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, ... [8] 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 [6].

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 [9, 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.

To reach this goal we will develop a heuristic based on a dynamic coupling metric. We will consider the most tightly coupled classes as *starting points* for exploratory program understanding. As these tightly coupled classes have *authority* in the system, i.e. they tell other classes what to do, they can be considered as ideal starting points.

2 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 [4]. A few well-know metrics are: cyclomatic complexity measure, coupling between objects (CBO), lines of code (LOC), ...

Calculating metrics for object-oriented systems, how-

	Apache Ant 1.6.1	Jakarta JMeter 2.0.1
Classes (traced)	127	189
Classes (total)	1 216	245
Lines of Code (LOC), total	98 681	22 234
Events	24 270 064	138 704
# objects @ runtime	18500	4180
Scenario	building Ant	1 simulation run
Execution time (without tracing)	23 s	82 s

- *The number of events as shown in Table 1 takes into account both method entries and exits. As thus, the number of method invocations is actually 50% of this number.*
- *What stands out is the big difference in the number of events between Ant and JMeter even though the execution time for the latter is much longer. This can be attributed to the fact that: (1) JMeter uses a lot of Java base classes for e.g. network-related functionality and (2) network-related operations can take a relatively long time due to the uncertain network conditions.*

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

ever, is a tedious task. The presence of polymorphism and late binding makes that a static metrics tool, i.e. a tool that only uses the source-code, cannot precisely determine the actual targets for a certain method call.

Consider a two-level class hierarchy with a single base class and a number of derived classes. When statically there is a reference to the base class, but the actual run-time type is that of one of the derived classes, the actual target is the method from the derived class. For this reason, statically, the possible targets are the base class and all derived classes.
However, within the compiler optimization community, researchers have observed that normally only a small fraction of the possible targets get called [1]. Thus the actual coupling is far less than what can be inferred from source-code, hence traditional coupling metrics inaccurately reflect polymorphism as it is used in well-designed object-oriented programs. For that reason, coupling metrics that are extracted from the run-time behaviour of applications have been defined [2].

There are also situations in which the actual coupling is less that what can be determined from source-code. Situations in which there is dead-code, i.e. code that is *never* executed, or code that is not executed in the considered execution scenarios, is also counted in static metric tools. This is not the case when using a dynamic coupling metric [15].

In the next section we will introduce a dynamic coupling metric called *Class Request For Service*.

3 Class Request For Service (CQFS)

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

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

Calculating the EOC for each participating object results

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

Table 2. Coupling matrix

in a matrix of coupling-values, as can be seen in Table 2. For program understanding purposes this information is too detailed: it is 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.

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

This leaves us with a simplified version of the matrix from Table 2 in which only one column remains. The result-set from Equation 2 still remains large. Considering the two case studies mentioned in Table 1 applying the OQFS technique we would still end up with 18500 and 4180 results for respectively Apache Ant and Jakarta JMeter. For understanding purposes this does not scale up at the cognitive level [17].

Furthermore, in the case of program understanding considering the class-level coupling instead of the object-level coupling is more intuitive for the end-user because of the direct relation between the results from the heuristic and the actual source-code. As such, we've developed the **Class Request For Service** metric.

$$CQFS(c_x) = \sum_{j=1}^{K} |\{M(o_i, o_j) | o_i \rightarrow c_x \wedge o_j \rightarrow c_y \\ \wedge o_i, o_j \in O \wedge c_x, c_y \in C \wedge c_x \neq c_y\}| \quad (3)$$

Applying this metric will count every method that a certain class calls during the execution of the program. Notice, that due to the fact that we are working with sets, calling the same method more than once, will not increment the coupling count.
Furthermore, the condition $c_x \neq c_y$ eliminates cohesion. We explicitly discount cohesion as we are interested in inter-class relationships and not in intra-class inner-workings.

4 Discussion

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.

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, i.e. different instances from the same class that have an identical behavior.

In the case of CQFS, notice that duplicate message-sends are eliminated because we work with sets. In the context of program understanding, this can be seen as an advantage, as the user gets presented with less information. On a negative note, we can say that some interesting forms of polymorphism are abstracted away. Take for example two instantiations of the same class. Due to a different parameter of one of the methods of these objects, the objects themselves can react differently due to polymorphism. Using CQFS, this behavior gets abstracted away.

4.1 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 Related work

Recent research has come up with a number of possible solutions:

- A novel solution has been formulated by Hamou-Lhadj and Lethbridge [7]. 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 [10], 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 [3]. 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.

- Ducasse et Al present a visualization technique called *polymetric views* [5]. These polymetric views can best be described as a variant of a class diagram in which the height, width and color of the class are measures of certain metrics. In this case, these dynamic metrics can be the (1) total number of calls, (2) number of invocations where the caller and receiver are the same and (3) number of calls where the receiver is not the same as the caller.

6 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.

We furthermore want to establish the preciseness of using static or dynamic coupling metrics as helping tools for exploratory program understanding. As such, we want to know from the the original developers of a system which classes they consider important and then verify it with the results of the coupling metrics.

References

[1] G. Aigner and U. U. Hölzle. Eliminating virtual function calls in c++ programs. In *Proceedings of the 10th European Conference on Object-Oriented Programming*, pages 142–166. Springer-Verlag, 1996.

[2] E. Arisholm, L. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(7), July 2004.

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

[4] 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.

[5] S. Ducasse, M. Lanza, and R. Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 309–318. IEEE, 2004.

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

[7] 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).

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

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

[12] D. Spinellis. *Code Reading: The Open Source Perspective*. Addison-Wesley, 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)*, pages 329–338. IEEE, 2004.

[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.