Preprint of paper to appear in Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR'06)

Scalability Solutions for Program Comprehension through Dynamic Analysis

Andy Zaidman

Lab on Reengineering (LORE)
Department of Mathematics and Computer Science
University of Antwerp, Belgium
Andy.Zaidman@ua.ac.be

Abstract

Dynamic analysis has long been a subject of study in the context of (compiler) optimization, program comprehension, test coverage, etc. Ever-since, the scale of the event trace has been an important issue. This scalability issue finds its limits on the computational front, where time and/or space complexity of algorithms become too large to be handled by a computer, but also on the cognitive front, where the results presented to the user become too large to be easily understood.

This research focusses on delivering a number of program comprehension solutions that help software engineers to focus on the software system during their initial program exploration and comprehension phases.

The key concepts we use in our techniques are "frequency of execution" and runtime "coupling". To validate our techniques we used a number of open-source software systems, as well as an industrial legacy application.

1 Introduction

Any reengineering operation is preceded by a phase in which the software engineer is trying to become familiar with the system up to a level that allows him/her to accomplish the reengineering task in an orderly fashion. This process is generally termed *program comprehension*[2]. Gaining understanding of a system is an intensive process that according to studies can take up to 40% of the budget of the total reengineering operation [3].

Dynamic analysis, the (post-mortem) study of running software, has long been studied for the purpose of program comprehension [1, 8, 10]. Recently however, a renewed interest in this topic can be witnessed [4, 5, 6, 12, 13].

This PhD work focusses on providing solutions to developers wanting to acquaint themselves with an unknown software system in the shortest possible timeframe. Furthermore, the techniques we present are specifically targeted at the initial phases of the program comprehension process, when one is trying to find *hooks* for understanding the system. These hooks that can than be revisited, with more in-depth knowledge coming from targeted source-code reading or by using other program comprehension solutions.

A further non-functional requirement for our work is scalability, a typical pitfall of dynamic analysis solutions. As such, the techniques we present here should be considered as heuristics, where some work has been done to find an optimum for both recall and precision.

The results of our techniques are validated against a number of open source case studies that provide extensive documentation and one industrial case study.

2 Process and techniques

In this section we discuss two techniques that we have developed over the course of this research project. The first technique we discuss (Section 2.2) uses the relative frequency of execution of methods as a basis for determining closely related methods. The second technique uses runtime coupling measures to identify key classes in a system (Section 2.3).

Each technique was validated on two open source case studies, before being tested on an industrial case study. The industrial case study was graciously provided to us by our industrial research partner Kava[1]. It concerns a legacy C ap-

[1] http://www.kava.be, the Royal Pharmacists Union of Antwerp

plication, that at the time, was being ported from *UnixWare* to *Linux*.

Before discussing the two techniques (sections 2.2 & 2.3), we first give an overview of the dynamic analysis process we follow in order to apply both our techniques.

2.1 Dynamic analysis process

Applying both the techniques we discuss in this paper can be seen as a 4-step process. This section briefly explains these steps:

1. **Define execution scenario**
   Applying dynamic analysis implies that the software system is executed at least once. Choosing an execution scenario is of the utmost importance as it can heavily influence the resultset. In our case, we tried to balance *coverage* and *preciseness* of the execution scenario. Coverage means the degree of possible execution paths that are executed, while preciseness refers to the fact that we only execute those features that we need (or want) to understand. In terms of UML, this would be the same as limiting the number of use cases that are executed.
2. **Trace the application**
   Once the execution scenario is defined, the program is executed. Every *call* to and *return* from method-calls is logged in the event trace. The mechanism to log calls and returns differs depending on the case study. For our Java case studies, we used a custom-made JVMPI profiler[2], while for our industrial case study, which was entirely written in C, we relied on *Aspicere*, an aspect-weaver-framework for C [11].

The last two steps are technique-dependent and will be discussed in the subsequent sections.

3. **Apply technique**
4. **Interpret results**

2.2 Frequency of execution based

Thomas Ball [1] introduced the concept of "Frequency Spectrum Analysis", a way to correlate procedures, functions and/or methods through their relative calling frequency. The idea is based around the observation that a relatively small number of methods/procedures is responsible for a huge event trace. As such, a lot of repeated calling of procedures happens during the execution of the program. By trying to correlate these frequencies, we can learn something about (1) the size of the inputset, (2) the size of the outputset and —most interesting for us— (3) calling relationships between methods/procedures.

[2]Java Virtual Machine Profiler Interface. For more information see: http://java.sun.com/j2se/1.4.2/guide/jvmpi/jvmpi.html

In [13] we built further upon this idea, by proposing a visualization of the trace that allows for visual detection of parts of the event trace that show tightly collaborating methods. This technique was applied on two open source case studies, namely *Apache Tomcat 4.1.18*[3] and *Fujaba 4.0*[4].

Figure 1 shows a small part of this visualization. To be more specific, it concerns the execution of Tomcat. The figure resembles a "*heartbeat*" and should also be interpreted in that way. The X-axis is time, whereas the Y-axis is the dissimilarity between successive method-calls based on their frequency of execution. In the case of Figure 1 this regular heartbeat indicates a lot of repetitive method-calls. After close inspection, this repetitive region, consisting of 10000 method-calls, was revealed to be the iteration over a linked list with a certain operation on each element of that list.

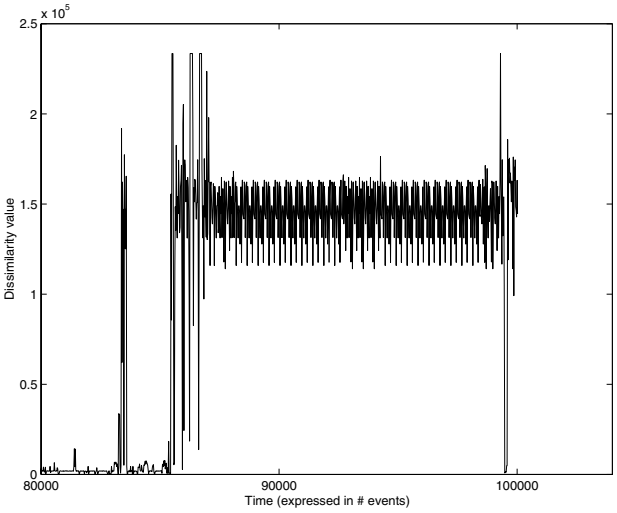

Figure 1. Visualization of a small part of the Tomcat trace.

For the two case studies above, the proposed visualization proved useful for navigating through the trace. Furthermore, it remained scalable for around the 10×10^6 method calls. However, when we tried to apply this exact visualization on the industrial case study from Kava that we had at our disposal, the visualization didn't scale. In this case study the event trace consisted of around 0.5×10^9 procedure calls.

Although we didn't manage to visualize this huge event trace in the style of Figure 1, we constructed an alternative visualization, which is shown in Figure 2. In this visualization, we tried to group together procedures (we are talking in terms of procedural languages for this case study) that

[3]http://tomcat.apache.org
[4]http://www.fujaba.de/

are executed the same number of times. Each box mentions the number of times the procedures that are contained in the box are executed and, additionally, the edge of each box also gives an indication of the level of cohesion between those procedures. For example:

- box (a) is *cohesive* as $\geq 50\%$ of the contained procedures originate from one module
- box (b) is *fully cohesive* as all contained procedures originate from the same module
- box (c) is *non-cohesive* as $< 50\%$ of the contained procedures originate from one module

This schema allowed us to not only quickly audit the system's structure, but also gain insight in wrapper-like solutions and procedures implementing low-level functionality [11].

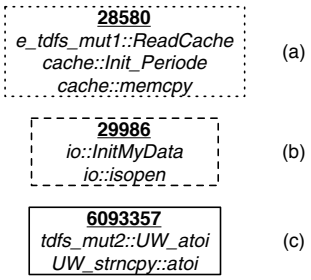

Figure 2. Frequency clusters.

2.3 Coupling based

The basic idea behind this technique is the fact that structural dependencies between modules of a system can indicate modules that are interesting for initial program comprehension [9]. As a measure we use runtime export coupling, which — provided we have a well-covering execution scenario — gives us all actual dependencies that happen at runtime. Modules which exhibit a high level of export coupling request other modules to do work for them (delegation) and often contain important control structures.

Coupling measures however are typically between two classes or modules, where we want to take into consideration the complete structural topology of the application. To overcome this strict binary relation between modules, we add a transitive measurement for reasoning over the topology. We use webmining techniques for this [12].

Webmining, a branch of datamining research, analyzes the topological structure of the web trying to identify important web pages based solely on their hyperlink structure. By interpreting call graphs as web graphs, we are able to retrieve important classes.

The HITS webmining algorithm [7] allows us to identify so-called *hubs* and *authorities* in (web) graphs. In terms of the Internet hubs are pages that mainly have a referring function, e.g. web directories, lists of personal pages, ... On the other hand, an authority contains useful and/or highly detailed information regarding a specific subject. In terms of program comprehension, hubs are modules that contain the core high-level logic of the application, while authorities are implementers of more low-level functions.

The resultset obtained from this heuristic is a list of all the modules of which containing procedures were executed during the scenario. These modules are ranked from being important to being irrelevant during early program comprehension phases.

A detailed description of this technique is available in [12]. The same paper also describes the validation of this technique on two open source case studies, namely Apache Ant 1.6.1[5] and Jakarta JMeter 2.0.1[6]. For these two case studies, validation was done by comparing the results obtained to extensive design documentation that was publicly available. For these two case studies our heuristic delivered a result with a recall of 90% and a precision of 60%. Although we are satisfied with the recall of 90%, we remain cautious over the fact that we have a reasonably low level of precision.

In [11] we applied this approach on an industrial legacy C system. In contrast to the open source case studies where we had to rely on documentation available on the internet, we were now able to validate the results we obtained with the original developers and current maintainers of the application. The results of this industrial experiment confirm the value of this technique.

3 Conclusion

The solutions we proposed during this research project can help developers and maintainers to (re)gain insight into the software they are developing, maintaining or extending. With a special focus on the early stages of program comprehension, they allow (1) to quickly determine sets of methods or procedures that work tightly together during a certain execution scenario and (2) to quickly identify those classes that are most interesting to investigate further for initial understanding purposes.

We validated our work with a number of object-oriented open source case studies, before applying it on a large industrial legacy application written in C. The open source case studies helped us demonstrate the effectiveness of the two techniques, when comparing the results against publicly available design documentation, while the industrial case study allowed us to assess the satisfaction of the developers and maintainers. Furthermore, the issue of scalability

[5] http://ant.apache.org
[6] http://jakarta.apache.org/jmeter/index.html

was also verified up to a certain point by applying it on a large-scale industrial case study.

4 Acknowledgements

I would like to thank prof. Serge Demeyer for giving me carte blanche when it comes to doing research, for giving precious advise and supporting me during this PhD work.

Andy Zaidman received support within the Belgium research project ARRIBA (*A*rchitectural *R*esources for the *R*estructuring and *I*ntegration of *B*usiness *A*pplications), sponsored by the IWT, Flanders.

References

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

[2] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *ICSE*, pages 482–498. IEEE, 1993.

[3] T. Corbi. Program understanding: Challenge for the 90s. *IBM Systems Journal*, 28(2):294–306, 1990.

[4] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *CSMR*, pages 314–323. IEEE, 2005.

[5] O. Greevy, A. Hamou-Lhadj, and A. Zaidman. Workshop on program comprehension through dynamic analysis (PCODA). In *WCRE*, pages 232–232. IEEE, 2005.

[6] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *CSMR*, pages 112–121. IEEE, 2005.

[7] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[8] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *ICSM*, pages 13–22. IEEE, 1999.

[9] M. P. Robillard. Automatic generation of suggestions for program investigation. *SIGSOFT Softw. Eng. Notes*, 30(5):11–20, 2005.

[10] T. Systa. Understanding the behavior of java programs. In *WCRE*, pages 214–223. IEEE, 2000.

[11] A. Zaidman, B. Adams, K. De Schutter, S. Demeyer, G. Hoffman, and B. De Ruyck. Regaining lost knowledge through dynamic analysis and aspect orientation — an industrial experience report. In *CSMR*. IEEE, 2006.

[12] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *CSMR*, pages 134–142. IEEE, 2005.

[13] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *CSMR*, pages 329–338. IEEE, 2004.