, 2003. In ICSE 2003 Workshop on Dynamic Analysis (WODA 2003) Portland, Oregon May 9, 2003.

[12] O. Y. Nickolayev, P. C. Roth, and D. A. Reed. Real-time statistical clustering for event trace reduction. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):144–159, Summer 1997.

[13] M. Renieris and S. P. Reiss. ALMOST: Exploring program traces. In *Proc. 1999 Workshop on New Paradigms in Information Visualization and Manipulation*, pages 70–77, 1999. http://citeseer.nj.nec.com/renieris99almost.html.

[14] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In H. Yang and L. White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 13–22. IEEE, 1999.

[15] R. Smith and B. Korel. Slicing event traces of large software systems. In *Automated and Algorithmic Debugging*, 2000.

[16] D. Spinellis. *Code Reading: The Open Source Perspective*. Addison-Wesley, 2003.

[17] T. Systa. Understanding the behavior of java programs. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 214–223. IEEE, 2000.

[18] T. Tilley, R. Cole, P. Becker, and P. Eklund. A survey of formal concept analysis support for software engineering activities. In G. Stumme, editor, *Proceedings of the First International Conference on Formal Concept Analysis - ICFCA'03*. Springer-Verlag, February 2003. to appear.

[19] R. J. Walker, G. C. Murphy, J. Steinbok, and M. P. Robillard. Efficient mapping of software system traces to architectural views. Technical Report TR-2000-09, 07 2000. http://citeseer.nj.nec.com/walker00efficient.html.

[20] N. Wilde. Faster reuse and maintenance using software reconnaissance, 1994. Technical Report SERC-TR-75F, Software Engineering Research Center, CSE-301, University of Florida, CIS Department, Gainesville, FL.