Analyzing large event traces with the help of a coupling metrics

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

Abstract

Gaining understanding of a large-scale industrial program is often a daunting task. In this context dynamic analysis has proven it's usefulness for gaining insight in object-oriented software. However, collecting and analyzing the event trace of large-scale industrial applications remains a difficult task. In this paper we present a heuristic that identifies interesting starting points for further exploratory program understanding. The technique we propose is based on a dynamic coupling metric, that measures interaction between runtime objects.

Keywords:
Reverse engineering, program comprehension, dynamic analysis, dynamic metrics

1 Introduction

Every software engineer has been in the position that he has to familiarize himself with the ins and outs of a piece of software in the shortest possible time. Most often, this is a daunting task and estimates go as far as stating 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 [12, 13].

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, the kind of system, ... [6] Because of this it is difficult to imagine that there exists one tool for all program understanding purposes. Ideally, a program understanding tool should *guide* the software engineer in his exploration process through the software [3]. When building program understanding tools, three strategies come to mind: pure static analysis, pure dynamic analysis or a combination of both. In the context of object-oriented software however, static analysis has proven to be inadequate to gain meaningful insight into the behavior of the application due to the late binding mechanism that's present in object-oriented systems [14]. However, dynamic analysis also has a major drawback: the huge amount of data that has to be analyzed, since even small applications generate tens of thousands of events (see Table 1). The scalability of the analysis algorithm is thus of the utmost importance [7, 11]. Because of the human cognition process, program understanding can never be a fully automated process: the user should be free to explore the software, with the help of specialized tools. Our aim is to develop a technique which gives the software engineer – *the user* – clues as to where to start his or her program understanding process. These clues consist of medium to high-level domain concepts from where the user can dig deeper into the code and/or dynamic behavior of the application.

	Jakarta Tomcat 4.1.18	Fujaba 4
Execution time (without tracing)	48s	70s
Classes	3 482	4 253
Events	1 076 173	772 872
Unique events	2 359	95 073

Table 1. Size of an event trace of two medium-size programs.

To reach this goal we will develop a heuristic based on a dynamic coupling metric. This heuristic will show us some medium to high-level domain concepts which can then serve as starting points for exploratory program understanding.

2 Scalability solutions

In order to overcome scalability issues a lot of techniques are presented as being *iterative*, meaning that the user has to repeatedly perform the technique on smaller sets of data, e.g. [9]. Scalability, however, is not only a computational issue: from a human standpoint the amount of analyzed data presented to the software engineer should be manageable [17].

Recent research has come up with two possible solutions:

- A novel solution has been formulated by Hamou-Lhadj and Lethbridge [4]. They represent the event trace as a tree in which they search neighbouring isomorphic subtrees. Identical neighbouring subtrees are pruned and replaced with a single occurrence which gets annotated with the total number of occurrences of the subtree. However, the problem of finding all isomorphic subgraphs in the tree is NP-complete [8], a problem referred to as the *subgraph isomorphism problem*. Their solution here is to set a minimum threshold for a certain subtree to be considered as being a candidate for exploratory program understanding.
- In [16] we explained a heuristic based on the frequency of execution of individual methods. The idea is based on the fact that methods work together to reach a common goal (e.g. accomplish a certain functionality). Thus, these methods are related through their frequency of execution [1]. We've engineered a heuristic which searches for and displays regions in the trace which contain similar trajectories in the frequency-time space. For finding this trajectories human intervention is required, so this is not a fully automated technique.

3 Dynamic metrics

Metrics have originally been designed to measure quality of (object oriented) code. These metrics are calculated from data that can be found directly in or can be extracted from the source code [2]. A few well-know metrics are: cyclomatic complexity measure, coupling between objects (CBO), lines of code (LOC), ...
Another field in which metrics are used is research in the area of clone detection. Here metrics serve as a way of finding regions of duplicated sourcecode. In this scenario the value of certain metrics is used to take *fingerprints* of regions of sourcecode. These fingerprints are then used to determine identical regions in the sourcecode [10].

Dynamic metrics, on the other hand, are often more precise because they give an image of what *is happening* (dynamic) versus what *may happen* (static) [15]. In this context Hitz et.al. [5] mentioned the difference between *class level coupling* and *object level coupling*. The former being a static metric, while the latter is a dynamic measure.

3.1 Coupling metrics

In terms of quality, *cohesion* within a class is desirable, while *coupling* between classes is undesirable. This stems from the idea that a class should be responsible for handling its own data. In practice however, coupling is unavoidable as classes collaborate in certain ways in order to perform the application functionality.

Within our problem-domain, coupling on a dynamic level has some interesting properties: because each instantiation of a collaboration between classes will exhibit the same level of coupling, the dynamic coupling metric becomes interesting for taking fingerprints of interaction patterns that exits between collaborating classes.

4 Implementation

4.1 The dynamic coupling metric

From a technical point of view, we will use the **Export Object Coupling** metric (EOC) to calculate the dynamic coupling [15] (see Equation 1). For understanding this formula, some helpful definitions are:

- o_i: is an instance of a class (an object)
- O: is the set of objects involved in a particular run of the program
- $M(o_i, o_j)$: is the set of messages sent from object o_i to object o_j during the program run
- MT: is the total number of messages exchanged between all objects in O

$$EOC(o_i, o_j) = \frac{|\{M(o_i, o_j) | o_i, o_j \in O \wedge o_i \neq o_j\}|}{MT} \times 100 \tag{1}$$

Calculating the EOC for each participating object results in a matrix of coupling-values, as can be seen in Table 2. For program understanding purposes this information is perhaps

$$\begin{array}{c|ccc}
 & object_1 & ... & object_n \\
\hline
object_1 & coupling_{1,1} & ... & coupling_{1,n} \\
... & ... & & ... \\
object_n & coupling_{n,1} & ... & coupling_{n,n}
\end{array}$$

Table 2. Coupling matrix

too detailed: it would be far more interesting to know how many unique messages a certain object has sent. For this, we can revert to the **Object Request For Service** metric [15]. Equation 2 gives us the total number of (unique) messages that object o_i has sent during the program run.

$$OQFS(o_i) = \sum_{j=1}^{K} EOC(o_i, o_j) \qquad (2)$$

This leaves us with a simplified version of the matrix from Table 2 in which only one column remains.

4.2 Possible interpretations

For our exploratory program understanding process we are mainly interested in classes which have a lot of responsibilities, i.e. classes which tell other classes to perform a certain action. Because of polymorphism and late binding, a certain class can issue different messages depending on e.g. the dynamic type of a parameter of a method. With this in mind, we can choose from two strategies to interpret the metric:

1. We can simply take the objects with the highest OQFS value and start our exploratory program understanding process there.
   A benefit of choosing this strategy is that other instances from the same class that exhibit different behavior will be listed separately.
   On the downside we have to note the fact that there is no easy way of removing duplicates.

2. On the other hand, we can build in another abstraction-step by not looking at individual objects, but instead at classes. If we were to change the formula for OQFS in Equation 2 into:

$$CQFS(c_i) = \sum_{j=1}^{K} |\{M(o_i, o_j)| \; o_i \; instance \; of \; c_i \wedge o_j \in O \wedge o_i \neq o_j\}| \qquad (3)$$

   We refer to this as the **Class Request For Service** (CQFS) metric. Basically it registers every message that the instantiations of a certain class sends during the execution of the program. Notice however, that duplicate message-sends are eliminated because we work with sets.
   Benefit of this strategy is the fact that duplicate instantiations from the same class are abstracted.
   A negative side to this technique is the fact that different objects that react differently due to the polymorphic behavior also get abstracted.

4.3 Which interpretation to choose

	duplicates	different actions due to polymorphism
OQFS	present	visible
CQFS	removed	abstracted

Table 3. Overview of the OCFS versus the CQFS metric

When comparing the OQFS with the CQFS metric (see Table 3, we have to keep in mind that the software engineer that uses our heuristic is trying to find clues as to where to start his software understanding exploration. As such, he's concerned with trying to find a class that has a high degree of responsibilities. When found, he can dig deeper into the code and/or zoom into sequence diagrams where this class is involved.
Thus, the fact that he is actually working with an abstraction isn't a problem for his program understanding process. Because of this, the CQFS metric seems to be more appropriate. However, experiments should validate this hypothesis.

4.4 "God classes"

The term ”god class” is used when a class performs most of the work of the system, relegating other classes to minor supporting roles. Both the OQFS and the CQFS metric will not focus particulary on such god classes, because:

- When looking at Equations 3 and 2 we see that the message sender and the message recipient cannot be the same: $o_i \neq o_j$

- Moreover, when a god class consists of *god methods*, these god methods will also show up only once in our analysis.

This means that although god classes won't disturb the heuristic, the analysis will remain largely incomplete for any systems that consists only of god classes.

4.5 Complexity

We already mentioned that a solution should before all be scalable. In order to get a complete analysis of the running program, we have to collect the dynamic coupling metric at runtime or calculate it post-mortem from the event trace.

Time complexity For both the OQFS and the CQFS metric we have to go over the entire trace once in order to calculate the coupling metric. The one step that remains to be done afterwards is to extract respectively the highest-scoring objects and classes. This means that the time complexity is **linear** in the number of events *n*, hence $O(n)$.

Memory complexity To compute the memory complexity, we first have to establish what data we have to keep track of:

- In the case of OQFS: each object that is created at runtime.
- In the case of CQFS: each class that is used during the program run.
- Which messages have already been accounted for in the metric calculation

This leaves us with a worst-case memory complexity of:
OQFS: $O(\#objects \times \#objects)$
CQFS: $O(\#classes \times \#objects)$
Because an object creation is considered as a message, we can put an upper bound on the number of objects with the total number of messages sent *n*. In general terms, the memory complexity is much more of an issue than the time complexity when analyzing large projects.

5 Conclusion and future work

We've presented a heuristic that searches for objects or classes that are candidates for starting exploratory program understanding. We base ourselves on the fact that classes which have more than average responsibilities have a greater coupling compared to other classes.

Our short-term goals are to validate the heuristic we've presented in this paper. The experiments we wish to set-up should also verify our hypothesis that it is better to abstract the metric up to the class-level, instead of remaining at the object-level.

References

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

[2] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactons on Software Engineering*, 20(6):476–493, 6 1994.

[3] M. A. Foltz. Dr. jones: A software archaeologist's magic lens. http://citeseer.nj.nec.com/457040.html.

[4] A. Hamoe-Lhadj and T. C. Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls, 2003. Paper presented at the Workshop on Dynamic Analysis (co-located with ICSE'03).

[5] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of the International Symposium on Applied Corporate Computing*, 1995.

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

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

[8] K. Mehlhorn. *Grahp Algorithms and NP completeness*. Springer Verlag, 1984.

[9] 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.

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

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

[12] F. Van Rysselberghe and S. Demeyer. Evaluating clone detection techniques. In *In proceedings of the International Workshop on Evolution of Large Scale Industrial Applications (ELISA)*, pages 25–36, 2003.

[13] 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.

[14] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1038–1044, 1992.

[15] S. M. Yacoub, H. H. Ammar, and T. Robinsion. Dynamic metrics for object oriented designs. In *Sixth IEEE International Symposium on Software Metrics*, pages 50–61. IEEE, 1999.

[16] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2003. To Appear.

[17] I. Zayour and T. C. Lethbridge. Adoption of reverse engineering tools: a cognitive perspective and methodology. In *Proceedings of the 9th International Workshop on Program Comprehension*, pages 245–255, 2001.