

Visualizing Similarities in Execution Traces

Bas Cornelissen

Delft University of Technology
The Netherlands
s.g.m.cornelissen@tudelft.nl

Leon Moonen

Delft University of Technology
The Netherlands
Leon.Moonen@computer.org

Abstract

The analysis of execution traces is a common practice in the context of software understanding. A major issue during this task is scalability, as the massive amounts of data often make the comprehension process difficult. A significant portion of this data overload can be attributed to repetitions that are caused by, for example, iterations in the software's source code.

In this position paper, we elaborate on a novel approach to visualize such repetitions. The idea is to compare an execution trace against itself and to visualize the matching events in a two-dimensional matrix, similar to related work in the field of code duplication detection. By revealing these similarities we hope to gain new insights into execution traces. We identify the potential purposes in facilitating the software understanding process and report on our findings so far.

1. Introduction

In the field of dynamic analysis, the post-mortem analysis of execution traces has been an active research topic for a long time. While traces can be rich in information and offer more accurate data than static analysis, they are typically rather large and not very human-readable. Significant efforts have been made to tackle this scalability issue (e.g., [11]) and many techniques and tools have been developed over time: in earlier work, for example, we proposed a set of techniques and a tool aimed at rendering execution traces more navigable [3].

One of the main causes for the information overflow in execution traces is the repetitive nature of certain event sequences. These sequences, which typically stem from loop constructs, consume huge amounts of space while offering little additional information to the reader. As a result, the development of summarization techniques has been conducted to the present day: Hamou-Lhadj et al. [6], for example, have addressed this issue by first identifying utility routines and consequently summarizing these routine. Kuhn et al. [8] represent traces as signals, which (to an extent) vi-

sualizes repetitions. De Pauw et al. [10] and Hamou-Lhadj et al. [5] propose algorithms to identify similar subtrees in traces.

In this position paper, we propose to adopt a visualization technique to gain more insight into large execution traces. Taken from the domain of code duplication detection, we use a technique involving *similarity matrices* to effectively depict the repetitive nature of a trace. Our focus is not so much on pattern identification or summarization as it is on the *visualization* of (large) groups of recurrent events. We identify the potential purposes of such visualizations, and present the results of our preliminary experiments.

2. Approach

The technique that we propose to use is similar to the work of Ducasse et al. who, in the field of code duplication visualization, proposed to use a scatter plot to compare source code to itself [4]. Their strategy is to perform a line-by-line comparison using a simple string matching function and to visualize the matches as dots in a two-dimensional matrix. The resulting matrix shows diagonal lines in case of duplicate code fragments.

It is our opinion that the same method is applicable in the context of execution trace analysis. We propose to compare a trace against itself and to visualize the (partial) matches, with the intent of showing the patterns that are formed by recurrent call sequences, thus providing more insight into the program's behavior.

In particular, it is our belief that a matrix visualization of this comparison process has the following purposes:

- **Recurrent pattern visualization.** Similar to the case of duplicated code detection, we expect recurrent (sequences of) calls to show up in the visualization as clearly visible patterns. Insight into repetitive behavior makes it easier to grasp large amounts of trace information, and is a first step towards the development of new trace abstraction techniques that exploit these repetitions.

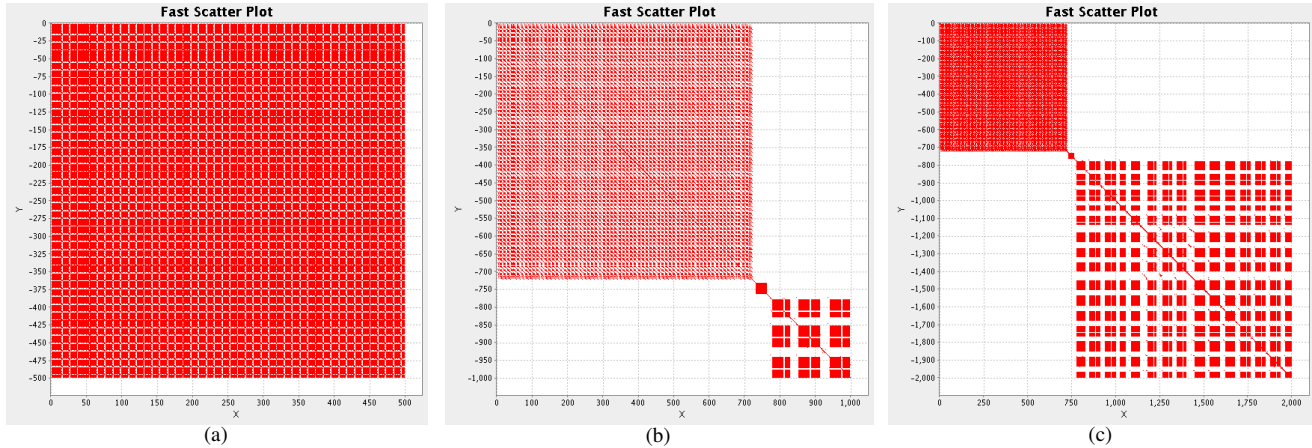


Figure 1. Similarity matrices of three partial Checkstyle traces: (a) 500 events, (b) 1000 events, and (c) 2000 events.

- **Execution phase visualization.** The interleaving between trace fragments that bear *different* degrees of similarity indicates execution phases and phase transitions [9, 3]. Initial knowledge of a program’s phases of execution alleviates the trace comprehension process, and can be useful as a first step towards a more focused examination.
- **Polymorphism visualization.** By varying the matching criterion used to compare two events, we can also detect recurrent call sequences of which the calls differ only slightly, e.g., in case of late binding. Additionally, studying occurrences of late binding can provide information about the program’s input and/or output [1].

3. Preliminary experiments

To assess the feasibility of our approach, we have performed a series of preliminary experiments in which we analyzed (parts of) an example trace. The subject system is CHECKSTYLE¹, an open-source tool that validates Java code. The program is instrumented, and executed according to a typical scenario during which all method and constructor calls are registered. The resulting execution trace contains roughly 32,000 events.

3.1. Visualizing repetitive behavior

Since our early prototype tools can not cope with traces of this magnitude – i.e., performing and visualizing 30,000 * 30,000 comparisons within a reasonable timeframe – in the first experiment we have processed three trace *fragments*. These fragments pertain to the first 500, 1,000, and 2,000

events of the CHECKSTYLE trace, and serve to provide a rough indication of the usefulness of our approach.

The matching criterion that we use is simple: we consider two calls to be similar if they involve the same caller, callee, signature, and runtime parameters.

In visualizing the resulting matches, we use the FastScatterPlot that is part of JFREECHART². While this visualization solution is not particularly efficient, it is sufficiently fast for the initial experiments described here.

3.1.1. Results

Figure 1 shows the similarity matrices for each of the three trace fragments. The axes each symbolize the trace, whereas the red dots represent the data points, i.e., similarities between events according to our matching criterion. We now take a closer look at the matrices and attempt to clarify their contents.

1. Judging by the density of the data points in Figure 1(a), the first 500 calls display a great degree of similarity. Upon closer inspection, we learn that a series of similar initialization tasks lies at the basis of these calls.
2. When considering the first 1,000 events (Figure 1(b)), we observe that the aforementioned stage ends somewhere between the 700th and 800th call. What follows almost instantly is another short repetition sequence. Finally, a very characteristic shape that looks like a grid of solid squares is shown at the lower right.
3. By broadening our perspective even more, we obtain the similarity matrix in Figure 1(c). The main conclu-

¹Checkstyle 4.3, <http://checkstyle.sourceforge.net/>

²JFreeChart 1.0.6, <http://www.jfree.org/jfreechart/>

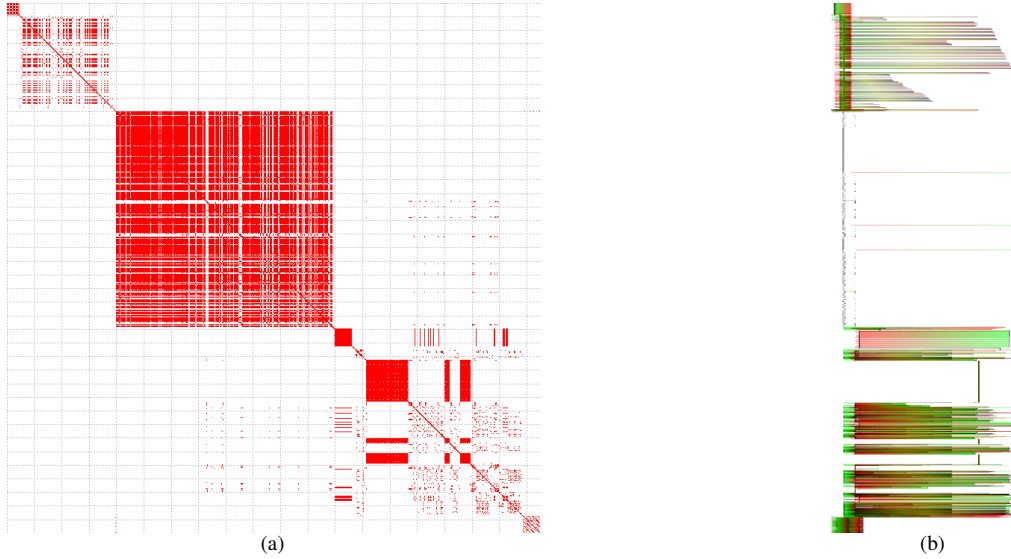


Figure 2. Visualizing the entire Checkstyle trace, using (a) a sampled similarity matrix, and (b) a massive sequence view.

sion here is that the characteristic pattern that we discovered in the previous step is continued all the way until the 2,000th event; close inspection reveals that a long series of exceptions is being created at this point.

3.2. Visualizing execution phases

In the second experiment, we focus on the visualization of execution phases. The idea is to process the entire CHECKSTYLE trace and to use the matrix visualization to recognize execution phases. To address the scalability issue that was described in the previous experiment, we choose to employ a *sampling* technique in the data generation process: we calculate datapoints for *sets* of calls. The rather straightforward criterion that we use for now is to consider the similarities between every n -th call with $n = 16$, thus reducing the 32,000 events in our trace to a dataset of 2,000 by 2,000 points.

Additionally, we employ an alternative visualization method that enables the visualization of execution phases. This second method is the *massive sequence view* from our earlier work [7, 3], which can easily cope with traces of up to 500,000 calls. Viewing both visualizations side by side allows for the comparison between the two techniques in the context of phase detection.

3.2.1. Results

The results of this experiment are shown in Figure 2, which shows the sampled matrix on the left and the massive sequence view on the right. Based on the views, we can draw

several conclusions:

1. The matrix visualization clearly shows the various stages that are negotiated during the execution. While initially we had suspected the sampling criterion to be too strict, it turns out that many datapoints satisfy both the sampling and the matching criteria, resulting in a series of very distinct shapes.
2. The datapoints are less dense in certain phases than they are in others. In the former case (i.e., less colored datapoints), either the matching criterion or the sampling criterion is satisfied less often, or a combination of both. Using a lower value for n would render the latter criterion more lenient, thus producing more colored datapoints.
3. The matrix view bears a striking resemblance to the massive sequence view, in that each phase in the one view can easily be identified in the other. In particular, the rather solid shapes in the matrix view correspond to the vertical lines in the massive sequence view. The similarity between the two types of views supports our claim that execution phases can be (largely) attributed to recurrent patterns, and that the visualization of these patterns is an effective tool in identifying those phases.

3.3. Visualizing polymorphism

As an example of visualizing occurrences of polymorphism, we analyzed a series of trace fragments while utilizing a different matching criterion. As a suitable criterion in this con-

text, we consider two calls to be polymorphic in case they have common callers and signatures, but different types of callees.

3.3.1. Results

One fragment in the CHECKSTYLE trace turned out to be particularly rich in such calls. The resulting similarity matrix of this fragment has been visualized in Figure 3, which shows six red “columns”. A closer look at the execution trace in this interval points out that calls with two certain signatures are being invoked on a series of `Check` instances; Moreover, in the case of six certain checks, these calls lead to additional non-polymorphic interactions, which accounts for the five “interruptions” in between the aforementioned columns.

3.4. Observations

From the results in this Section, we can formulate a series of general observations.

Observation 1. The similarity matrices effectively show the degrees of repetition within trace fragments. The repetitiveness is reflected by the density of the data points: the less whitespace there is between the data points, the smaller and more repetitive the call sequences are.

Observation 2. The matrices allow for the recognition of *phases* in the program’s execution. In the matrix that shows 2,000 events, we can already distinguish between various stages. By sampling the data points and showing all 32,000 events, the identification of CHECKSTYLE’s major phases and their transitions requires little effort.

Observation 3. When using different matching criteria, various types of similarities can be highlighted in the execution trace. Our experiments point out that occurrences of polymorphism are an example type of events that can easily be distinguished in this manner.

Observation 4. With regard to data generation, our current prototype implementation is not very scalable: processing the entire trace requires 900,000,000 string comparisons. This task would greatly benefit from the use of more involved data structures.

Observation 5. Sampling the input data seems to be a promising technique: even the very straightforward sampling criterion with $n = 16$ in our experiment yields meaningful results.

Observation 6. The visualization aspect of our technique is not trivial, as (in case of no sampling) large traces will typically produce massive amounts of data points. For this reason, more effort is required to develop or reuse techniques that make optimal use of screen real-estate.

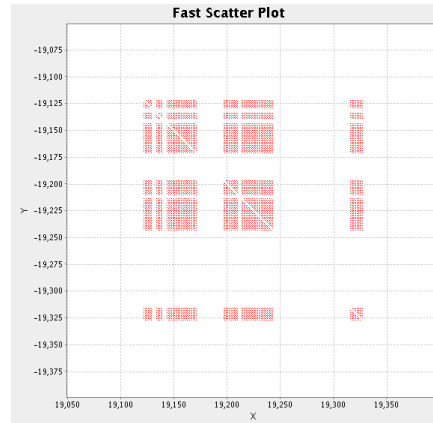


Figure 3. Matrix visualization of a trace fragment involving polymorphism.

4. Open Issues

In order for our technique to be applicable in practice, we need to address several important issues.

4.1. Matching criterion

In our preliminary experiments we have employed a visualization that only allows for one type of data point, i.e., one color. For this reason we have utilized a very strict matching criterion: either two calls match, or they do not match.

It would be interesting to use a criterion that is more lenient. As an example, one could consider assigning scores to partial matches (e.g., in case only the runtime parameters do not match) and using different colors for those data points.

4.2. Scalability

An important observation in our experiments was the lack of scalability. If we are to deal with real-life execution traces we can not resort to matching every single call, as this requires (1) too many calculations, and (2) too much screen real-estate.

4.2.1. Data generation

One of the potential solutions to the data generation problem is to consider blocks of calls, i.e., to group a number of calls according to some criterion and to compare these blocks. The difficult part is to come up with a suitable selection criterion: simply considering mutually exclusive blocks of fixed numbers of calls is dangerous, as it potentially separates repeating call sequences. To devise a selection criterion we will examine the role of *stack depths* during the execution, and investigate whether depth changes can offer hints in selecting suitable call blocks.

Another solution is the optimization of the comparison operations themselves. By calculating hashes for the trace events and storing these hashes in hash buckets, the comparison process becomes significantly faster as string comparisons are no longer necessary.

4.2.2. Sampling criterion

While the sampling experiment provided good results, we suspect that this technique would benefit from more elaborate sampling criteria. The criterion used so far involves the consideration of every n -th event; another could be to consider groups of n events and to calculate a mean value for each group, or to introduce a minimum threshold for the amount of events in a group that satisfies the matching criterion.

4.2.3. Visualization

In order to visualize the potentially large amounts of data points that result from the comparison process, we need abstraction techniques to visualize the information in a meaningful way. Various techniques from the domains of information visualization and computer graphics (e.g., mipmapping, interpolation, or event clustering) can be used to handle such larger visualizations.

5. Conclusions and Future Work

In this position paper we have elaborated on a technique to visualize similarities in program executions. By comparing an execution trace to itself on an event-by-event basis and by showing the matches in a similarity matrix, little effort is required from the viewer to detect repeated call sequences and to determine the degree of similarity in certain execution phases. Moreover, such visualizations can lead to a greater understanding of a program's execution phases and occurrences of polymorphism. We conducted preliminary experiments that pointed out the issues that are to be tackled for our technique to be useful in practice.

Having overcome these issues, we have several future directions for our research. As a first direction we seek to investigate the *value* of information on repeated sequences, e.g., to find out how information on such sequences can be utilized to (automatically) shorten traces. Among the example applications are the dynamically reconstructed sequence diagrams from our earlier work [2] which could be rendered more compact.

Secondly, we want to determine to which extent the matrix visualization allows for the detection of phases during the execution, as was done using an alternative visualization in earlier work [3]. As a potential solution to the scalability issue, we will investigate and propose suitable sampling

criteria.

Finally, another application that we consider to be useful is the visualization of polymorphism. The matching criterion can be adjusted such that polymorphic calls are visualized, and the visualization of such occurrences may provide a deeper insight into the program's behavior and, by extension, its inputs and/or outputs.

Acknowledgments

This research is sponsored by NWO via the Jacquard Reconstructor project.

References

- [1] T. Ball. The concept of dynamic analysis. *ACM SIGSOFT Software Eng. Notes*, 24(6):216–234, 1999.
- [2] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman. Visualizing testsuites to aid in software understanding. In *Proc. 11th European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 213–222. IEEE, 2007.
- [3] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proc. 15th Int. Conf. on Program Comprehension (ICPC)*, pages 49–58. IEEE, 2007.
- [4] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. 3rd Int. Conf. on Software Maintenance (ICSM)*, pages 109–118. IEEE, 1999.
- [5] A. Hamou-Lhadj and T. C. Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls. In *Proc. 1st ICSE Int. Workshop on Dynamic analysis (WODA)*, pages 1–6, 2003.
- [6] A. Hamou-Lhadj and T. C. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proc. 14th Int. Conf. on Program Comprehension (ICPC)*, pages 181–190. IEEE, 2006.
- [7] D. Holten, B. Cornelissen, and J. J. van Wijk. Visualizing execution traces using hierarchical edge bundles. In *Proc. 4th Int. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 47–54. IEEE, 2007.
- [8] A. Kuhn and O. Greevy. Exploiting the analogy between traces and signal processing. In *Proc. 22nd Int. Conf. on Software Maintenance (ICSM)*, pages 320–329. IEEE, 2006.
- [9] S. P. Reiss. Dynamic detection and visualization of software phases. In *Proc. 3rd ICSE Int. Workshop on Dynamic analysis (WODA)*, pages 1–6, 2005. ACM SIGSOFT Sw. Eng. Notes 30(4).
- [10] J. F. Morar W. De Pauw, S. Krasikov. Execution patterns for visualizing web services. In *Proc. Symposium on Software Visualization (SOFTVIS)*, pages 37–45. ACM, 2006.
- [11] A. Zaidman. *Scalability Solutions for Program Comprehension through Dynamic Analysis*. PhD thesis, University of Antwerp, 2006.